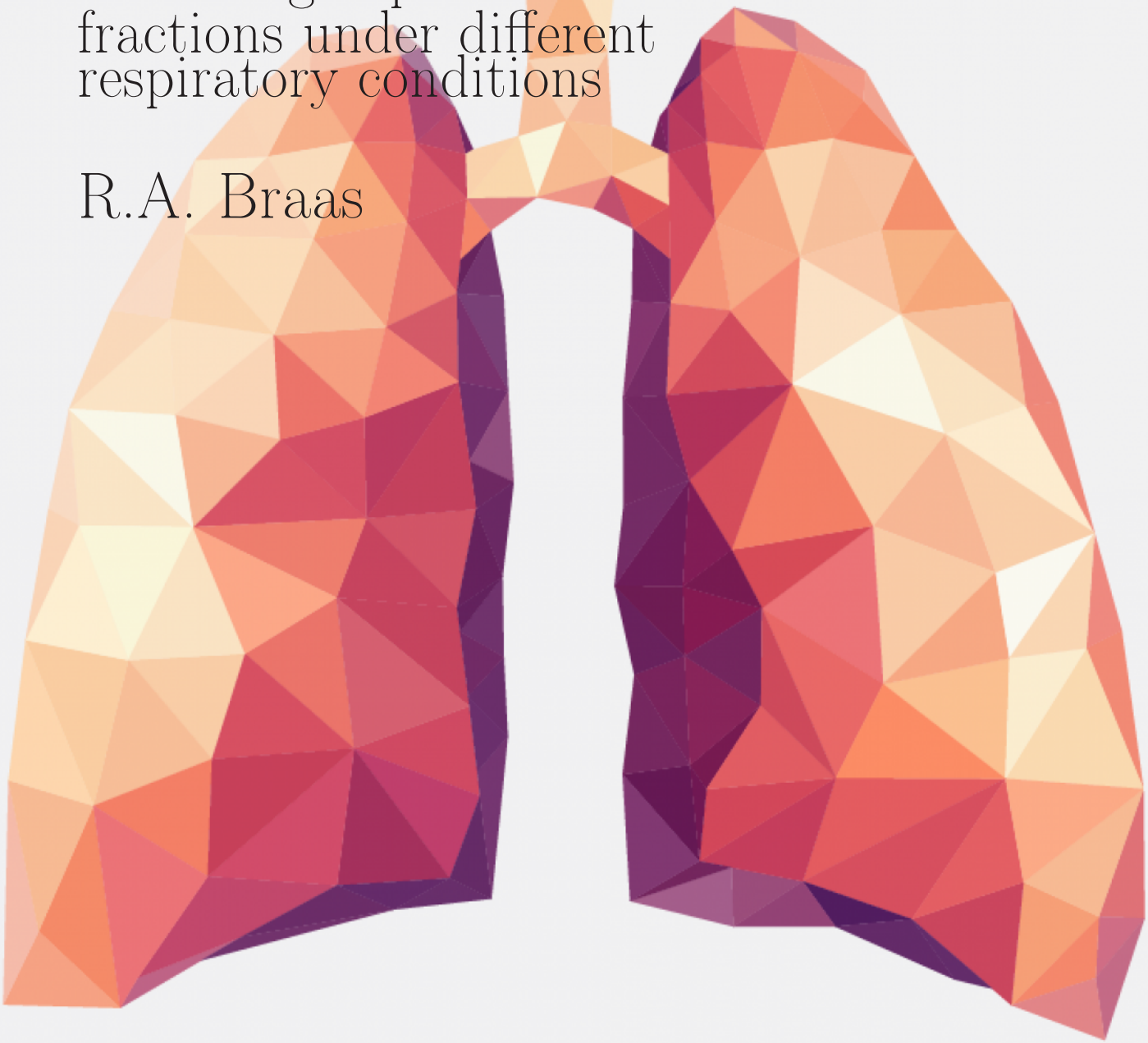
# Aerosol dynamics in human lungs

## Modelling deposition fractions under different respiratory conditions

R.A. Braas

Delft University of Technology



TU Delft

# Aerosol dynamics in human lungs

## Modelling deposition fractions under different respiratory conditions

by

R.A. Braas

to obtain the degree of Bachelor of Science

in Applied Mathematics,

and Applied Physics,

at the Delft University of Technology,

to be defended publicly on Monday July 20, 2020 at 09:00.


| | | |
|---|---|---|
| Student number: | 4449215 | |
| Project duration: | September, 2019 – July, 2020 | |
| Thesis committee: | Prof.dr.ir. C. Vuik | TU Delft, supervisor |
| | Prof.dr. S. Kenjereš | TU Delft, supervisor |
| | Dr.ir. R. van der Toorn | TU Delft |
| | Assoc.Prof.dr. J.M. Thijssen | TU Delft |


An electronic version of this thesis is available at `https://repository.tudelft.nl/`.

TU Delft Delft University of Technology

# Abstract

Knowledge of particle deposition is important in clinical settings or when discussing environmental effects of aerosols on humans. Particle deposition in the human respiratory tract is determined by breathing patterns and lung morphology, as well as particle properties and deposition mechanisms. In this study we develop a 1-dimensional model that numerically solves the general dynamic aerosol equation in the human respiratory tract. The model can be used to calculate deposition fractions for a range of initial parameters. We use Weibel's morphometric model to describe the lung geometry.

The model is validated by comparing it with previous numerical results, and running sensitivity tests to examine its consistency with parametric variations. The model proved to be computationally efficient, requiring just seconds to run a simulation. We use this to perform a number of parametric studies, most notably changing the tidal volume and the breathing rate. For both of these, an increase in either the volume or the rate decreased the deposition fraction across the spectrum of particle sizes, apart from at the tails of the distribution. We also examine the effect of particle density on the deposition fraction, which increases with an increasing density. The source code is published along with this thesis, allowing anyone to perform arbitrary parametric studies of their own.

# Contents

# 1

# Introduction

Aerosols are by no means a modern invention. They have been around since before humans inhabited the earth, and indeed, the English language has a lot of words to describe aerosol groups: dust, mist, fumes, smoke, etc. Aerosols are small particles suspended in air or other gases, so the term not only encompasses the environmental phenomena described above, but also deodorants or cough droplets. People have studied aerosols for a long time, but recently, they have gained the eye of the public when discussing the environmental effects of soot in large cities, or when discussing the infectiousness of COVID-19 through aerosols.

The focus of this study is to create a computational model of aerosol deposition in the lungs. Knowledge of aerosol deposition and distribution in the lungs is of importance when studying environmental effects, or in clinical settings. There is a lot more flexibility in computational models as opposed to empirical studies of aerosol deposition - it is possible to tweak every parameter to perform parametric studies on this deposition fraction. The drawback is of course, that numerical models need extensive empirical verification before the results are deemed trustworthy enough. But where the application of empirical models is restricted to the particular experimental conditions for which the model was tested, numerical models can vary the lung conditions, physiology of the patient, and even the morphology of the patient's lungs. It is suddenly trivial to model any type of patient, where in a lab setting this could be more difficult to control.

The flexibility of computational models extends to more than just parametric variation. Many considerations must be made in their creation, such as choosing whether to model 1 or 3 dimensions; choosing a stochastic or deterministic approach; which physical effects to take into account; which numerical method to use; up to even choosing to model the particles using Eulerian or Lagrangian mechanics. The present study attempts to recreate a one dimensional model developed by Mitsakou, Helmis, and Housiadas [1]. Current 3-dimensional models have the advantage of providing a complete description of aerosol dynamics, but have the disadvantage of requiring a lot of computational resources. The 1-dimensional approach allows us to quickly determine the deposition ratios for particles of different sizes under a range of respiratory conditions.

We will use this model to examine the effect of various parameters on the deposition fraction. Most notably, we will vary the breathing conditions like the tidal volume and the breathing rate, to see the effect on the total absorption. This will be used to shine light on the infectiousness of COVID-19 through aerosols and the health hazards of smoke inhalation during the bushfires

that devastated Australia in the summer of 2019-20.

Chapter 2 introduces the mechanics behind aerosol dynamics, and a description of lung morphology and lung models. The numerical method used to model the aerosol dynamics is designed in Chapter 3, along with an analysis of its convergence. After the relevant theory and numerics are introduced, validation of the model will be treated in Chapter 4. Extensive verification is needed to ensure that the model accurately describes aerosol deposition under a range of conditions, which will be the subject of Chapter 5. Here, we perform parametric studies to analyse the effect of different breathing conditions on the deposition fraction. Finally, Chapter 6 summarises the findings and provides some recommendations for further research.

# 2

# Theory

The goal of this chapter is to introduce the underlying theory behind calculating aerosol deposition in the lungs. We first introduce the lung model, after which we will discuss the behaviour of aerosol dynamics. Calculations of deposition fractions and the velocity field is discussed next, and this chapter concludes by specifying various constants used in the model, and introducing the medical terminology required to understand the parametric studies in Chapter 5.

## 2.1. Lung model

The model used in this research is the Weibel model "A" [2], which describes the lungs as a series of bifurcations. Lungs are composed of a series of tubular branches, starting with the trachea (windpipe), and ending with the alveoli. The Weibel model has 23 generations: the trachea (generation 0) splits up into two bronchi (generation 1), and this process continues all the way up to generation 23 (alveoli). The model is symmetric, and therefore easy to describe using a 1-dimensional model.

A schematic of the model is shown in Figure 2.1. The model consists of purely conducting airways for the first 16 generations. Generation 16 marks the start of the transitional section, and from generation 16 onward, the lungs become alveolated and contract and expand due to breathing mechanics. The size of the later generations is very small, but because the model bifurcates at every generation, the number of alveolar sacs in the last generation is 8 388 608. This also means that the volume and surface area of the lungs greatly increases towards the end. This is why this type of model is also known as the "trumpet" model, and an illustration of this can be found in Figure 2.2.

The parameters that define the Weibel model are the generation length, the generation diameter, the branching angles, the gravity angles, and the number of airways per generation. Both the generation length and the diameter decrease with the generation number, while the number of airways per generation obviously increases. The branching angle is the angle that the next generation makes with the previous, and it varies from 20° to 30° for the first generations, and is around 45° for the later generations. The gravity angle is the angle the airway makes with the gravity vector (when standing upright) and ranges from about 40° to 60°. In this research, the data for the Weibel model "A" is obtained from [3] and is included in Appendix A.

Note that this model does not account for the nose or mouth and its accompanying structure. If the nose or mouth need to be modelled, it is possible to add a generation before generation 0,

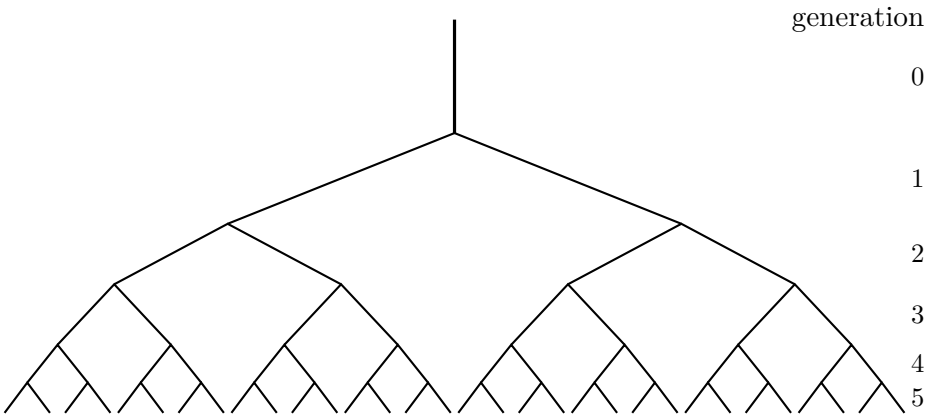



Figure 2.1. Schematics of hierarchy by Weibel's model. The bifurcations continue until the 23rd generation.

as has been done in [1] for example.

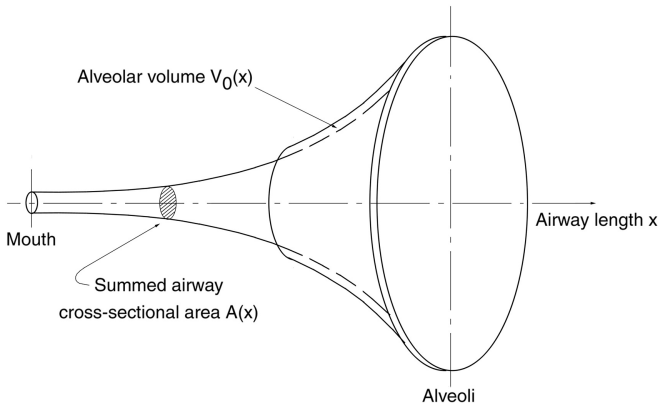



Figure 2.2. Illustration of the one-dimensional "trumpet" model. Image taken from [4].

## 2.2. Aerosol dynamics

Aerosol dynamics are governed by a number of physical processes, and we can mathematically model these to get a description of the complete particle distribution. The models describing these dynamics can be split into three different groups: particle movement, particle-surface interaction, and particle-particle interaction. The main processes describing particle movement are Brownian diffusion and advection, which describe how particles move en masse. Brownian diffusion also describes how smells disperse through the air, whereas advection is usually associated with wind. Apart from these two main processes, particle movement can also arise from electric fields or temperature gradients.

The second group, particle-surface interaction, models the situation in which particles either deposit on a surface, or release from it. In this research, we are particularly concerned with particle deposition on the walls of the lungs.

The third group is comprised of processes with particle-particle interactions. This is primarily coagulation, where aerosols collide with each other to form larger aerosols (or split to form smaller ones). Other processes included are growth laws, in which other particles attach to the aerosol in a manner known as gas-to-particle interaction.

These groups of processes can be summarised in an equation known as the Aerosol General Dynamics Equation (GDE). This equation describes how aerosols behave as a group. The one-dimensional form of the equation reads [1]

$$\begin{aligned}\frac{\partial}{\partial t}(A_T q_i) = &- \frac{\partial}{\partial x}(A_A u q_i) + \frac{\partial}{\partial x}\left(A_T D_{\text{eff}} \frac{\partial q_i}{\partial x}\right) - V_d \Gamma q_i \\ &+ \left(\frac{\partial}{\partial t}(A_T q_i)\right)_{\text{coagulation}} + \left(\frac{\partial}{\partial t}(A_T q_i)\right)_{\text{growth}}.\end{aligned} \tag{2.1}$$

Our goal is to determine $q_i(x,t)$ for all $x$ (the position in the respiratory tract) and $t$ (the time coordinate). The index $i$ iterates over the different particle sizes, making it possible to model particles of different diameters, while including particle-particle interaction for particles of different sizes. The equation also includes $A_A$ and $A_T$, which are the airway cross-sectional areas (of all airways). The difference between the two terms is that $A_T$ includes a time dependency for the alveolated generations, which vary in diameter with time.

The first term on the right side of (2.1) is the advection term. This describes how particles behave under an external flow field. In our case, this external field is the velocity of the air $u$ traveling through our lungs due to breathing. It is the dominant force for aerosol migration, and it reverses sign every breathing cycle. Advection is modelled to occur only in the conducting part of the airway, which is why $A_A$ is used for the advection term (advection does not occur through the alveolar part).

The second term is the Brownian diffusion term. This arises from fluctuating forces exerted by surrounding molecules, and the random motion results in a net movement of aerosols against the direction of the concentration gradient. The strength of the diffusion is described by the effective diffusion coefficient: $D_{\text{eff}}$, and it also depends on the particle size.

The linear term $-V_d \Gamma q_i$ describes the deposition of particles into the lungs. Deposition is an interaction in which particles collide with the surface (the airway walls) and are absorbed. This is the term we are most interested in, and it depends on the deposition velocity $V_d$, which will be described in Section 2.2.1. The parameter $\Gamma = n\pi d_T$ is the wetted perimeter, which depends on the number of airways per generation $n$ and the airway diameter $d_T$. In essence, the amount of deposition depends on the perimeter of the airway, the deposition velocity and the local concentration of aerosols.

The coagulation term describes particles changing their respective particle size group by combining (or splitting) with other particles. It results from particle-particle interactions and is one of the ways the particle sizes are interdependent. Particle coagulation also leads to a reduction in the overall concentration of particles.

Unlike coagulation, which modifies the size distribution of aerosols, gas-to-particle conversion increases the aerosol mass concentration, while also affecting the size distribution. The latter can be modelled by so called "growth" laws and allows for calculating the changing size distribution function. There are a number of growth laws, for example: molecular bombardment, surface reaction, droplet-phase reaction. The growth law can be calculated using Mason's theory [5]. This process, along with particle coagulation, will not be considered in this research.

### 2.2.1. Deposition velocity

The deposition velocity is the velocity of the aerosols that deposit on the surface of the airways. It is comprised of three separate effects: gravitational settling, Brownian diffusion, and impaction due to curves in the lung bronchi and bronchioles. This is explained in detail in [1] and is included here for completion.

**Gravitational settling** The simplest of the three is the deposition due to gravitational settling. Like all objects in a gravitational field, aerosols are also affected by it, and it is written as follows:

$$V_s = u_s \sin\theta. \tag{2.2}$$

Here the velocity $u_s$ is determined from Stokes' law, and the angle $\theta$ is the angle the airways make with the gravity vector. We assume that the particle terminal velocity is reached instantly, and then the velocity $u_s$ follows from a balance of forces of the gravitational force and the Stokes' drag force [6]

$$0 = mg - 3\pi\mu_g d u_s \tag{2.3}$$

$$\implies u_s = \frac{\rho d^2 g}{18\mu_g}, \tag{2.4}$$

where we have assumed that the aerosols are spherical. In this equation, $\rho$ is the particle density, $d$ its diameter, $g$ is the gravitational field strength, and $\mu_g$ the fluid viscosity. This expression does not yet account for particle slipping, but this is easily rectified by multiplying the expression by the Cunningham correction factor, which accounts for slip effects for small particles [7].

**Brownian diffusion** The deposition velocity arising from Brownian diffusion can be found from mass transfer theory, and depends on Sherwood's number in the following manner [8],

$$V_d = \frac{D_B Sh}{d_T}. \tag{2.5}$$

Here the Brownian diffusion constant $D_B$ is given by the Stokes-Einstein equation [9], found by Einstein and Sutherland independently all the way back in 1905,

$$D_B = \frac{k_B T_g C_c}{3\pi\mu_g d}. \tag{2.6}$$

Here, $k_B$ is Boltzmann's constant, $T_g$ is the temperature of the fluid (air), and $C_c$ is Cunningham's correction factor; the other constants have already been introduced. Sherwood's number $Sh$ is found using an empirical relation for circular ducts. We use the same source for the empirical relation as [1]; however, they copied it incorrectly. The correct expression for Sherwood's number is shown below [10]

$$Sh = \begin{cases} 1.077(x^*)^{-1/3} - 0.7 & \text{for } x^* \leq 0.01, \\ 3.657 + 6.874(10^3 x^*)^{-0.488}\exp(-57.2x^*) & \text{for } x^* > 0.01. \end{cases} \tag{2.7}$$

Here, the value $x^*$ is the dimensionless length measured from the beginning of the airway (not the beginning of the respiratory tract). The dimensionless length depends on Reynolds and Schmidt's numbers in the following manner:

$$x^* = \frac{x}{d_T Re Sc} \tag{2.8}$$

Reynolds number is given by the standard expression $Re = \rho u d_T/\mu$. On the other hand, Schmidt's number is given by $\mu/\rho D_B$.

**Impaction** Finally, the last component of the deposition velocity stems from the fact that particles may not be able to follow the fluid path due to their inertia. This effect is greatest for large (>1 µm) particles [11]. Due to the different branching angles $\phi$ in the lungs, the impaction velocity is significant and is given by.

$$V_i = \begin{cases} 0 & \text{(over first 80\% of the airway length),} \\ Stk\, u\phi d_T/0.2L & \text{(over last 20\% of the airway length),} \end{cases} \tag{2.9}$$

where $Stk$ is Stokes number, and $L$ is the branch length. This expression is based on a formal analysis (instead of empirical modelling) and is described in detail in [1].

## 2.3. Derivation of the velocity field

To solve the GDE (2.1) we need to solve for the velocity field $u(x,t)$. This is done by solving the continuity equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u)}{\partial x} = 0. \tag{2.10}$$

Before deriving the continuity equation specific to our problem, we need to define how the lung volume varies with time. This is necessary because our cross sectional area $A_T$ varies with time, according to the lung volume.

Inspiration and expiration (inhalation and exhalation) can be modelled by a simple function $f(t)$ that varies from -1 to +1. For our purposes, we will use a sinusoidal function with a breathing rate of 12 breaths per minute, which is the normal respiratory rate for an adult [12]. The lung volume is then defined as follows,

$$V_L = \left(V_{\text{FRC}} + \frac{V_T}{2}\right) + \frac{V_T}{2} f(t). \tag{2.11}$$

It consists of a constant part, the functional residual capacity $V_{\text{FRC}}$, and a varying part that depends on the tidal volume $V_T$, and the breathing function $f(t)$. These terms will be explained in Section 2.6, and in particular, Figure 2.3. The cross sectional area $A_T$, or rather, the time dependent diameter $d_T$, relates to the lung volume as follows

$$\frac{d_T}{d_A} = \left(\frac{V_L}{V_{\text{FRC}}}\right)^{1/3}. \tag{2.12}$$

These equations specify $A_T$ for all $x$ and $t$, which we will use to calculate the velocity profile.

We can derive the continuity equation for our problem by considering how the total volume of air varies over a section in the RT. If the section in question starts at a point $x = a$ and ends at a point $x = b$, then the total volume of air varies with the flux over the section boundaries

$$\int_a^b \frac{\partial}{\partial t} A_T \, dx = F(a) - F(b), \tag{2.13}$$

where $F(x)$ is the flux function. Since advection only happens in the conducting part of the airway $A_A$ (not $A_T$), the flux function is $F(x,t) = u(x,t)A_A(x)$. Inserting this into (2.13) and simplifying yields,

$$\int_a^b \frac{\partial}{\partial t} A_T \, dx = -uA_A\Big|_a^b. \tag{2.14}$$

Solving this for $u$, we obtain

$$u(b,t) = \frac{1}{A_A(b)} \left[ u(a,t)A_A(a) - \int_a^b \frac{\partial}{\partial t} A_T(x,t) \, dx \right], \tag{2.15}$$

which holds for all $a$, $b$, and $t$. This last property allows us to calculate $u(x,t)$: we can simply fill in whatever $b$, because we already know $A_T(x,t)$ and $u(0,t)$. Indeed, the inlet velocity is simply $u(0,t) = \frac{\partial}{\partial t} V_L(t)/A_A(0)$. Note that in the entire derivation here, we have assumed a constant air density $\rho$.

Luckily for us, this calculation can be decoupled from the partial differential equation for the aerosol concentrations, because the "continuous" quantity in the continuity equation is the volume of air, which we assume to be incompressible, and that it does not depend on the concentration of aerosols. In this way, the velocity profile only depends on the breathing pattern specified, and can therefore be calculated ahead of time. In fact, it can be calculated analytically for most breathing functions. The numerical approximation will be treated in Section 3.5.4.

## 2.4. Calculating the deposition fraction

The deposition fraction is the ratio of the total number of particles introduced into the respiratory tract, and the amount of deposited particles. The former is easy to calculate: it is the influx of particles into the respiratory tract (RT), and since this only happens at the inlet (trachea), the expression is as follows

$$\text{total RT} = \int_0^{T/2} q(0,t) A_A(0) u(0,t)\, dt, \tag{2.16}$$

where $T$ is the period of the breathing cycle. Note that we only integrate over half of a period: the inspiration phase.

The total deposition is simply the deposition term of (2.1) integrated over $G$, which could be a generation or the entire respiratory tract, depending on what we're interested in. In this case, we do integrate over the entire period, because we want the deposition fraction over a whole breathing cycle. The total deposition is shown below:

$$\text{total deposition} = \int_0^T \int_G q(x,t) V_d(x,t) \Gamma(x,t)\, dx\, dt. \tag{2.17}$$

The deposition fraction is the ratio of the two previous expressions, as follows [1]

$$\text{deposition fraction} = \frac{\int_0^T \int_G q(x,t) V_d(x,t) \Gamma(x,t)\, dx\, dt}{\int_0^{T/2} q(0,t) A_A(0) u(0,t)\, dt}. \tag{2.18}$$

Several numerical integration methods are possible, in this case we use a standard library that implements the trapezoidal rule. The reason we choose this integration method instead of e.g. Simpson's rule, is because we require the integration to be independent of our starting and ending points. This will play a role when we are calculating local deposition (as opposed to total deposition). If we use the trapezoidal rule, then we ensure that local and total deposition calculations do not differ.

## 2.5. Aerosol parameters and other constants

We are primarily concerned with comparing the deposition rates of aerosols of different sizes. However, the particle diameter has an effect on a number of other aerosol properties. In this section, these properties are discussed as well as how they depend on the aerosol diameter.

First, the particle density ranges from about $0.5\,\mathrm{g\,cm^{-3}}$ to $3.0\,\mathrm{g\,cm^{-3}}$ [13]. We will take a standard value of $1.0\,\mathrm{g\,cm^{-3}}$ Note that this is the density of a single particle, and so it does not depend on the concentration of the particle in the fluid.

Next, we have the Cunningham slip correction factor. This correction reduces the amount of drag force on particles (because of slip effects). It depends on the particle diameter $d$ and the mean free path $\lambda$:

$$C = 1 + \frac{\lambda}{d}\left[2.34 + 1.05\exp\left(-0.39\frac{d}{\lambda}\right)\right]. \tag{2.19}$$

The Cunningham correction factor for a range of particle sizes is shown in Table 2.1.

Another factor that depends heavily on the particle diameter is the particle relaxation time. This characterises the time that a particle needs to adjust its velocity when exposed to a new environment. The shorter the time, the quicker the particle adjusts to new force conditions. Because of inertial effects, smaller particles tend to have a much shorter relaxation time than large particles, which tend to maintain their old path. The expression for the relaxation time is as follows

$$\tau = \frac{\rho d^2 C}{18\mu}. \tag{2.20}$$

The relaxation time is used for calculating Stokes' number, which we need to calculate the deposition velocity due to inertial effects.

In addition to particle-specific properties, we also need to take into account properties of the air in the lungs. The density of air is $1\,\mathrm{kg\,m^{-3}}$, and its viscosity is $18.1 \times 10^{-6}\,\mathrm{Pa\,s}$. The mean free path of air (used in the previous paragraphs) is $0.066 \times 10^{-6}\,\mathrm{m}$ [14]. Finally, we assume the temperature to be room temperature (293 K), and we discount warming up of the air in our calculations.

Table 2.1. Cunningham slip correction factor and the particle relaxation time for a range of particle sizes.

| Particle diameter [µm] | Cunningham slip correction | Relaxation time [s] |
|---|---|---|
| 0.01 | 23.0 | $2.12 \times 10^{-8}$ |
| 0.1 | 2.93 | $2.70 \times 10^{-7}$ |
| 1 | 1.15 | $1.06 \times 10^{-5}$ |
| 10 | 1.02 | $9.35 \times 10^{-4}$ |

## 2.6. Breathing patterns

Respiration generally refers to the absorption of $O_2$ and the removal of $CO_2$ from the body. Breathing functions to move air into and out of the lungs through the trachea, where it undergoes a gas exchange in the alveoli. During a normal respiratory pattern, the most important factor describing breathing mechanics is the tidal volume $V_T$. This is the amount of air inhaled or exhaled in one breath. After each breath, the lungs still contain a residual volume that cannot be voluntarily expirated, as well as some expiratory reserve volume that is the amount of tidal expiration that can be exhaled with maximum effort. Similarly, during normal breathing, there is some inspiratory reserve volume that can be inhaled with maximum effort. For an illustration of the different respiratory volumes refer to Figure 2.3.

For an average young adult male, the total lung capacity is 5900 ml, and the tidal volume during relaxed, quiet breathing is 500 ml [12]. The medical term for relaxed breathing is eupnoea. In this thesis, we will investigate breathing patterns during exercise, also known as hyperpnea. Breathing patterns like Biot's respiration and Cheyne-Stokes respiration that occur during drug overdoses or in a clinical setting will not be investigated. Likewise, hyperventilation and

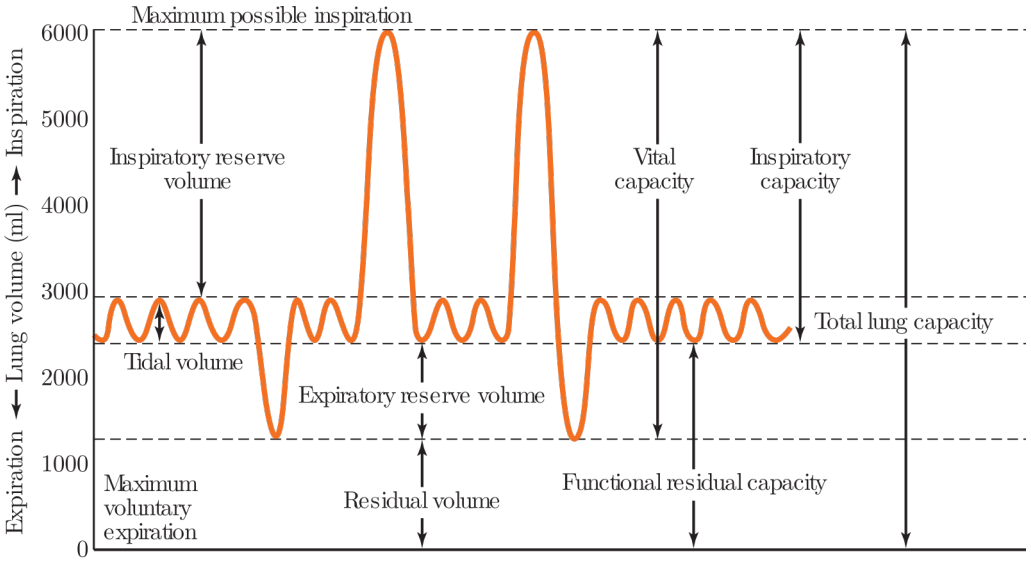



Figure 2.3. Respiratory volumes and capacities for an average young adult male [12].

hypoventilation will also not be considered, because although these situations may have a significant effect on the deposition fraction, they are also infrequent and of short duration.

Hyperpnea is the body's response to requiring more oxygen. It primarily results in taking deeper breaths, or an increase in the tidal volume, but it can also result in faster breathing. Usually, hyperpnea is a response to the body's activity or environment; for example, exercise or high altitude. We will both investigate deeper breathing as well as changing the respiratory rate.

# 3

# Numerics

## 3.1. Developing the numerical scheme

The goal of this chapter is to develop a numerical scheme that will solve the partial differential equation (2.1). We will solve this equation without the growth and coagulation terms. This means that we will drop the index $i$ from the solution $q(x,t)$, because we are taking the interaction between particle classes out of consideration. To simplify matters further, we will write the coefficients for the time derivative and the advection, diffusion and deposition terms as follows:

$$\begin{aligned}\alpha &= A_T,\\ \gamma &= A_A u,\\ \beta &= A_T D_{\text{eff}},\\ \rho &= V_d \Gamma.\end{aligned}$$

Thus, the equation we wish to solve is as follows,

$$\frac{\partial}{\partial t}(\alpha q) = -\frac{\partial}{\partial x}(\gamma q) + \frac{\partial}{\partial x}\left(\beta \frac{\partial q}{\partial x}\right) - \rho q. \tag{3.1}$$

Our numerical scheme will be based on the finite volume method, which divides the spatial domain into intervals. The solution $q$ will be estimated by averaging over this volume. In each time step, the solution will change due to the flux through the cell boundaries, and due to the linear term (the deposition term). Now let $Q_j^n$ be the approximation of the solution $q$ at the $j$th grid cell at time step $n$. Then this value will approximate the average solution as [15]

$$Q_j^n \approx \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} q(x,t_n)\,dx. \tag{3.2}$$

Integrating (3.1) in space gives

$$\frac{d}{dt}\int_{x_{i-1/2}}^{x_{i+1/2}} \alpha q(x,t)\,dx = f(q(x_{i-1/2},t)) - f(q(x_{i+1/2},t)) - \int_{x_{i-1/2}}^{x_{i+1/2}} \rho q(x,t)\,dx, \tag{3.3}$$

where $f$ is the flux function depending on the solution $q$. We can use this expression to develop a mixed implicit/explicit time marching algorithm. The mixed implicit/explicit scheme will be based on a modified version of the rectangle rule for integration,

$$\int_{t^n}^{t^{n+1}} \alpha q(x_j,t)\,dx = \Delta t[(1-\theta)q(x_j,t^n) + \theta q(x_j,t^{n+1})]. \tag{3.4}$$

This method, also known as the $\theta$-method, allows a parameter $\theta$ to vary between 0 and 1, where a value of 0 is a fully explicit method, and a value of 1 is fully implicit. When $\theta = 0.5$, it is known as the Crank-Nicolson method. Integrating in time from $t^n$ to $t^{n+1}$ and rearranging, yields

$$\begin{aligned}\alpha Q_j^{n+1} = \alpha Q_j^n + \frac{\Delta t}{\Delta x}\Big[&(1-\theta)\left(F_{j-1/2}^n - F_{j+1/2}^n - \Delta x \rho Q_j^n\right) \\ &+ \theta\left(F_{j-1/2}^{n+1} - F_{j+1/2}^{n+1} - \Delta x \rho Q_j^{n+1}\right)\Big].\end{aligned} \tag{3.5}$$

Here $F_{j-1/2}^n$ is an approximation to the average flux through $x_{j-1/2}$:

$$F_{j-1/2}^n \approx \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} f(q(x_{j-1/2}, t))\, dt. \tag{3.6}$$

Depending on the term (advection or diffusion), we will approximate this flux using the values of the neighbouring grid cells: $Q_{j-1}$ and $Q_j$. We can estimate this flux based on the values of $Q_{j-1}^n$ and $Q_j^n$. An illustration of the grid point system is shown in Figure 3.1.

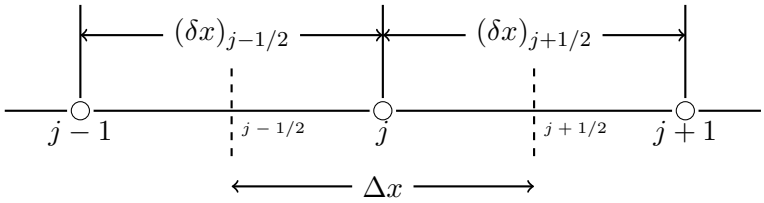



Figure 3.1. Grid-point cluster for the one-dimensional problem.

The flux $f(q)$ for our problem is

$$f(q_x, q, x) = -\beta(x) q_x + \gamma(x) q. \tag{3.7}$$

For the derivative $q_x$, we need to make a profile assumption. The simplest is a piecewise-linear profile, so that the derivatives on the cell edges are well defined (if $\beta$ is constant, then this reduces to the central difference scheme). Unlike the diffusion term, the advection term cannot be discretised using central differences, doing so can result in solutions that are not physically realistic [16]. Instead, the discretisation scheme used for the advection term is the upwind-scheme. In this way, the value of the advective property is equal to the value on the upwind side of the face.[1] Then, the flux $F_{j-1/2}^n$ can be defined as

$$F_{j-1/2}^n = -\beta_{j-1/2}\left(\frac{Q_j - Q_{j-1}}{\Delta x}\right) + \gamma_{j-1/2} Q_{j-1}. \tag{3.8}$$

Expressions for $F_{j+1/2}^n$, $F_{j-1/2}^{n+1}$, and $F_{j+1/2}^{n+1}$ can be found using a similar approach. Inserting these approximations for the flux into equation (3.5) yields (for the inspiration phase, and $\theta = 0$)

$$\begin{aligned}\alpha Q_j^{n+1} = \alpha Q_j^n + \frac{\Delta t}{\Delta x}\Big[&\left(-\beta_{j-1/2}^n\left(\frac{Q_j^n - Q_{j-1}^n}{\Delta x}\right) + \gamma_{j-1/2}^n Q_{j-1}^n\right) \\ &- \left(-\beta_{j+1/2}^n\left(\frac{Q_{j+1}^n - Q_j^n}{\Delta x}\right) + \gamma_{j+1/2}^n Q_j^n\right) - \Delta x \rho Q_j^n\Big].\end{aligned} \tag{3.9}$$

[1] Note that the upwind face changes depending on whether we are in the inspiration or expiration phase.

A similar expression can be derived for the arbitrary $\theta$ and for the expiration phase. The discretisation equation (3.9) leads to a system of linear equations, which can be put in matrix form (see Section 3.3). The equation describes how interior points depend on their neighbours; what remains to be done is determine equations for the end-points: the boundary conditions.

## 3.2. Boundary conditions

Boundary conditions specify how the solution behaves on the boundary values (the beginning and the end of the respiratory tract). There are two main types of boundary conditions that are of our concern: boundary conditions of the first-type (Dirichlet), and boundary conditions of the second-type (Neumann). Dirichlet boundary conditions specify the value of the solution on the boundary (e.g. the inlet concentration is $1\,\mu g\,m^{-3}$). Neumann boundary conditions specify the value the derivative of the solution on the boundary. In our case, the boundary condition on the right side (at the alveoli) is a Neumann boundary condition: the concentration gradient is zero at the end of the respiratory tract, meaning that no aerosols travel further than this.

Boundary conditions of the first-type pose no additional problems in the discretisation process because the concentration at the boundary is already known. The second-type is slightly more involved, because only the flux is known at the boundary. Consider the right boundary $j = N$ during the inspiration phase. The flux $F_N$ is given, and since $F_N \approx F_{N+1/2}$, we can insert this into (3.5) to obtain an equation for $Q_N$, without requiring a ghost cell $Q_{N+1}$.

The numerical model allows for a custom specification of boundary conditions. That is, it is possible to choose the boundary condition type on both sides. In our case, it is very much necessary that the boundary conditions are able to change. When applying the model on the Weibel lung geometry, the boundary condition at the trachea changes type when switching from expiration to inspiration or vice versa [1]. During inspiration, the trachea boundary condition is of the first-type with a specified concentration of the aerosol; during expiration, the flux is zero. Unlike the trachea, the boundary condition at the alveoli is always a second-type where the flux is zero.

## 3.3. Matrix form of the discretisation

We obtained a discretisation equation in Section 3.1. For each time step $t^n$, there is a system of linear discretisation equations. These equations can be written in matrix form. If we denote $\boldsymbol{Q}^n$ as the vector containing the solution at the grid points $\{x_0, x_1, \ldots\}$ at a time step $t^n$, then the matrix form the the discretisation reads

$$\boldsymbol{Q}^{n+1} = \boldsymbol{Q}^n + \frac{\Delta t}{\Delta x}[(1-\theta)(\mathbf{A}^n\boldsymbol{Q}^n + \mathbf{b}^n) + \theta(\mathbf{A}^{n+1}\boldsymbol{Q}^{n+1} + \mathbf{b})]. \tag{3.10}$$

The system is linear, so it is actually possible to solve this for $\boldsymbol{Q}^{n+1}$ directly, as the derivation below shows:

$$\begin{aligned}
\boldsymbol{Q}^{n+1} &= \boldsymbol{Q}^n + \frac{\Delta t}{\Delta x}[(1-\theta)(\mathbf{A}^n\boldsymbol{Q}^n + \mathbf{b}^n) + \theta(\mathbf{A}^{n+1}\boldsymbol{Q}^{n+1} + \mathbf{b}^{n+1})],\\
\boldsymbol{Q}^{n+1} - \frac{\Delta t}{\Delta x}\theta\mathbf{A}^{n+1}\boldsymbol{Q}^{n+1} &= \boldsymbol{Q}^n + \frac{\Delta t}{\Delta x}(1-\theta)\mathbf{A}^n\boldsymbol{Q}^n + \frac{\Delta t}{\Delta x}((1-\theta)\mathbf{b}^n + \theta\mathbf{b}^{n+1}),\\
(I - \frac{\Delta t}{\Delta x}\theta\mathbf{A}^{n+1})\boldsymbol{Q}^{n+1} &= (I + \frac{\Delta t}{\Delta x}(1-\theta)\mathbf{A}^n)\boldsymbol{Q}^n + \frac{\Delta t}{\Delta x}((1-\theta)\mathbf{b}^n + \theta\mathbf{b}^{n+1}),\\
\boldsymbol{Q}^{n+1} &= (I - \frac{\Delta t}{\Delta x}\theta\mathbf{A}^{n+1})^{-1}[(I + \frac{\Delta t}{\Delta x}(1-\theta)\mathbf{A}^n)\boldsymbol{Q}^n + \frac{\Delta t}{\Delta x}((1-\theta)\mathbf{b}^n + \theta\mathbf{b}^{n+1})].
\end{aligned}$$

This expression makes it very easy to calculate the concentration for the next time step, both explicitly and implicitly. For a higher dimensional problem, the computational cost of solving

the system implicitly immediately becomes clear, as it requires a costly $\mathcal{O}(n^3)$ matrix inversion. In our case, we can use the tridiagonal matrix algorithm to do this in $\mathcal{O}(n)$.

## 3.4. Consistency, stability and convergence

In this section we analyse how our numerical method converges to the correct solution as the grid is refined. In particular, we will look at the stability and consistency of our numerical method, and use these to prove it converges. A numerical method can be said to converge if the difference between the solution and the numerical approximation tends to zero as the grid is refined. To analyse the convergence, we introduce the global truncation error at a time $T = n\Delta t$ as

$$E^n = Q^n - q^n;$$

our goal is to keep $E^n$ bounded for arbitrary $n$, by choosing the grid size appropriately. What it means to be convergent, is formalised in the following definition.

**Definition 1** (Convergence)**.** A numerical method $\mathcal{N}$ is called convergent a time $T$ in the norm $||\cdot||$ if

$$\lim_{\Delta t \to 0} ||E^n|| = 0. \tag{3.11}$$

In general, it is difficult to obtain an analytical expression for the global truncation error as the number of time steps increases. Instead, we will make use of a result known as the Lax Equivalence Theorem, which states that a consistent approximation to a well-posed initial-value problem is convergent if and only if it is stable. Before we can define consistency, we need to introduce a concept called the local truncation error, which indicates how much error is introduced at a single time step. If we let $\mathcal{N}$ represent our numerical operator mapping the solution from one time step to the next, then the local truncation error is defined as:

$$\tau^n = \frac{1}{\Delta t}\left[\mathcal{N}(q^n) - q^{n+1}\right]. \tag{3.12}$$

**Definition 2** (Consistency)**.** A numerical method $\mathcal{N}$ is called consistent with the differential equation if the local truncation error vanishes as $\Delta t \to 0$ for all smooth functions $q(x,t)$ satisfying the differential equation [15].

Where consistency aims to get a bound on the error introduced in a single time step, stability aims to extend this bound to the global truncation error, thus proving convergence. To prove that our numerical method is stable, we will use von Neumann stability analysis. The idea behind this is expanding $Q_j^n$ as a Fourier series, and proving that $\hat{Q}^n$ is bounded. Subsequently, it is possible to obtain a bound on $Q_j^n$ using Parseval's relation. Parseval's relation requires using the 2-norm, instead of say, the $\infty$-norm. For completeness, we will give the definition of stability and Lax's Equivalence Theorem.

**Definition 3** (Stability)**.** A numerical method $\mathcal{N}$ is called stable in the norm $||\cdot||$ if for some $C > 0$,

$$||\mathcal{N}_j^n|| \leq C \tag{3.13}$$

for all $n$ and $j$ such that $0 \leq nj \leq T$.

**Theorem 1** (Lax Equivalence Theorem [17])**.** Let $\mathcal{N}$ be a consistent approximation to a well-posed linear initial-value problem. Then $\mathcal{N}$ is convergent if and only if it is stable.

Using the definitions from the previous section, we can now go about proving consistency and stability for our numerical method. To simplify matters, we will assume that our differential

equation contains constant coefficients. In addition, we will do the derivation for the explicit case, and show how this can be extended to the mixed explicit-implicit scheme.

### 3.4.1. Consistency

Applying our numerical method (3.9) to the true solution gives the local truncation error

$$\tau^n = \frac{1}{\Delta t}\left(\frac{1}{\alpha}\left[\alpha q_j + \frac{\Delta t}{\Delta x}\left(\beta\frac{q_{j+1}-2q_j+q_{j-1}}{\Delta x} + \gamma(q_{j-1}-q_j) - \Delta x\rho q_j\right)\right] - q_j^{n+1}\right) \tag{3.14}$$

where we have assumed constant coefficients for simplicity. The polynomial approximation of $q_{j-1}^n$, $q_{j+1}^n$ and $q_j^{n+1}$ about $q_j^n$ gives the following

$$q_{j+1}^n = q_j + \Delta x q_x + \frac{1}{2}\Delta x^2 q_{xx} + \mathcal{O}(\Delta x^3), \tag{3.15}$$

$$q_{j-1}^n = q_j - \Delta x q_x + \frac{1}{2}\Delta x^2 q_{xx} + \mathcal{O}(\Delta x^3), \tag{3.16}$$

$$q_j^{n+1} = q_j + \Delta t q_t + \mathcal{O}(\Delta t^2). \tag{3.17}$$

If we insert this into our expression for the local truncation error, we obtain the following

$$\begin{aligned}
\tau^n &= \frac{1}{\Delta t}\left(\frac{1}{\alpha}\left[\alpha q_j + \frac{\Delta t}{\Delta x}\left(\beta\frac{\Delta x^2 q_{xx} + \mathcal{O}(\Delta x^3)}{\Delta x} - \gamma\Delta x q_x + \mathcal{O}(\Delta x^2) - \Delta x\rho q_j\right)\right] - q_j^{n+1}\right),\\
\tau^n &= \frac{1}{\Delta t}\left(\frac{1}{\alpha}\left[\alpha q_j + \Delta t\left(\underbrace{\beta q_{xx} - \gamma q_x - \rho q}_{\alpha q_t} + \mathcal{O}(\Delta x)\right)\right] - q_j^n - \Delta t q_t + \mathcal{O}(\Delta t^2)\right),\\
\tau^n &= \frac{1}{\Delta t}\left(\Delta t\mathcal{O}(\Delta x) + \mathcal{O}(\Delta t^2)\right),\\
\tau^n &= \mathcal{O}(\Delta x) + \mathcal{O}(\Delta t).
\end{aligned} \tag{3.18}$$

Here the error is dominated by both a $\Delta t$ and $\Delta x$ term, so we see that the method is first-order accurate.[2] This proves that our numerical method is consistent with the differential equation.

For the mixed explicit-implicit case, the term in (3.14) will be split into an explicit $1-\theta$ part, and an implicit $\theta$ part. In this case, the $q_j^{n+1}$ would be split as such as well, but only the part pertaining to the explicit solution would be Taylor expanded around $q_j^n$. Similarly, the implicit term in brackets would be Taylor expanded about $q_j^{n+1}$. This results in first-order accuracy, as before.

### 3.4.2. Stability

In this section we will prove our numerical method is stable using von Neumann stability analysis. We follow the approach stipulated in LeVeque et al. [15], writing $Q_j^n$ as follows

$$Q_j^n = e^{i\xi j\Delta x}. \tag{3.19}$$

Rewriting our finite difference scheme and grouping by like terms, we obtain

$$\begin{aligned}
\alpha Q_j^{n+1} &= \alpha Q_j + \frac{\Delta t}{\Delta x}\left(\beta\frac{Q_{j+1}-2Q_j+Q_{j-1}}{\Delta x} + \gamma(Q_{j-1}-Q_j) - \Delta x\rho Q_j\right),\\
&= \left(\alpha - \frac{2\beta\Delta t}{\Delta x^2} - \frac{\Delta t}{\Delta x}\gamma - \Delta t\rho\right)Q_j + \frac{\Delta t}{\Delta x^2}\beta Q_{j+1} + \left(\frac{\Delta t}{\Delta x}\gamma + \frac{\Delta t}{\Delta x^2}\beta\right)Q_{j-1}.
\end{aligned} \tag{3.20}$$

[2] In fact, expanding up to $\mathcal{O}(\Delta x^4)$ shows that the diffusion term is second-order accurate.

Substituting the coefficients of $Q_j$, $Q_{j+1}$ and $Q_{j-1}$ by $A$, $B$ and $C$, respectively, and applying (3.19) to our finite difference scheme yields

$$\begin{aligned} \alpha Q_j^{n+1} &= AQ_j + BQ_{j+1} + CQ_{j-1}, \\ &= Ae^{i\xi j\Delta x} + Be^{i\xi(j+1)\Delta x} + Ce^{i\xi(j-1)\Delta x}, \\ &= \left(A + Be^{i\xi\Delta x} + Ce^{-i\xi\Delta x}\right) e^{i\xi j\Delta x}. \end{aligned} \tag{3.21}$$

Then, the amplification factor becomes

$$g(\xi, \Delta x, \Delta t) = \frac{1}{\alpha}\left[A + Be^{i\xi\Delta x} + Ce^{-i\xi\Delta x}\right]. \tag{3.22}$$

Here, if $|g(\xi, \Delta x, \Delta t)| \leq 1$ for all $\xi$ then the numerical scheme is stable. Absolute stability is obtained if the amplification factor is strictly less than 1. Because the amplification factor consists of multiple terms, it is not trivial to see when the inequality holds. We can make progress by invoking the triangle inequality, which gives

$$|g(\xi, \Delta x, \Delta t)| = \left|\frac{1}{\alpha}\left[A + Be^{i\xi\Delta x} + Ce^{-i\xi\Delta x}\right]\right| \leq \frac{1}{\alpha}\left[|A| + |B| + |C|\right]. \tag{3.23}$$

By looking at $A$, $B$, and $C$, it seems plausible that the amplification factor can become less than unity depending on the input parameters and grid size. In particular, if we look at $B = \frac{\Delta t}{\Delta x^2}\beta$, we see that $\Delta t$ must be of the same order as $\Delta x^2$. We require this because if they are of the same order, then $B \to \infty$, rendering our numerical scheme unstable.

Invoking the triangle inequality does not guarantee that we can prove that the numerical scheme is stable. In fact, it could be the case that our upper limit (using the triangle inequality) is always larger than one, but that the numerical scheme is still stable for some $\Delta x$ and $\Delta t$. A better approach is plotting the function $g$ and inspecting it visually. It can be difficult to visualise functions $f : \mathbb{C} \to \mathbb{C}$ but there are some tools[3] that accomplish this. We can also parameterise the domain and plot it on an Argand diagram, this is done in Figure 3.2.

For the mixed explicit-implicit case we can substitute $q_j^n = g^n e^{i\xi j\Delta x}$ into our finite difference scheme, and we obtain

$$g(\xi, \Delta x, \Delta t) = \frac{\frac{1}{\alpha}\left(A + Be^{i\xi\Delta x} + Ce^{-i\xi\Delta x}\right)(1-\theta)}{1 - \frac{1}{\alpha}\left(A + Be^{i\xi\Delta x} + Ce^{-i\xi\Delta x}\right)\theta}. \tag{3.24}$$

If $\theta = 0$, then this reduces to our familiar expression (3.22). Note that this expression is not valid for $\theta = 1$, which results in an amplification factor of zero, regardless of the grid size and $\xi$.

We know from Section 3.4.1 that our numerical approximation is consistent with the boundary value problem, and in this section we have seen that it is also stable. This proves, using Lax' equivalence theorem, that our numerical approximation is convergent.

[3] `https://people.ucsc.edu/~wbolden/complex/#(a-(2bt/x%5E2)-(t/x)g-tp)+(t/x%5E2)be%5E(iz)+((t/x)g+(t/x%5E2)b)e%5E(-iz)`

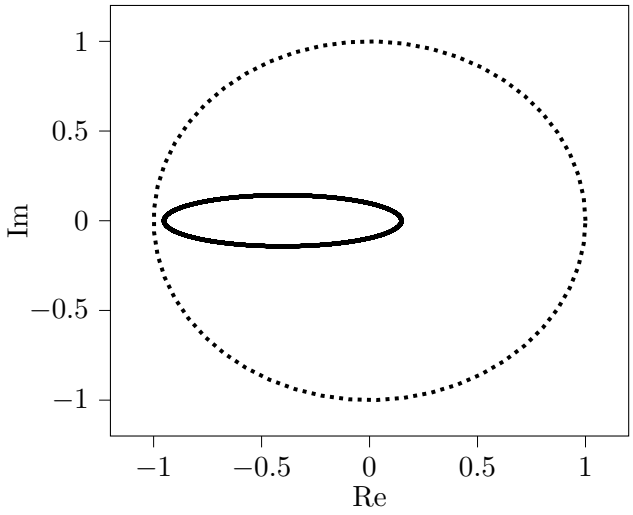



Figure 3.2. Argand diagram showing how $g$ maps the domain $\mathbb{R}$ to $\mathbb{C}$. Clearly, for these particular values of $\Delta x$ and $\Delta t$, the image of $g$ is contained in the unit circle.

## 3.5. Miscellaneous observations

In this section a few comments are made to give a more complete view of how our differential equation is discretised, as well as some considerations made in the process.

### 3.5.1. The CFL condition

The CFL condition, named after Courant, Friedrichs and Lewy, is a necessary condition for a finite volume scheme if we expect it to be convergent. It requires that the numerical domain of dependence contains the true domain of dependence. In other words, for a given point $Q_j^{n+1}$, the numerical method must include every other point that is able to affect it. For the fully implicit method, the CFL condition is always satisfied. The reason for this is that determining the approximation at $t^{n+1}$ requires knowledge of all values at $t^n$, meaning that no matter the grid spacing or time step, we are always using the entire physical domain of dependence [18]. For the explicit upwind method without diffusion, the CFL condition is

$$\left|\frac{\gamma \Delta t}{\Delta x}\right| \leq 1. \tag{3.25}$$

If we were solving the equation explicitly, our grid size would have to conform to this criterion.

### 3.5.2. Upwind vs central difference

Usually the advection term is not discretised using a central difference scheme, as doing so can result in solutions that are not physically realistic [16]. The benefit of using a central difference scheme for the advection term, is that it is second-order accurate. This allows for a larger grid size while keeping the numerical scheme consistent, reducing computational time. In general, the order of accuracy is only as good as the worst order of accuracy in the approximation. Since we have both a diffusion and advection term, the order of accuracy of our method will be equivalent to the order of accuracy of the worst of the two. In our case, the advection term is first-order accurate, but since we are only trying to solve the problem for a few breathing cycles, pursuing higher order accuracy is unnecessary, because computational time is not an issue.

### 3.5.3. Nonuniform grid spacing

Instead of using a grid with uniform grid spacing, we can employ a so called nonuniform grid. This means that the distances $(\delta x)_{j-1/2}$ and $(\delta x)_{j+1/2}$ are not necessarily equal. Usually,

nonuniform grid spacing is used to deploy computing power more efficiently. In places where $q$ varies steeply with $x$ the grid should be finer than where $q$ changes rather slowly with $x$.

These concerns are a valid reason to use a nonuniform grid, but in our case there is another reason. This stems from the fact that the first generation is a lot longer than say, the 21st. If we use a nonuniform grid, it is possible distribute the grid points over the generations equally. This is especially useful when using fewer grid points, because in that case it is possible that final generations have only a single grid point, and if this grid point is in the first 80% of the airway, then the impaction velocity over the whole generation is zero (see Section 2.2.1).

### 3.5.4. Numerical calculation of the velocity field

In Section 2.3 we derived an expression for the velocity profile. Here, we will briefly touch on how to numerically evaluate this profile. By looking at equation (2.15), we see that we need to be able to numerically evaluate the derivative $\frac{\partial}{\partial t}A_T$, and the spatial integral over this derivative. The first thing to notice is that $\frac{\partial}{\partial t}A_T = 0$ for generations $< 16$, simplifying the equation to a simple mass balance. If we evaluate this expression for the later generations, then we can start to iterate over $x$ and $t$ to derive the entire velocity profile.

Fortunately, it simply suffices to use a central difference rule for the derivative, and any numerical integration method such as the trapezoidal rule. There exists a central difference function `numpy.gradient` that calculates the central difference for us, and uses forward and backward differences for the two endpoints. Similarly, the trapezoidal rule is implemented in `scipy.integrate.trapz`. We can then choose $a = 0$ or any other $a <$ generation 16 and start the iteration process.

# 4

# Model verification

As in any numerical study, it is important to verify that the numerical results are physically accurate. In addition to standard sanity checks such as interchanging the boundaries or using physical common sense, another way to do this is by comparing numerical results to experimental results. In our case, this is not possible, but it is possible to compare the results to those of Mitsakou, Helmis, and Housiadas [1]. Additionally, it is also possible to apply the model to a problem that has a known analytical solution. This increases (by induction) the confidence in our solution for the extended problems (but it is by no means certain that the solutions for the extended problems are correct!).

## 4.1. Comparison with analytical solution

We begin our verification with the last approach: we test our model on the 1-dimensional advection-diffusion equation, with a first-type boundary condition on the left side, $q(0,t) = 2$, and a second-type boundary condition on the right $q_x(3,t) = 0$. The governing equation reads

$$\frac{\partial q}{\partial t} = -u\frac{\partial q}{\partial x} + D\frac{\partial^2 q}{\partial x^2}, \tag{4.1}$$

where the initial condition is $q(x,0) = 0$.

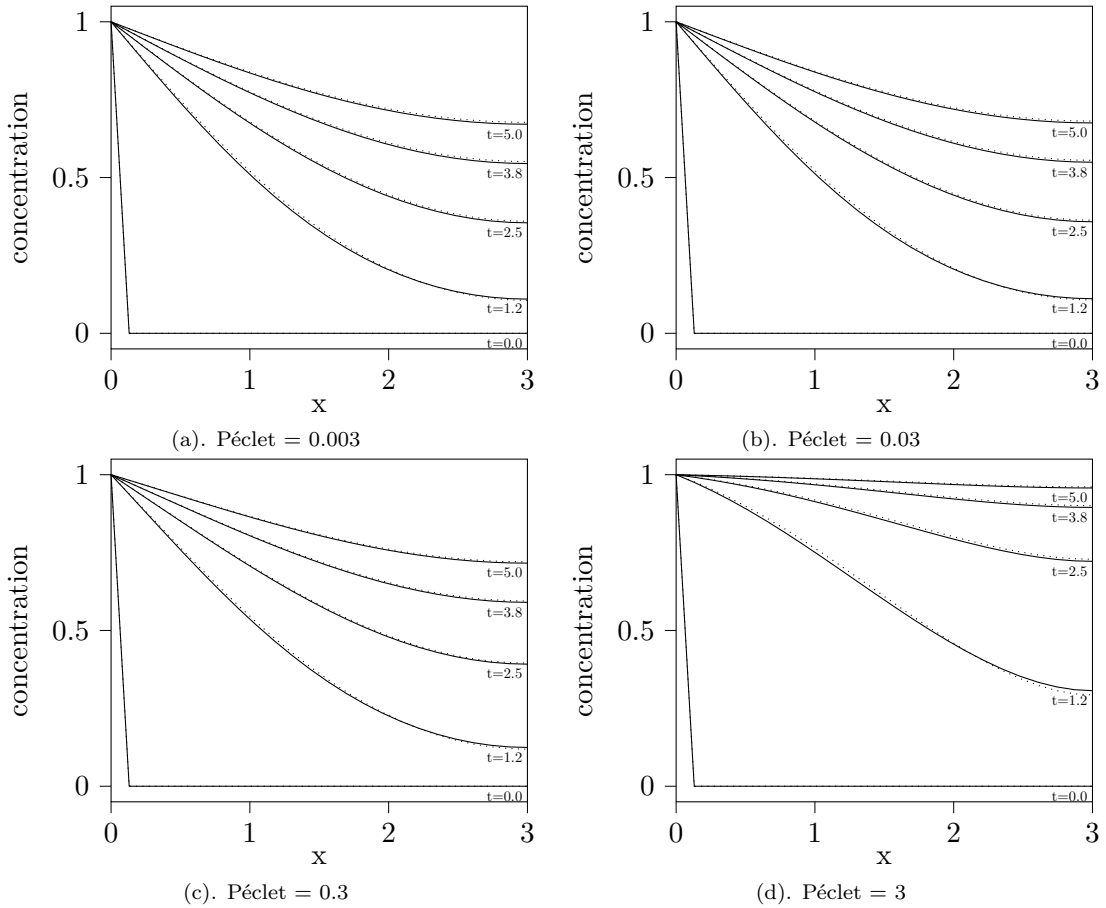
The analytical solution to this equation is known, and has been compiled (along with solutions to similar problems) by Genuchten and Alves [19]. The comparison between their analytical solution and the solution of our model is shown in Figure 4.1. It is easy to see that for the range of different Péclet numbers[1], the numerical model holds up well.

In addition to the non-stationary problem, the solution for the stationary problem was also compared with the numerical model. The exact solution to the stationary problem is known, it is

$$q(x) = q(0) + [q(L) - q(0)]\frac{\exp(\mathrm{Pe}\, x/L) - 1}{\exp(\mathrm{Pe}) - 1}, \tag{4.2}$$

where Pe is the Péclet number and the boundary conditions are both of the Dirichlet type. A comparison of this solution with that of the numerical model at large times is shown in Appendix B.

[1] The Péclet number is defined as the ratio of the rate of advection to the rate of diffusion of a physical quantity, or $\mathrm{Pe} = Lu/D$.




(a). Péclet = 0.003 (b). Péclet = 0.03

(c). Péclet = 0.3 (d). Péclet = 3

Figure 4.1. The solutions to a 1-dimensional advection-diffusion equation for different Péclet numbers. The dotted lines represent the numerical solutions, and the solid lines represent the analytical solutions.

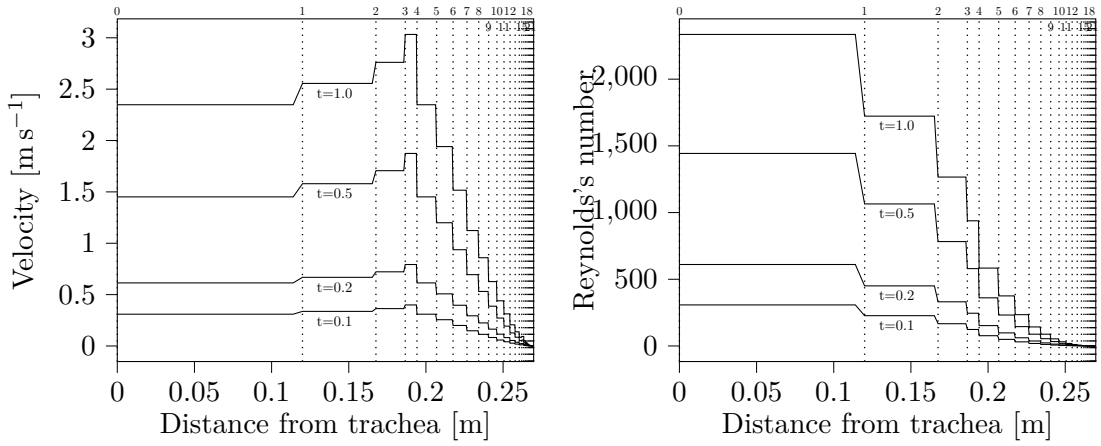
## 4.2. Comparison with literature

In this section we concern ourselves with comparing our results to those of Mitsakou, Helmis, and Housiadas [1]. We will primarily compare the deposition fractions for a range of different particle sizes, as they do not specify the aerosol concentration for a range of different time steps. Before the literature comparison, we will look at the velocity profile and concentration profile ourselves, to see if they are physically viable.

### 4.2.1. Velocity profile

As explained in Section 2.3, we first have to calculate the velocity field before we can solve the GDE (2.1). The velocity is of course closely related to the breathing function, and the tidal volume. In our case we have a symmetric breathing function, and we use a tidal volume of 1000 ml.

From the inlet velocity, we can calculate the total velocity profile over the entire RT. This profile will be a step function in the non-alveolated generations, because the velocity over a generation is assumed to be constant (due to the continuity equation and the fact that the airway diameter is constant over a generation). The velocity profile over a range of different time steps is shown in Figure 4.2a. From the velocity profile and the (changing) airway diameter, it is possible to calculate Reynolds number as well. This is done in Figure 4.2b. Likewise, this is also a stepwise function due to the stepwise properties of generation values.




(a). Velocity profile in the RT.

(b). Reynolds number in the RT.

Figure 4.2. The velocity profile and Reynolds number plotted during the first second of a symmetric breathing cycle of period 4 s. At $t = 1$ s the velocity reaches its peak. The dotted lines in these and subsequent figures signify the start of a new generation. Naturally, the tidal volume and breathing function have a large impact on the velocity profile.

It is clear that the velocity is highest in the third generation, and approaches zero for the later generations. This is because the cross sectional surface area decreases up until the third generation, and after this increases monotonically. Throughout the cycle, Reynolds number varies between 1 and around 2500, which just about reaches the critical value that marks the change into the turbulent regime. In the RT, Reynolds number decreases monotonically due to the decreasing velocity, and decreasing characteristic linear dimension (diameter). In the later generations the velocity is almost negligible, because most of the air moves into the alveolar sacs instead of continuing through the conducting part of the RT. This also conforms to our expectation, and for these reasons the velocity profile looks to be physically acceptable. What remains to be seen is whether the concentration profile is acceptable as well.

### 4.2.2. Concentration profile

By solving the aerosol general dynamic equation (2.1), we are able to determine the concentration $q(x, t)$ for the entirety of the respiratory tract and also at every time step $t$. We thus obtain a complete description of the aerosol dynamics. The concentration profile depends heavily on the particle size. If the particles are small, then they are able to reach the transitional bronchioles (generation $\geq 17$), and even the alveolar ducts (generation $\geq 20$). The larger particles however, do not penetrate as deeply, and because of this they also leave the RT quicker than the smaller particles, which can stay in the RT even at the end of the expiration phase (depending on the tidal volume).

Figure 4.3 shows the concentration profile at different time steps for particles of size $d = 0.01\,\mu$m. The different time steps correspond to the first second of a symmetric breathing cycle of period 4 s, in 0.1 s intervals. It is remarkable that the concentration profile very quickly approaches zero when the expiration phase starts. And because the concentration is zero at the end of expiration, a steady breathing pattern is established instantaneously. This seems counter-intuitive, especially if you pay attention to your own breathing. If you breathe in deeply, it can take quite a while before all the air has exited your lungs. However, the aerosols do not penetrate the lungs so deeply (compared to the much smaller $N_2$, $O_2$, and $CO_2$ molecules), which is why the aerosols exit the lungs much quicker. For a study specifically describing the effect of these smaller molecules, consult [20].

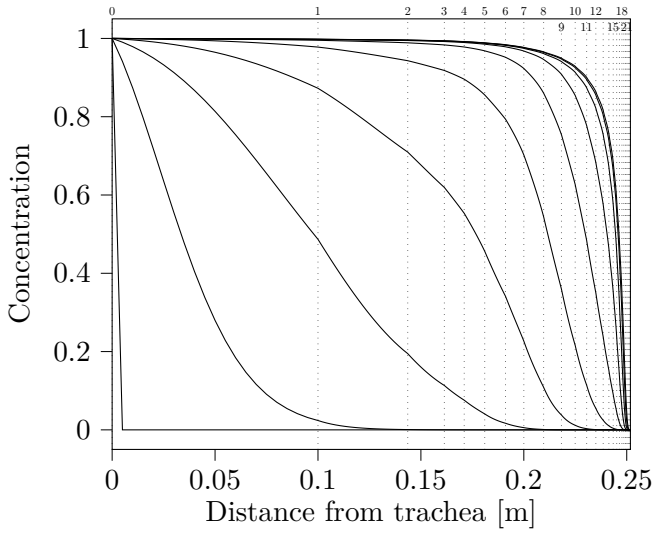



Figure 4.3. The concentration profile of an aerosol of diameter 0.01 µm plotted during the first second of a symmetric breathing cycle of period 4 s. Starting at 0, the concentration increases with each time step (in 0.1 s intervals), until it stabilises. Note that the total time span is the first half of the inspiration phase.

In principle, our model accepts particle sizes in the range of 0.1 nm, but applying our model on the constituent gasses of air (with those sizes) results in nonphysical results. This is because the concentration of these gasses is much higher, at which point our original partial differential equation does not describe our particles any longer. Indeed, the deposition will be much lower than described by the general aerosol equation.

### 4.2.3. Deposition fraction

The goal of our programme is to calculate the deposition fraction for different classes of aerosol particles. The difference in particles that leads to the biggest change in the deposition fraction is the particle diameter. Indeed, smaller particles will experience strong diffusion, whereas the dominant deposition mechanism for bigger particles is gravitational settling or impaction.

In Section 4.1 we tested our model on a known analytical solution. This is not enough however, because this gives us confidence only for a particular situation. Our models aims to determine the deposition ratio of aerosols of different sizes, and we can compare our results to [1]. The literature study compares the deposition ratio for particles of different sizes, ranging from 0.01 µm to 10 µm in diameter. The lung conditions are a tidal volume of 1000 ml and a period of respiration of 4 s. Their results, along with ours are shown in Figure 4.4.

The results match very closely, even though we do not take into account particle-particle and gas-to-particle interactions (coagulation and growth). The offset cannot entirely be explained by the neglection of these two terms. Including these two interactions increases or decreases the particle diameter during the simulation process either by growth or coagulation. Because of the high humidity in the lungs, particles are expected to grow significantly, which increases their effective particle diameter. This would shift our results to the left, but this is not entirely explained by Figure 4.4.

More likely, the offset is explained by a difference in starting conditions. For example, the particle density influences the curve as well. In this case, the particle density is $1\,\mathrm{g\,cm^{-3}}$, but if we increase this two or threefold, the deposition fraction increases for all particle sizes.

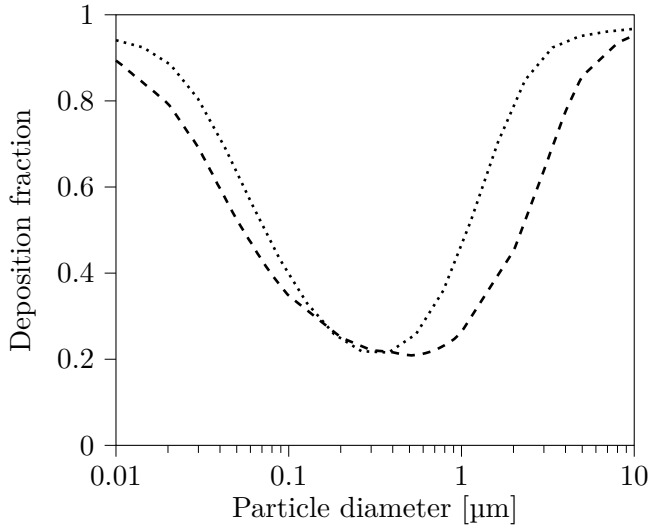


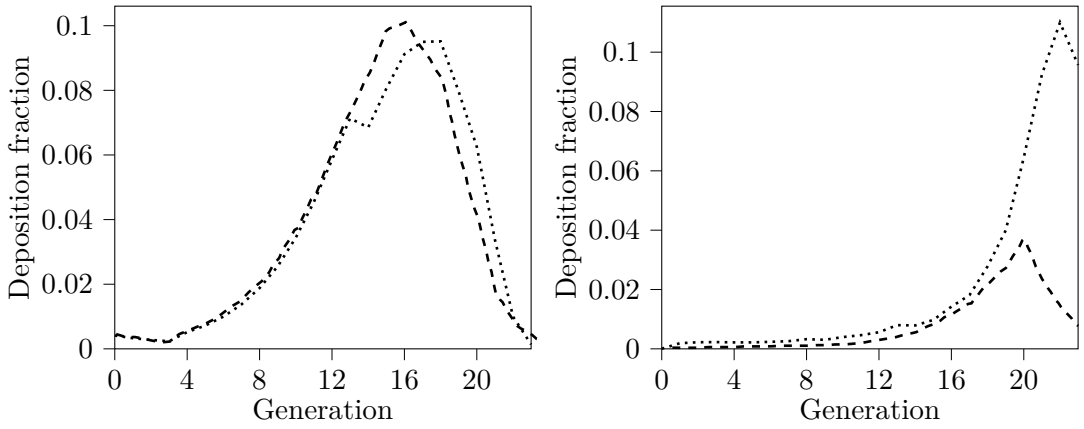

Figure 4.4. Comparison of the calculated deposition fraction with literature [1] as a function of particle diameter. The dotted line shows our results.

The larger particles (>1 µm) are especially affected, however, as their relaxation time is much higher (which influences impaction), and the particle relaxation time depends non-linearly on the particle diameter. It is a bit unclear what the exact starting conditions of the literature study [1] are. They produce a number of graphs with seemingly different deposition fraction curves, some of which look more similar to our results in Figure 4.4.

Instead of calculating the total deposition fraction, it is also possible to calculate local deposition fractions, for example the deposition fraction per generation. This is done in Figure 4.5, where we also compare our results to those compiled by [1]. Here, we have calculated the local deposition fraction for particles with diameters of 0.01 µm and 1 µm.

The results for the smaller particles match closely with literature, as can be seen in Figure 4.5a. The model seems to overestimate the deposition fraction for the larger class of particles from generation 20 onward. This could be due to an overestimation of the diffusion velocity, which especially affects generations with smaller diameters. Here, the flexibility of our programme starts to be appreciated, because we can turn off particular deposition mechanisms to see which is the cause of the irregularity.

In this case, gravitational sedimentation seems to be culpable for the difference in Figure 4.5b. This does not necessarily cast doubt on the validity of our model; the literature study is slightly vague in how they obtained the gravity angles for their model. Although they specified that they obtained the angles from Weibel's model "A", this model neither specifies individual branching angles, nor the angles of inclination with respect to the horizontal (the gravity angles) [21]. If the gravity angles are smaller in the later generations, the results would look more similar. Nevertheless, the preceding comparisons give us the confidence that our model performs well under a range of different conditions, and we will use this to perform a parametric study in Chapter 5.




(a). $d = 0.01\,\mu m$ (b). $d = 1\,\mu m$

Figure 4.5. Comparison of the calculated local deposition fraction with literature [1] for particles of size 0.01 µm and 1 µm The dotted line shows our results.

# 5

# Parametric studies

Knowing how the breathing rate affects aerosol deposition is important, as the recent example of bush fires in Australia (2019-20) demonstrates. This bushfire season, also known as the Black Summer, scorched over 18 million hectares of land, and killed more than 400 people. Smoke inhalation was the cause of 417 of these deaths [22]. Bushfire smoke covered large areas of land, and sports events were cancelled because of concerns of smoke inhalation, while normal exercise was discouraged. In this parametric study, we will examine how the breathing pattern affects deposition rates and total deposition of aerosols. Exercise results in an elevated heart rate and quickened breathing, so the expectation is that the total deposition will increase - we wish to quantify this.

In addition, the world is currently plagued by the COVID-19 pandemic, and preliminary case studies have indicated that airborne transmission plays a profound role in spreading the virus [23]. For this reason, we will pay special attention to particles of around 1 µm in diameter, which is the predominant size of coughed droplets [24].

## 5.1. Sensitivity runs

Before investigating the effect of different breathing conditions, we will do a series of sensitivity runs to check how the model responds to other parametric variations. These variations differ from changing the breathing conditions in that they pertain to the computational model in question, and not the environmental conditions of the patient.

All of the simulations have been run with a time step of $\Delta t = 0.1\,\mathrm{s}$, and a grid size of 480 nodes. These nodes have been distributed equally over the different generations, with 20 nodes in each generation. With these parameters, a single simulation takes a few seconds to complete on a personal computer. Examining the effect of particle size (to produce plots like Figure 5.2) generally takes a few minutes for a particular set of initial parameters, making it very easy to quickly analyse a number of parametric variations.

### 5.1.1. Geometry rescaling

The first of these is checking how the scaling the generation lengths and radii affects the deposition fraction. It is customary to scale Weibel's lung model by a factor of $(V_{\mathrm{FRC}}/4.8\,\mathrm{l})^{1/3}$, as is explained in Appendix A. The deposition fractions with and without this scaling factor are shown in Figure 5.1a.

For lungs with a $V_{\mathrm{FRC}} = 3300\,\mathrm{ml}$, the rescaling of the geometry makes almost no difference.

This effect becomes more pronounced when the functional residual capacity becomes smaller, which makes the scaling factor larger. A possible explanation for the slightly higher deposition fractions when the geometry is rescaled is that the bronchi and bronchioles become smaller, resulting in more deposition due to diffusion and impaction. This is also seen in Figure 5.1b, which explores the effect of the rescaling on the local deposition fraction. Here, the deposition increases in the first few generations, but seems to equalise beyond generation 16, where the effect of the rescaling is offset by the varying diameter $d_T$ due to breathing mechanics.

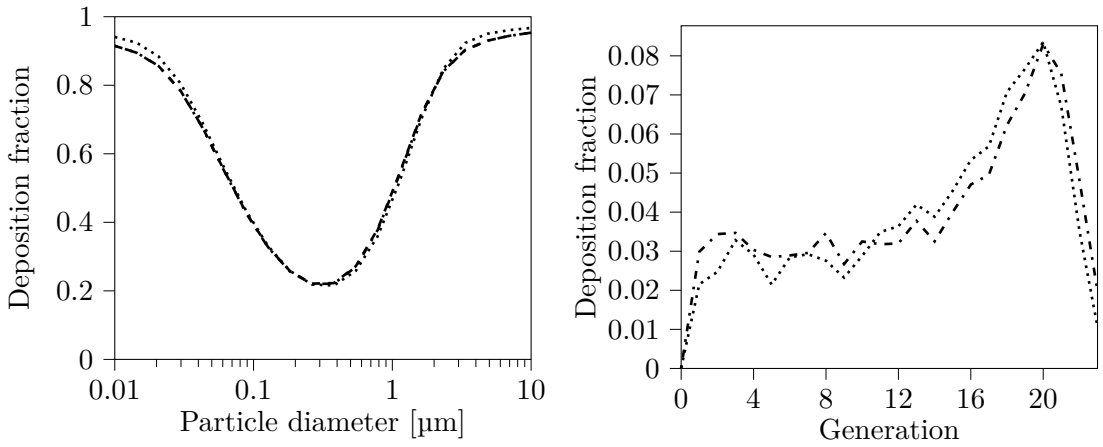



Figure 5.1. (a) Deposition fractions with (dotted) and without (dot-dashed) a rescaling of Weibel's model, for a range of different particle sizes. The dashed line also includes a rescale but with $V_{\mathrm{FRC}} = 2400\,\mathrm{ml}$, and (b) the local deposition fraction for the same parameters but with $d = 4\,\mu\mathrm{m}$.

### 5.1.2. Weibel's lung geometry or Yeh and Schum's

Weibel's model "A" is only one of many lung models devised over the years. Some vary in the volume of the alveolar sacs or the number of generations, others include the asymmetry of the left lung. We will look at a proposed geometry by Yeh and Schum, which varies in the generation length and radius, making a comparison simple. The different lung geometries are compared in Figure 5.2.

Note that in these calculations we have discounted the final generation in Yeh and Schum's model, which is why the difference between the two models is very pronounced. Across the entire spectrum, Weibel's model reports more deposition than Yeh and Schum's. The final generation of Yeh and Schum's model consists of $3 \times 10^8$ alveolar sacs each with a length of 250 µm and a diameter of 300 µm [3]. Future work could look into implementing Yeh and Schum's typical path lung model in its entirety, which would include quasi 3-dimensional geometries and allow for more accurate comparison. The quasi 3d structure is due to their model's asymmetry - they specify six different segments, considering the difference between the right and left lung, and their top, middle and lower lobes. Each of these six segments has their own number of alveolar sacs. Implementing these corrections, deposition fractions would be more akin to those seen in Weibel's model, as can be seen in [4].

### 5.1.3. Time-dependent vs fixed geometry

Lastly, we will look at the effect that the time-dependent geometry has on particle deposition. For generations >16, the generation radii are allowed to increase or decrease to account for the varying lung capacity, as explained in Section 2.3. The effect of the time-varying geometry is shown in Figure 5.3a

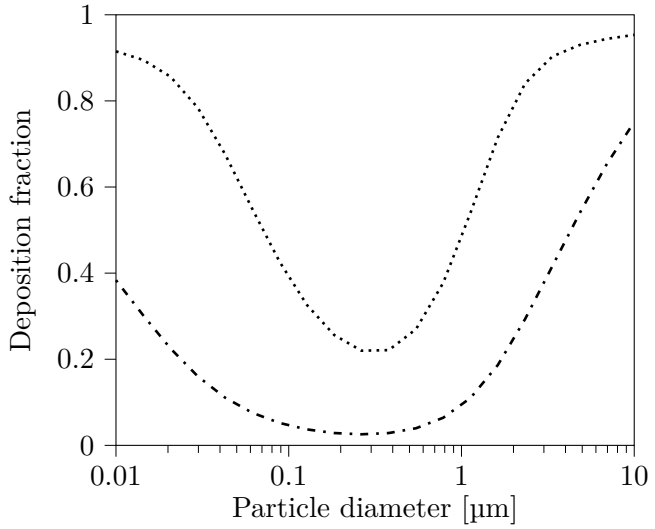



Figure 5.2. Deposition fractions for the parameters of Weibel's model "A" (dotted) and Yeh and Schum's (dashed). For the latter model, the final generation has not been taken into consideration.

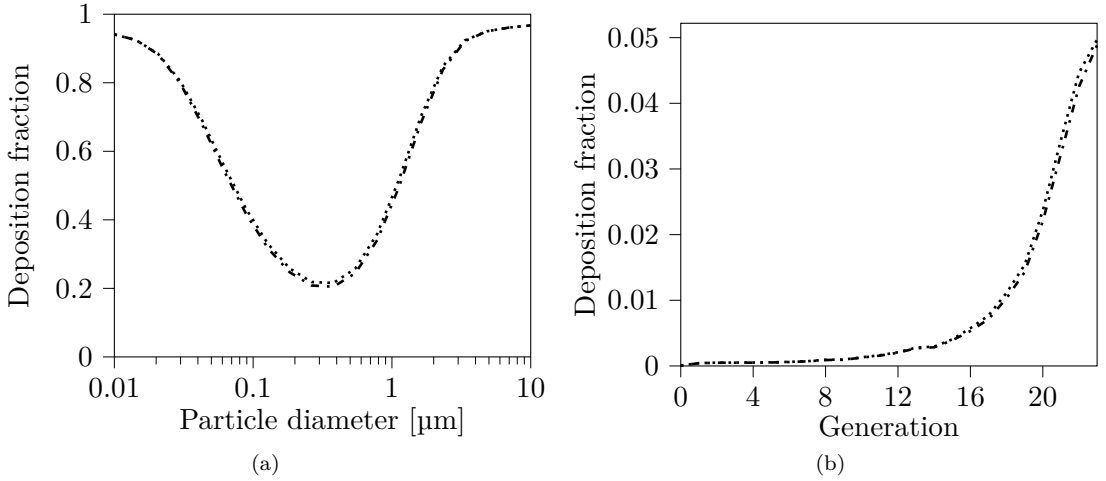



Figure 5.3. (a) Deposition fractions for time-dependent (dotted) and fixed lung geometry (dashed), and (b) the local deposition fraction for the same parameters but with $d = 0.3\,\mu\text{m}$.

The time-dependent geometry makes almost no difference on the deposition fraction across the range of particle sizes. The slightly higher deposition fractions when the geometry is time-dependent can be explained by the fact that the wetted perimeter increases in the last few generations, increasing the deposition surface. But the difference is very small, and is only seen for $d = 0.3\,\mu\text{m}$, where the difference in deposition is largest.

## 5.2. Deposition fractions under eupnoea and hyperpnea

In this section we will investigate how the deposition fraction varies of a range of particle sizes under different respiratory conditions. The main parameter that we will change is the tidal volume $V_T$. During exercise, the functional residual capacity decreases [25] and the tidal volume increases. Figure 5.4a shows the deposition fraction for a number of respiratory conditions.

For the most part, the deposition fraction decreases as the tidal volume increases. At the tails, the process seems to reverse. The results match closely with the results of Mitsakou, Helmis,

and Housiadas [1]. Unfortunately, we cannot compare the tail behaviour because the literature only tested the model under regular breathing circumstances.

Instead of looking at the deposition fraction, we can also look at how the total deposition develops over time, for different tidal volumes. This will enable us to explain whether the banning of exercise in Australia during the bushfires was academically founded. The cumulative absorption of aerosols is shown in Figure 5.4b. Indeed, we see that despite the smaller deposition fraction, the cumulative absorption of the larger tidal volume increases at almost twice the speed compared to the smaller one. This of course, is entirely as expected, because the air intake rate is a lot higher.

Note that the "normalised" absorption in Figure 5.4b is the total absorption assuming a normalised concentration at the inlet. In this sense, the absorption is not "normalised" in the standard sense of the word. We also notice the step-wise increase of absorption - absorption mostly occurs during the inspiration phase.

In addition to tidal volume variation, we also investigated the effect of breathing rate on the deposition fraction. Breathing rates of 10, 12 and 15 breaths per minute result in an almost uniform decrease of the deposition fraction across the spectrum. However, because of the similarity between this figure and Figure 5.4a, we have not included it here.

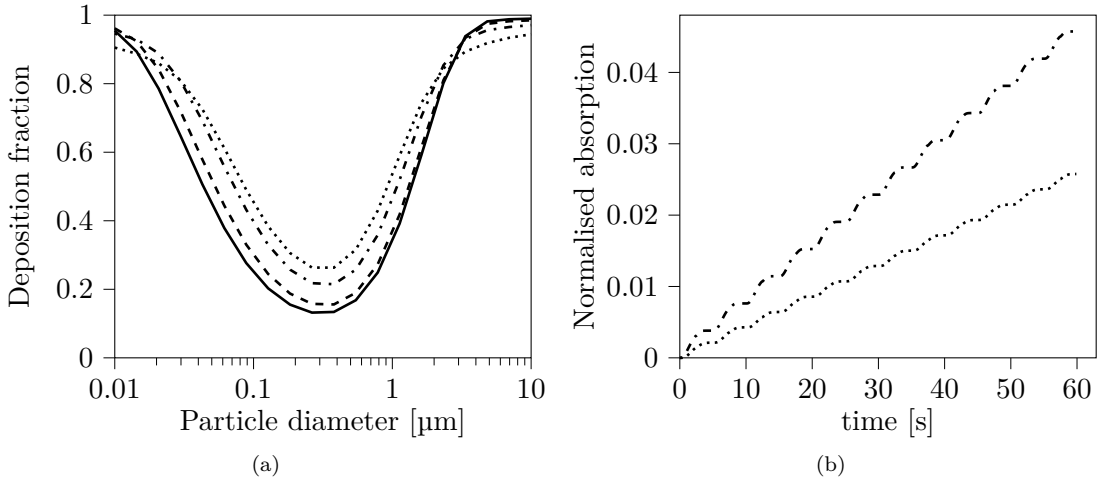



Figure 5.4. (a) Deposition fractions for tidal volumes of 500 ml, 1000 ml, 2000 ml and 3000 ml (dotted to solid, respectively) for a range of particle sizes, and (b) the cumulative absorption during the first minute for particles of size 0.3 µm with tidal volumes of 1000 ml and 3000 ml (dotted and dashed, respectively).

## 5.3. The effect of aerosol density

In Section 4.2.3 we briefly discussed the effect of particle density on the deposition fraction of different classes of particles. This was in relation to the literature comparison, because it was unclear what parameters they ran their programme with, making the comparison more difficult. Figure 5.5 show the effect of particle density on the deposition fraction.

Particle density does not seem to have a great effect for the smaller particles, whose deposition mechanism is mostly due to diffusion. The larger classes of particles, whose main deposition mechanism is impaction, experience a greater effect. The deposition fraction increases for particles bigger than about 0.1 µm. Overall, the trough for medium sized particles decreases and shifts to the left.

Cough droplets, which have a density of $1.0\,\mathrm{g\,cm^{-3}}$ and a typical particle diameter of 1 µm [24],

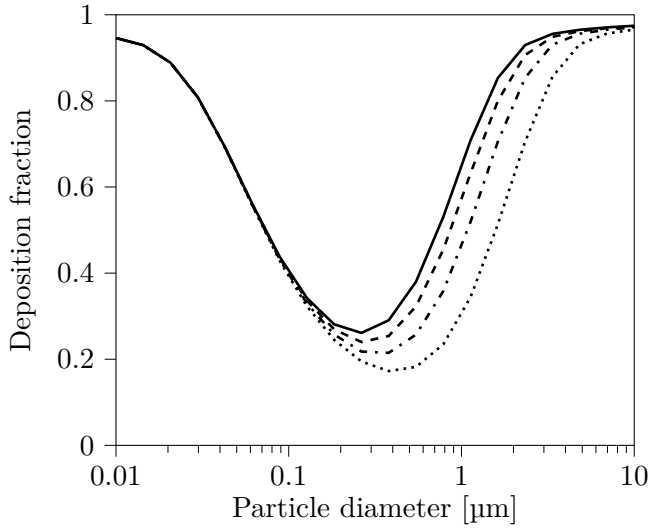



Figure 5.5. Deposition fractions for particle densities of $0.5\,\mathrm{g\,cm^{-3}}$, $1.0\,\mathrm{g\,cm^{-3}}$, $1.5\,\mathrm{g\,cm^{-3}}$ and $2.0\,\mathrm{g\,cm^{-3}}$ (dotted to solid, respectively) for a range of particle sizes.

seem to deposit around 50% of the time. This indicates that COVID-19 borne droplets have a high chance of being absorbed in the lungs, but modelling the actual transmission risk is out of the scope of this thesis.

# 6

# Conclusions and recommendations

In this study, we developed a 1-dimensional model describing the behaviour of aerosols in the human respiratory tract. It proved capable of accurately calculating the local particle deposition, while taking into account both particle properties and respiratory conditions of the patient. The model was based on a model developed by Mitsakou, Helmis, and Housiadas [1], who investigated particle deposition fractions for a range of different particle sizes. We validated our results by comparing them with literature, and we obtained satisfactory agreement across the entire spectrum of particle sizes.

Our study had a similar focus as the aforementioned one, and we extended their work by considering the deposition fraction under a range of different respiratory conditions. This was facilitated by the little computational effort required to generate results, requiring just seconds to calculate the deposition fraction during a breathing cycle. In contrast, present 3-dimensional models require millions of control volumes to model only a part of the RT, which demonstrates the usefulness of our simplistic 1-dimensional approach.

After verifying that the numerical scheme was convergent, we did extensive verification on both the concentration profile as well as the (local) deposition fraction. This was used to perform a sensitivity analysis of the computational parameters on the model, which proved to respond in a consistent way. Varying the physiological parameters of the patient was done next, where we varied the tidal volume and the breathing rate and assessed their influence on the deposition fraction. Both an increase in tidal volume as well as an increase in the breathing rate proved to decrease the deposition fraction across the spectrum of particle sizes, apart from at the tails of the distribution. Despite this, an increase in tidal volume resulted in increased normalised absorption, due to the extra air intake.

Apart from varying physiological parameters, we also looked at particle-specific parameters like the density. It was reported that an increase in particle density increased the deposition fraction for the larger class of particles ($> 0.1\,\mu m$).

There are a number of improvements to this study that can be realized in a future work. We briefly experimented with different morphological models such as that of Yeh and Schum, but we did not implement it in its entirety. By using this morphological description, it will be possible to report lobar deposition instead of merely the average deposition for the whole lungs. In addition, we did not model hydroscopic growth, which has a big effect on particle deposition due to the high relative humidity ($\approx 99.5\%$) prevailing in the human lungs [1]. If this growth is also implemented, the model could be used to accurately model regional deposition while taking

into account all of the important physical interactions that aerosols undergo in the lungs.

Of lesser importance, a higher order numerical scheme can be developed in an attempt to further reduce the computational time required to run simulations. Moreover, we made various assumptions that can be challenged, such as that the terminal settling velocity is reached instantly, and that impaction is the same during inspiration and expiration. Addressing these issues can improve the model's predictions, but care must be taken as not to lose sight of the bigger picture. A numerical model can be used as a benchmark, but empirical verification should always be employed in a real world scenario.

# References


[1] C. Mitsakou, C. Helmis, and C. Housiadas. "Eulerian modelling of lung deposition with sectional representation of aerosol dynamics". In: Journal of Aerosol Science 36.1 (2005), pp. 75–94. ISSN: 0021-8502. DOI: https://doi.org/10.1016/j.jaerosci.2004.08.008. URL: http://www.sciencedirect.com/science/article/pii/S0021850204003283.

[2] Ewald R Weibel, André Frédérick Cournand, and Dickinson W Richards. Morphometry of the human lung. Vol. 1. Springer, 1963.

[3] "Chapter 2 Morphometry of the human respiratory system". In: Inhaled Particles. Ed. by Chiu-sen Wang. Vol. 5. Interface Science and Technology. Elsevier, 2005, pp. 7–30. DOI: https://doi.org/10.1016/S1573-4285(05)80006-7. URL: http://www.sciencedirect.com/science/article/pii/S1573428505800067.

[4] Werner Hofmann. "Modelling inhaled particle deposition in the human lung—A review". In: Journal of Aerosol Science 42.10 (2011), pp. 693–724. ISSN: 0021-8502. DOI: https://doi.org/10.1016/j.jaerosci.2011.05.007. URL: http://www.sciencedirect.com/science/article/pii/S0021850211000875.

[5] B. J. Mason. The physics of clouds, by B. J. Mason. English. 2nd ed. Clarendon Press Oxford, 1971. ISBN: 0198516037.

[6] Keith James Laidler. Student solutions manual for physical chemistry. eng. Menlo Park, Calif., [etc.]: Benjamin/Cummings Pub. Co, 1982. ISBN: 0805356835.

[7] Emma Cunningham and Joseph Larmor. "On the velocity of steady fall of spherical particles through fluid medium". In: Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character 83.563 (1910), pp. 357–365. DOI: 10.1098/rspa.1910.0024. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1910.0024. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1910.0024.

[8] L.P.B.M. Janssen and M.M.C.G. Warmoeskerken. Transport Phenomena Data Companion. 3rd ed. Delft: VSSD, 2006. ISBN: 9789071301599.

[9] Albert Einstein et al. "On the motion of small particles suspended in liquids at rest required by the molecular-kinetic theory of heat". In: Annalen der physik 17 (1905), pp. 549–560.

[10] Ramesh K Shah and Alexander Louis London. Laminar Flow Forced Convection in Ducts. A source book for compact heat exchanger analytical data. Academic Press, 1978. ISBN: 0-12-020051-1.

[11] Sheldon K. Friedlander. Smoke, Dust, and Haze. Fundamentals of Aerosol Dynamics. 2nd ed. Oxford University Press, 2000. ISBN: 978-0-195129-99-1.

[12] Kim E. Barrett et al. Ganong's Review of Medical Physiology. 23rd ed. McGraw-Hill Medical, July 2009, p. 593. ISBN: 978-0-07-160568-7.

[13] B. Sarangi et al. "Aerosol effective density measurement using scanning mobility particle sizer and quartz crystal microbalance with the estimation of involved uncertainty". In: Atmospheric Measurement Techniques 9.3 (2016), pp. 859–875. DOI: 10.5194/amt-9-859-2016. URL: https://www.atmos-meas-tech.net/9/859/2016/.

[14] S.G Jennings. “The mean free path in air”. In: Journal of Aerosol Science 19.2 (1988), pp. 159–166. ISSN: 0021-8502. DOI: https://doi.org/10.1016/0021-8502(88)90219-4. URL: http://www.sciencedirect.com/science/article/pii/0021850288902194.

[15] Randall J LeVeque et al. Finite volume methods for hyperbolic problems. Vol. 31. Cambridge university press, 2002.

[16] Suhas Patankar. Numerical Heat Transfer and Fluid Flow. CRC Press, 1980.

[17] Lloyd N. Trefethen. “Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations”. unpublished text. 1996. URL: https://people.maths.ox.ac.uk/trefethen/pdetext.html.

[18] David Wells (https://scicomp.stackexchange.com/users/1891/david-wells). Understanding the Courant–Friedrichs–Lewy condition. Computational Science Stack Exchange. Version 2016-11-07. URL: https://scicomp.stackexchange.com/q/25439.

[19] Martinus van Genuchten and W.J. Alves. Analytical Solutions of One Dimensional Convective Dispersive Solute Transport Equations. Vol. 1661. June 1982.

[20] F.H.C. De Jongh. “Ventilation modelling of the human lung”. doctoral thesis. Delft University of Technology, 1995. URL: http://resolver.tudelft.nl/uuid:ed787c31-f9de-45c6-b2da-974578d0f581.

[21] Peter R. Byron. Respiratory Drug Delivery. 1st ed. Boca Raton: CRC Press, 1989. DOI: https://doi.org/10.4324/9780203710456.

[22] Nicolas Borchers Arriagada et al. “Unprecedented smoke-related health burden associated with the 2019–20 bushfires in eastern Australia”. In: Medical Journal of Australia (Feb. 2020). DOI: 10.5694/mja2.50545. URL: https://onlinelibrary.wiley.com/doi/abs/10.5694/mja2.50545.

[23] Mahesh Jayaweera et al. “Transmission of COVID-19 virus by droplets and aerosols: A critical review on the unresolved dichotomy”. In: Environmental Research 188 (2020), p. 109819. ISSN: 0013-9351. DOI: https://doi.org/10.1016/j.envres.2020.109819. URL: http://www.sciencedirect.com/science/article/pii/S0013935120307143.

[24] Shinhao Yang et al. “The Size and Concentration of Droplets Generated by Coughing in Human Subjects”. In: Journal of aerosol medicine : the official journal of the International Society for Aerosols in Medicine 20 (Feb. 2007), pp. 484–94. DOI: 10.1089/jam.2007.0610.

[25] M.T. Sharratt et al. “Exercise-induced changes in functional residual capacity”. In: Respiration Physiology 70.3 (1987), pp. 313–326. ISSN: 0034-5687. DOI: https://doi.org/10.1016/0034-5687(87)90013-2. URL: http://www.sciencedirect.com/science/article/pii/0034568787900132.

[26] Warren H. Finlay. “5 - Introduction to the respiratory tract”. In: The Mechanics of Inhaled Pharmaceutical Aerosols. Ed. by Warren H. Finlay. London: Academic Press, 2001, pp. 93–103. ISBN: 978-0-12-256971-5. DOI: https://doi.org/10.1016/B978-012256971-5/50006-7. URL: http://www.sciencedirect.com/science/article/pii/B9780122569715500067.

# A

# Morphometry data

Table A.1 shows the characteristics of the respiratory tract for an average adult with a lung volume of 5600 ml. Although Yeh and Schum devised the trachea as generation 1, we have started the numbering at 0 to facilitate comparison with Weibel's model.

Table A.1. Characteristics of Yeh and Schum's model [3].

| Generation number | Number of airways per generation | Airway diameter [cm] | Airway length [cm] | Gravity angle | Branching angle |
|---|---|---|---|---|---|
| 0 | 1 | 4.02 | 10.0 | 0° | 0° |
| 1 | 2 | 3.12 | 4.36 | 20° | 33° |
| 2 | 4 | 2.26 | 1.78 | 31° | 34° |
| 3 | 8 | 1.654 | 0.965 | 43° | 22° |
| 4 | 16 | 1.302 | 0.995 | 39° | 20° |
| 5 | 32 | 1.148 | 1.010 | 39° | 18° |
| 6 | 64 | 0.870 | 0.890 | 40° | 19° |
| 7 | 128 | 0.746 | 0.962 | 36° | 22° |
| 8 | 256 | 0.644 | 0.867 | 39° | 28° |
| 9 | 512 | 0.514 | 0.667 | 45° | 22° |
| 10 | 1024 | 0.396 | 0.556 | 43° | 33° |
| 11 | 2048 | 0.312 | 0.446 | 45° | 34° |
| 12 | 4096 | 0.236 | 0.359 | 45° | 37° |
| 13 | 8192 | 0.184 | 0.275 | 60° | 39° |
| 14 | 16 384 | 0.146 | 0.212 | 60° | 39° |
| 15 | 32 768 | 0.120 | 0.168 | 60° | 51° |
| 16 | 65 536 | 0.108 | 0.134 | 60° | 45° |
| 17 | 131 072 | 0.100 | 0.120 | 60° | 45° |
| 18 | 262 144 | 0.094 | 0.092 | 60° | 45° |
| 19 | 524 288 | 0.090 | 0.080 | 60° | 45° |
| 20 | 1 048 576 | 0.088 | 0.070 | 60° | 45° |
| 21 | 2 097 152 | 0.088 | 0.063 | 60° | 45° |
| 22 | 4 194 304 | 0.086 | 0.057 | 60° | 45° |
| 23 | 8 388 608 | 0.086 | 0.053 | 60° | 45° |

Table A.2 shows the characteristics of Lung Model A for an average adult with a lung volume of 4800 ml. The average male adult has a functional residual capacity of approximately 3000 ml, so the lengths and diameters of the lung model are usually scaled by a factor of $(V_{\mathrm{FRC}}/4.8\,\mathrm{l})^{1/3}$ [26]. The gravity and branching angles are generally taken as 45° and 30°, respectively [21].

Table A.2. Characteristics of the Weibel model "A" [3].

| Generation number | Number of airways per generation | Airway diameter [cm] | Airway length [cm] |
|---|---|---|---|
| 0 | 1 | 1.80 | 12.0 |
| 1 | 2 | 1.22 | 4.76 |
| 2 | 4 | 0.83 | 1.90 |
| 3 | 8 | 0.56 | 0.76 |
| 4 | 16 | 0.45 | 1.27 |
| 5 | 32 | 0.35 | 1.07 |
| 6 | 64 | 0.28 | 0.90 |
| 7 | 128 | 0.23 | 0.76 |
| 8 | 256 | 0.186 | 0.64 |
| 9 | 512 | 0.154 | 0.54 |
| 10 | 1024 | 0.130 | 0.46 |
| 11 | 2048 | 0.109 | 0.39 |
| 12 | 4096 | 0.095 | 0.33 |
| 13 | 8192 | 0.082 | 0.27 |
| 14 | 16 384 | 0.074 | 0.16 |
| 15 | 32 768 | 0.050 | 0.133 |
| 16 | 65 536 | 0.049 | 0.112 |
| 17 | 131 072 | 0.040 | 0.093 |
| 18 | 262 144 | 0.038 | 0.083 |
| 19 | 524 288 | 0.036 | 0.070 |
| 20 | 1 048 576 | 0.034 | 0.070 |
| 21 | 2 097 152 | 0.031 | 0.070 |
| 22 | 4 194 304 | 0.029 | 0.067 |
| 23 | 8 388 608 | 0.025 | 0.075 |

# B

# Steady state verification

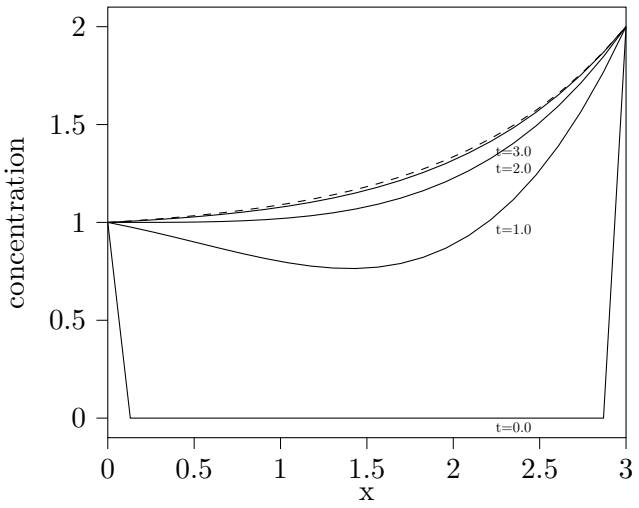



Figure B.1. The steady state solution of the advection-diffusion equation, along with the models approach to this solution at various time steps. The solution is given by (4.2), for Pe = 3.

# C

# Source code

The programme is structured in an object oriented manner, where `solution.py` contains a class that specifies how the eventual solution to the problem is stored. Initially, a constructor is called in `constants.py` that loads preset constants and passes them to `model.py`. Here, the bulk of the work is done, including the time iteration and building the discretisation matrix using `advection.py`, `diffusion.py` and `deposition.py`. The file `results.py` uses this to create figures and other results. Finally, `exact.py` is used to make the Péclet figure (it implements the analytical solution for the 1d advection-diffusion problem).

The directory structure of the files is shown below. For completeness, we also included the entire source code, starting on the next page. Most of the documentation is in the files themselves.

```
.
├── constants.py
├── exact.py
├── model.py
├── results.py
├── solution.py
└── terms
    ├── advection.py
    ├── deposition.py
    └── diffusion.py
```

## C.1. constants.py

```
"""
Physical constants and accompanying Weibel geometry.
"""


import numpy as np
import scipy.constants
from scipy import integrate


# Weibel model
gravity_angle = np.ones(24) * 45 * np.pi/180
branching_angle = np.ones(24) * 30 * np.pi/180
generations = np.arange(0, 24)
number_airways_per_generation = np.power(2, generations)
weibel_length = np.array([
    12.0e-2, 4.76e-2, 1.90e-2, 0.76e-2, 1.27e-2, 1.07e-2,
    0.90e-2, 0.76e-2, 0.64e-2, 0.54e-2, 0.46e-2, 0.39e-2,
    0.33e-2, 0.27e-2, 0.16e-2, 0.133e-2, 0.112e-2, 0.093e-2,
    0.083e-2, 0.070e-2, 0.070e-2, 0.070e-2, 0.067e-2, 0.075e-2,
])
weibel_radius = np.array([
    1.80e-2, 1.22e-2, 0.83e-2, 0.56e-2, 0.45e-2, 0.35e-2,
    0.28e-2, 0.23e-2, 0.186e-2, 0.154e-2, 0.130e-2, 0.109e-2,
    0.095e-2, 0.082e-2, 0.074e-2, 0.050e-2, 0.049e-2, 0.040e-2,
    0.038e-2, 0.036e-2, 0.034e-2, 0.031e-2, 0.029e-2, 0.025e-2
]) / 2
yeh_schum_length = np.array([
    10e-2, 4.36e-2, 1.78e-2, 0.965e-2, 0.995e-2, 1.01e-2, 0.89e-2, 0.962e-2,
    0.867e-2, 0.667e-2, 0.556e-2, 0.446e-2, 0.359e-2, 0.275e-2, 0.212e-2,
    0.168e-2, 0.134e-2, 0.12e-2, 0.092e-2, 0.08e-2, 0.07e-2, 0.063e-2,
    0.057e-2, 0.053e-2
])
yeh_schum_radius = np.array([
    2.01e-2, 1.56e-2, 1.13e-2, 0.827e-2, 0.651e-2, 0.574e-2, 0.435e-2, 0.373e-2,
    0.322e-2, 0.257e-2, 0.198e-2, 0.156e-2, 0.118e-2, 0.092e-2, 0.073e-2,
    0.06e-2, 0.054e-2, 0.05e-2, 0.047e-2, 0.045e-2, 0.044e-2, 0.044e-2,
    0.043e-2, 0.043e-2
])
yeh_schum_gravity_angle = np.array([
    0, 20, 31, 43, 39, 39, 40, 36, 39, 45, 43, 45,
    45, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60
]) * np.pi/180
yeh_schum_branching_angle = np.array([
    0, 33, 34, 22, 20, 18, 19, 22, 28, 22, 33, 34,
    37, 39, 39, 51, 45, 45, 45, 45, 45, 45, 45, 45
]) * np.pi/180
cumulative_length = np.append(0, np.cumsum(weibel_length))


# Grid
N = 20
N_zeros = np.zeros(N*len(generations))
# Time
theta = 1
```

```
dt = 0.1
t_start = 0
t_final = 5.1
# General
EPS = 1e-15
run_flags = ["diffusion", "advection", "deposition"]

# Boundary conditions
N_r = 0
right_boundary = ("neumann", N_r)
initial_condition = N_zeros

# Physics constants
boltzmann_constant = scipy.constants.Boltzmann
gravity = scipy.constants.g

# Particle constants
particle_density = 1000 # from Wikipedia
particle_diameter = 10e-6
# Fluid properties
fluid_density = 1
fluid_viscosity = 18.1e-6 # viscosity of air
fluid_gas_temperature = 293
mean_free_path = 0.066e-6

# Lung constants
functional_residual_capacity = 0.0033
tidal_volume = 0.001
breathing_rate = 12/60 # 12 breaths per minute
# We start the respiratory cycle in the inspiration phase.
breathing_pattern = lambda t: -np.cos(t*breathing_rate*2*np.pi)


class Constructor():
    """
    Represents the parameters required for airway calculations.

    All parameters are always in SI units. The eventual output is in the from
    the `self.unpack` function.

    Parameters
    ----------
    initial_condition: numpy array
        Initial concentration in the respiratory tract
    right_boundary: tuple
        Tuple of boundary type and value. Should always be Neumann and 0, stems
        from when the programme could solve arbitrary airways.
    N: int
        Number of grid points per generation
    theta: float in [0, 1]
        Parameter theta for the theta-method. A value of 0 is completely
        explicit, and a value of 1 is completely implicit.
    dt: float
        Time step size
    t_start: float
        Initial time step (should be zero)
```

```
    t_final: float
        Final time step
    fluid_density: float
        Density of air
    fluid_viscosity: float
        Viscosity of air
    fluid_gas_temperature: float
        Temperature of air
    mean_free_path: float
        Mean free path (lambda) in air
    particle_density: float
        Density of the particles (not concentration)
    particle_diameter: float
        Diameter of the particles
    generations: numpy array
        Generation numbers (0-23)
    generation_radius: numpy array
        Generation radii
    generation_length: numpy array
        Generation lengths
    number_airways_per_generation: numpy array
        Number of airways per generation
    branching_angle: numpy array
        Branching angles per generation
    gravity_angle: numpy array
        Gravity angles per generation
    functional_residual_capacity: float
        V_FRC of the patient
    tidal_volume: float
        V_T of the patient
    breathing_rate: float
        Breathing rate of the patient
    breathing_pattern: function
        periodic function, f: [0, infty] -> [-1, 1]
    run_flags: list
        List of run flags (diffusion, advection, or deposition)
    constant_diameter: bool
        Whether to include expanding lung effects
    rescale_geometry: bool
        Whether to rescale the model according to Weibel's original size.
    """
    ALVEOLI_INDEX = 16
    LEFT_CONCENTRATION = 1
    LEFT_FLUX = 0
    WEIBEL_ORIGINAL_FRC = 0.0048

    def __init__(
        self,
        initial_condition=initial_condition,
        right_boundary=right_boundary,
        N=N, # per generation
        theta=theta,
        dt=dt,
        t_start=t_start,
        t_final=t_final,
        fluid_density=fluid_density,
```

```
        fluid_viscosity=fluid_viscosity,
        fluid_gas_temperature=fluid_gas_temperature,
        mean_free_path=mean_free_path,
        particle_density=particle_density,
        particle_diameter=particle_diameter,
        generations=generations,
        generation_radius=weibel_radius,
        generation_length=weibel_length,
        number_airways_per_generation=number_airways_per_generation,
        branching_angle=branching_angle,
        gravity_angle=gravity_angle,
        functional_residual_capacity=functional_residual_capacity,
        tidal_volume=tidal_volume,
        breathing_rate=breathing_rate,
        breathing_pattern=breathing_pattern,
        run_flags=run_flags,
        constant_diameter=False,
        rescale_geometry=True,
    ):
        args = locals().copy()
        del args['self']
        for key, value in args.items():
            setattr(self, key, value)
        # Variables constant for the duration of the simulation
        self.T = np.concatenate([np.arange(self.t_start, self.t_final, self.dt),
                                 [self.t_final]])
        if self.rescale_geometry:
            self.scale_geometry()
        self.cumulative_length = np.append(
            0, np.cumsum(self.generation_length))
        self.x = np.concatenate([
            np.linspace(self.cumulative_length[i], self.cumulative_length[i+1],
                        self.N, endpoint=False)
            for i in range(len(self.cumulative_length)-1)])
        # The grid spacing below is edge based: i.e. the spacing is defined by
        # which edge is in the "center".
        self.grid_spacing = self.x[1:]-self.x[:-1]
        self.lung_volume = self.get_lung_volume()
        self.A_A = self.get_generation_number(
            self.number_airways_per_generation
            * np.pi
            * np.power(self.generation_radius, 2)
        )
        # Variables that can change for each time step during the simulation
        self.airway_diameter_all = self.get_airway_diameter()
        if self.constant_diameter:
            self.airway_diameter_all[:, :] = self.generation_radius*2
        self.A_T_all = self.get_generation_number(
            self.number_airways_per_generation
            * np.pi
            * np.power(self.airway_diameter_all, 2)
            / 4
        )
        self.inlet_velocity = self.get_inlet_velocity()
        self.velocity_all = self.get_velocity_profile()
        self.left_boundary_all = self.get_left_boundary()
```

```
224         self.initial_condition = self.initial_condition
225         self.cunningham_slip_correction = self.get_cunningham_slip_correction()
226         self.particle_relaxation_time = self.get_particle_relaxation_time()
227         self.gravity_angle = self.get_generation_number(self.gravity_angle)
228         self.terminal_settling_velocity = self.get_terminal_settling_velocity()
229         self.gravitational_settling_velocity = (
230             self.get_gravitational_settling_velocity())
231         self.brownian_diffusion_coefficient = self.get_brownian_diffusion_coefficient()
232 
233     def scale_geometry(self):
234         """
235         Scale geometry according to Weibels original lung volume of 4.8 litre
236         """
237         self.generation_radius = self.generation_radius * np.power(
238             self.functional_residual_capacity / self.WEIBEL_ORIGINAL_FRC, 1/3)
239         self.generation_length = self.generation_length * np.power(
240             self.functional_residual_capacity / self.WEIBEL_ORIGINAL_FRC, 1/3)
241 
242     def get_generation_number(self, quantity):
243         """
244         Quantity extended to the grid points instead of generation numbers
245 
246         Take a quantity of length 24 (amount of generations), and conform it to
247         `self.x`, while taking into account which generation each value is in.
248 
249         If the quantity is 2-dimensional, the function is applied recursively
250         (row-wise), and the output will be 2-dimensional.
251 
252         Parameters
253         ----------
254         Quantity: numpy array
255             Quantity on generation numbers
256 
257         Returns
258         -------
259         numpy array
260             Quantity on grid points
261         """
262         if len(quantity.shape) > 1:
263             return np.array([self.get_generation_number(quantity[i, :])
264                              for i in range(quantity.shape[0])])
265         bin = np.digitize(self.x, self.cumulative_length)-1
266         return quantity[bin]
267 
268     def get_lung_volume(self):
269         """
270         Calculate the lung volume at time steps `self.T`
271 
272         Returns
273         -------
274         numpy array
275             Lung volume
276         """
277         lung_volume = (self.functional_residual_capacity
278                        + self.tidal_volume/2 * (1+self.breathing_pattern(self.T)))
279         return lung_volume
```

```
280
281     def get_airway_diameter(self):
282         """
283         Calculate the airway diameter due to breathing mechanics.
284
285         Returns
286         -------
287         numpy array
288             Airway diameter at every x and t.
289         """
290         airway_diameter = self.generation_radius * 2
291         d_T = np.zeros([len(self.lung_volume), len(airway_diameter)])
292         d_T[:, :] = np.copy(airway_diameter)
293         # Airway diameter only varies with time from generation >= ALVEOLI_INDEX
294         # (alveolated airways)
295         d_T[:, self.ALVEOLI_INDEX:] = (
296             d_T[:, self.ALVEOLI_INDEX:]
297             * np.power(self.lung_volume/self.functional_residual_capacity, 1/3)
298             .reshape(-1, 1))
299         return d_T
300
301     def get_alveoli_index(self):
302         """
303         Find the alveoli index on the spatial grid.
304
305         Returns
306         -------
307         int
308             index where the alveoli start
309         """
310         return np.nonzero(
311             (self.x > self.cumulative_length[self.ALVEOLI_INDEX]) == 1)[0][0]
312
313     def get_inlet_velocity(self):
314         """
315         Calculate the inlet velocity at all time steps
316
317         Returns
318         -------
319         numpy array
320             Inlet velocity at all time steps
321         """
322         volume_derivative = np.gradient(self.lung_volume, self.dt)
323         inlet_velocity = volume_derivative / self.A_A[0]
324         return inlet_velocity
325
326     def get_velocity_profile(self):
327         """
328         Calculate the velocity profile u(x, t)
329
330         Returns
331         -------
332         numpy array
333             Velocity profile at all x and t
334         """
335         alveoli_index = self.get_alveoli_index()
```

```
336         copy_array = self.inlet_velocity.reshape(-1, 1) * self.A_A[0]
337         u = np.hstack(
338             [np.copy(copy_array) for i in range(self.A_T_all.shape[1])])
339         # For the time dependent part:
340         deriv = np.gradient(self.A_T_all[:, alveoli_index:], self.dt, axis=0)
341         u /= self.A_A
342         for idx_t in range(len(self.T)):
343             for idx_x in range(alveoli_index, len(self.x)):
344                 u[idx_t, idx_x] = (u[idx_t, alveoli_index-1] * self.A_A[alveoli_index-1]
345                                    - integrate.trapz(
346                     deriv[idx_t, :idx_x-alveoli_index+1],
347                     self.x[alveoli_index:idx_x+1]
348                 )) / self.A_A[idx_x]
349         return u
350 
351     def get_left_boundary(self):
352         """
353         Determine the left boundary condition at all time steps
354 
355         The boundary condition changes from inspiration to expiration and vice
356         versa
357 
358         Returns
359         -------
360         list of tuples
361             Left boundary condition at all time steps.
362         """
363         left_boundary_all = list(self.velocity_all[:, 0] >= 0)
364         left_boundary_all = [("dirichlet", self.LEFT_CONCENTRATION)
365                              if bc else ("neumann", self.LEFT_FLUX)
366                              for bc in left_boundary_all]
367         return left_boundary_all
368 
369     def get_particle_relaxation_time(self):
370         """
371         Calculate particle relaxation time.
372 
373         Returns
374         -------
375         float
376             particle relaxation time
377         """
378         # https://aerosol.ees.ufl.edu/aerosol_trans/section07.html
379         return (self.particle_density * np.power(self.particle_diameter, 2)
380                 * self.cunningham_slip_correction
381                 / (18 * self.fluid_viscosity))
382 
383     def get_cunningham_slip_correction(self):
384         """
385         Calculate Cunningham slip correction factor.
386 
387 
388         Returns
389         -------
390         float
391             Cunningham slip correction factor
```

```
392         """
393         # https://aerosol.ees.ufl.edu/aerosol_trans/section06_c.html
394         cunningham_slip_correction = (
395             1 + (self.mean_free_path/self.particle_diameter)
396             * (2.34 + 1.05 * np.exp(
397                 -0.39 * self.particle_diameter/self.mean_free_path)))
398         return cunningham_slip_correction
399 
400     def get_gravitational_settling_velocity(self):
401         """
402         Calculate the gravitational settling velocity.
403 
404         Returns
405         -------
406         numpy array
407             Gravitational settling velocity for all x
408         """
409         return self.terminal_settling_velocity * np.sin(self.gravity_angle)
410 
411     def get_terminal_settling_velocity(self):
412         """
413         Calculate the terminal settling velocity.
414 
415         Returns
416         -------
417         float
418             Terminal settling velocity
419         """
420         return (self.particle_density * self.particle_diameter**2 * gravity
421                 * self.cunningham_slip_correction / (18 * self.fluid_viscosity))
422 
423     def get_brownian_diffusion_coefficient(self):
424         """
425         Calculate the Brownian diffusion coefficient from Stokes-Einsten equation.
426 
427         Returns
428         -------
429         float
430             Brownian diffusion coefficient
431         """
432         return ((boltzmann_constant * self.fluid_gas_temperature
433                  * self.cunningham_slip_correction)
434                 / (3 * np.pi * self.fluid_viscosity * self.particle_diameter))
435 
436     def unpack(self):
437         """
438         Unpack all constants for use in Airway or ExactAirway
439 
440         Returns
441         -------
442         dict
443             Dict of parameters
444         """
445         boundaries = {
446             'initial_condition': self.initial_condition,
447             'left_boundary_all': self.left_boundary_all,
```

```
448             'right_boundary': self.right_boundary,
449         }
450         space_discretisation = {
451             'x': self.x,
452             'grid_spacing': self.grid_spacing,
453         }
454         time_integration = {
455             'theta': self.theta,
456             'T': self.T,
457             'dt': self.dt,
458         }
459         general = {
460             'A_T_all': self.A_T_all,
461             'A_A': self.A_A,
462             'velocity_all': self.velocity_all,
463             'fluid_density': self.fluid_density,
464             'airway_diameter_all': self.get_generation_number(
465                 self.airway_diameter_all),
466         }
467         meta = {
468             'run_flags': self.run_flags,
469         }
470         deposition = {
471             'number_airways_per_generation': self.get_generation_number(
472                 self.number_airways_per_generation),
473             'branching_angle': self.get_generation_number(self.branching_angle),
474             'generation_length': self.get_generation_number(self.generation_length),
475             'cumulative_length': self.get_generation_number(self.cumulative_length),
476             'fluid_viscosity': self.fluid_viscosity,
477             'gravitational_settling_velocity': self.gravitational_settling_velocity,
478             'particle_relaxation_time': self.particle_relaxation_time,
479             'brownian_diffusion_coefficient': self.brownian_diffusion_coefficient,
480             'breathing_rate': self.breathing_rate,
481         }
482         return {
483             'boundaries': boundaries,
484             'space_discretisation': space_discretisation,
485             'time_integration': time_integration,
486             'general': general,
487             'meta': meta,
488             'deposition': deposition,
489         }
```

## C.2. model.py

```
"""
This module can be used to simulate the inhalation of particles in the lungs,
using a 1-D advection-diffusion model.
"""


import numpy as np
from scipy.sparse import dia_matrix, identity, SparseEfficiencyWarning
from scipy.sparse.linalg import spsolve
from scipy import integrate
import solution as da
from terms.advection import Advection
from terms.diffusion import Diffusion
from terms.deposition import Deposition

import warnings
warnings.simplefilter('ignore', SparseEfficiencyWarning)


class Airway(da.Solution):
    """
    Represents an airway in the lungs using a Weibel-like geometry.

    Takes the output of `Constructor.unpack()` as arguments.
    """

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.total_deposition = []
        self.A_A_interface = self.grid_to_interface(self.A_A)
        self.initialise_time_dependent_variables()
        self.sol = self.make_df(self.theta_method())

    def initialise_time_dependent_variables(self):
        self.velocity = self.velocity_all[0, :]
        self.v_interface = self.grid_to_interface(self.velocity)
        self.airway_diameter = self.airway_diameter_all[0, :]
        self.left_boundary = self.left_boundary_all[0]
        self.A_T = self.A_T_all[0, :]
        self.A_T_interface = self.grid_to_interface(self.A_T_all[0, :])
        self.diffusion_constant = self.get_effective_diffusion_coefficient()
        self.D_interface = self.grid_to_interface(self.diffusion_constant)

    def grid_to_interface(self, grid_values):
        """
        Take grid points values and turn them into values specified on interfaces.

        Parameters
        ----------
        grid_values: numpy array
            Grid values

        Returns
        -------
        numpy array
```

```
56          Values specified on interface
57      """
58      # This function used to calculate the harmonic mean. This was changed
59      # to ensure a mass balance when stepping over generation boundaries.
60      return grid_values[:-1]
61
62  def build_discretisation_matrix(self):
63      """
64      Build the discretisation matrix
65
66      Returns
67      -------
68      numpy array
69          Discretisation matrix A
70      numpy array
71          Discretisation vector b
72      """
73      N = len(self.x)
74      A = np.zeros([3, N])
75      b = np.zeros(N)
76      # OPTIMIZE: This can probably be optimized if every separate term does
77      # not make its own matrix, and instead adds to a given one.
78      if "diffusion" in self.run_flags:
79          diffusion = Diffusion(
80              self.x, self.D_interface, self.A_T_interface,
81              self.left_boundary, self.right_boundary, self.grid_spacing
82          )
83          A += diffusion.A
84          b += diffusion.b
85      if "advection" in self.run_flags:
86          advection = Advection(
87              self.x, self.v_interface, self.fluid_density, self.A_A_interface,
88              self.left_boundary, self.right_boundary
89          )
90          A += advection.A
91          b += advection.b
92      if "deposition" in self.run_flags:
93          deposition = Deposition(
94              self.x, self.airway_diameter,
95              self.number_airways_per_generation, self.fluid_viscosity,
96              self.velocity, self.fluid_density, self.branching_angle,
97              self.generation_length, self.cumulative_length,
98              self.grid_spacing, self.gravitational_settling_velocity,
99              self.particle_relaxation_time,
100             self.diffusion_constant, self.brownian_diffusion_coefficient
101         )
102         A += deposition.A
103         b += deposition.b
104         # For deposition calculations:
105         self.deposition_coefficients = deposition.A[1, :]
106     A, b = self.add_boundary_conditions(A, b)
107     offsets = [-1, 0, 1]
108     A = dia_matrix((A, offsets), shape=(A.shape[1], A.shape[1]))
109     return (A, b)
110
111 def add_boundary_conditions(self, A, b):
```

```
112         """
113         Add boundary conditions to A and b.
114 
115         When the boundary condition is of type Neumann, advection is disregarded
116         depending on whether the velocity is positive or negative (because we
117         are using the upwind scheme).
118 
119         For a boundary condition of type Dirichlet, the matrix is not resized.
120         Instead, the effect of the boundary is placed in vector b, and the effect
121         of the boundary in the matrix A is set to zero. The Dirichlet conditions
122         are added later.
123 
124         Parameters
125         ----------
126         A: numpy array
127             Discretisation matrix A
128         b: numpy array
129             Discretisation vector b
130 
131         Returns
132         -------
133         numpy array
134             Discretisation matrix A with boundary conditions
135         numpy array
136             Discretisation vector b with boundary conditions
137 
138         See Also
139         --------
140         fix_boundaries: Add Dirichlet conditions.
141         """
142         left = 0
143         right = 0
144         if self.left_boundary[0] == "dirichlet":
145             if "diffusion" in self.run_flags:
146                 left += self.A_T_interface[0] * \
147                     self.D_interface[0] / self.grid_spacing[0]
148             if "advection" in self.run_flags and self.velocity[0] > 0:
149                 left += self.A_A_interface[0] * \
150                     self.velocity[0] * self.fluid_density
151             b[1] = left * self.left_boundary[1]
152             A[0, 0] = 0
153         elif self.left_boundary[0] == "neumann" and "diffusion" in self.run_flags:
154             b[0] = self.left_boundary[1]
155         if self.right_boundary[0] == "dirichlet":
156             if "diffusion" in self.run_flags:
157                 right += self.A_T_interface[-1] * \
158                     self.D_interface[-1] / self.grid_spacing[-1]
159             if "advection" in self.run_flags and self.velocity[0] < 0:
160                 right -= self.A_A_interface[-1] * \
161                     self.velocity[-1] * self.fluid_density
162             b[-2] = right * self.right_boundary[1]
163             A[2, -1] = 0
164         elif self.right_boundary[0] == "neumann" and "diffusion" in self.run_flags:
165             b[-1] = self.right_boundary[1]
166         return (A, b)
167 
```

```
168     def handle_grid_spacing(self):
169         """
170         Return grid spacing with a mesh based centre.
171 
172         This grid spacing, unlike self.grid_spacing is mesh based. That is, the
173         spacing has a meshpoint in the center instead of an edge.
174 
175         Returns
176         -------
177         numpy array
178             mesh based grid spacing
179         """
180         dxs = 0.5*(self.grid_spacing[1:] + self.grid_spacing[:-1])
181         dxs = np.concatenate(([0.5*dxs[0]], dxs, [0.5*dxs[-1]]))
182         return dxs
183 
184     def get_total_deposition(self):
185         """
186         Calculate the local (per generation) deposition per time step.
187 
188         Returns
189         -------
190         list of numpy arrays
191             local deposition per time step
192         """
193         w = np.copy(self.w)
194         dxs = self.handle_grid_spacing()
195 
196         bin = np.digitize(self.x, self.cumulative_length)-1
197         change_indices = np.concatenate(
198             ([0],
199              np.where(bin[:-1] != bin[1:])[0]+1,
200              [-2])
201         )
202         change_pairs = zip(change_indices[:-1], change_indices[1:]+1)
203         local_deposition = []
204         for pair in change_pairs:
205             local_deposition.append(
206                 integrate.trapz(
207                     (-self.deposition_coefficients[slice(*pair)]
208                      * w[slice(*pair)] / dxs[slice(*pair)]),
209                     self.x[slice(*pair)])
210             )
211         self.total_deposition.append(local_deposition)
212 
213     def get_effective_diffusion_coefficient(self):
214         """
215         Calculate effective diffusion coefficient.
216 
217         Returns
218         -------
219         numpy array
220             Effective diffusion coefficient
221         """
222         equals_array = (self.velocity == 0)
223         greater_array = (self.velocity > 0)
```

```
224         smaller_array = (self.velocity < 0)
225         return abs(equals_array * self.brownian_diffusion_coefficient
226                    + greater_array * (self.brownian_diffusion_coefficient
227                                       + (1.08 * self.velocity * self.airway_diameter))
228                    + smaller_array * (self.brownian_diffusion_coefficient
229                                       + (0.37 * self.velocity * self.airway_diameter))
230                    )
231
232     def set_time_dependent_terms(self, time_index):
233         self.A_T = self.A_T_all[time_index, :]
234         self.A_T_interface = self.grid_to_interface(
235             self.A_T_all[time_index, :])
236         self.airway_diameter = self.airway_diameter_all[time_index, :]
237         self.velocity = self.velocity_all[time_index, :]
238         self.v_interface = self.grid_to_interface(self.velocity)
239         self.diffusion_constant = self.get_effective_diffusion_coefficient()
240         self.D_interface = self.grid_to_interface(self.diffusion_constant)
241         self.left_boundary = self.left_boundary_all[time_index]
242
243     def fix_boundaries(self):
244         """
245         Add Dirichlet conditions to `self.w`
246         """
247         if self.left_boundary[0] == "dirichlet":
248             self.w[0] = self.left_boundary[1]
249         if self.right_boundary[0] == "dirichlet":
250             self.w[-1] = self.right_boundary[1]
251
252     def theta_method(self):
253         """
254         Solve the boundary value problem with the theta method.
255
256         For details and readability, consult the thesis.
257
258         Returns
259         -------
260         numpy array
261             Concentration of aerosol at every x and t
262         """
263         self.w = self.initial_condition
264         self.N_theta = len(self.w)
265         self.I_theta = identity(self.N_theta, format='dia')
266         w_array = [self.w]
267         self.spacing_reciprocal = 1/self.handle_grid_spacing()
268         A, b = self.build_discretisation_matrix()
269         self.fix_boundaries()
270         if "deposition" in self.run_flags:
271             self.get_total_deposition()
272         for idx_t in range(1, len(self.T)-1):
273             w_old = np.copy(self.w)
274             A_old, b_old = A.copy(), np.copy(b)
275             A_T_old = np.copy(self.A_T)
276             self.set_time_dependent_terms(idx_t)
277             A, b = self.build_discretisation_matrix()
278             factor = dia_matrix((self.A_T, 0), shape=(len(self.A_T), len(self.A_T))) \
279                 - (self.dt * self.theta
```

```
                    * A.multiply(self.spacing_reciprocal[:, np.newaxis]))
            inv_factor = spsolve(factor, self.I_theta)
            self.w = inv_factor.dot(np.ravel(
                (dia_matrix((A_T_old, 0), shape=(len(A_T_old), len(A_T_old)))
                 + self.dt * (1-self.theta)
                 * A_old.multiply(self.spacing_reciprocal[:, np.newaxis]))
                .dot(w_old)
                + (self.dt * self.spacing_reciprocal
                   * (self.theta * b + (1-self.theta) * b_old))))
            self.fix_boundaries()
            w_array.append(self.w)
            if "deposition" in self.run_flags:
                self.get_total_deposition()
        w_array = np.vstack(w_array)
        return w_array


if __name__ == "__main__":
    from constants import Constructor
    numerical_solution = Airway(**Constructor().unpack())
    numerical_solution.surface_plot()
    print(numerical_solution.deposition_fraction())
```

## C.3. solution.py

```
"""
This module is used to visualise the results of the solution of the 1D
advection-diffusion equation
"""


import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import pandas as pd
from scipy import integrate
from constants import cumulative_length


class Solution():
    """A solution of a 1d advection-diffusion equation.

    The solution is stored in a pandas DataFrame and has various methods to
    display it.

    Parameters
    ----------
    boundaries: dict
    space_discretisation: dict
    time_integration: dict
    general: dict
    meta: dict
    deposition: dict

    Returns
    -------
    Solution of 1d advection diffusion equation in human lungs

    """

    def __init__(
        self,
        boundaries=None,
        space_discretisation=None,
        time_integration=None,
        general=None,
        meta=None,
        deposition=None
    ):
        for dict in [boundaries, space_discretisation, time_integration,
                     general, meta]:
            for key, value in dict.items():
                setattr(self, key, value)
        if deposition is not None:
            for key, value in deposition.items():
                setattr(self, key, value)
        self.sol = None

    def make_df(self, mat):
        """
```

```
56          Create solution DataFrame
57
58          Returns
59          -------
60          a pandas DataFrame
61          """
62          # We have to remove the last time point, because we cannot calculate
63          # this using an implicit method, which is why the iteration stops one
64          # time step early in theta_method in model.py
65          self.T = self.T[:-1]
66          return pd.DataFrame(mat, index=self.T, columns=self.x)
67
68      def surface_plot(self):
69          """
70          Show a 3d surface plot
71          """
72          fig = plt.figure()
73          ax = fig.gca(projection='3d')
74          X, T = np.meshgrid(self.x, self.T)
75          surf = ax.plot_surface(X, T, self.sol.to_numpy(), cmap='viridis')
76          fig.colorbar(surf)
77          ax.set_xlabel('x')
78          ax.set_ylabel('t')
79          ax.set_zlabel('N')
80          ax.set_zlim(-1, 2)
81          plt.show()
82
83      def time_evolution_plot(self, time_steps=0, times=[],
84                              show_generations=False):
85          """
86          Show a time evolution plot of the concentration
87
88          Parameters
89          ----------
90          time_steps: int
91              Number of time steps to plot
92          times: list
93              Exact time steps to plot
94          show_generations: bool
95              Whether to show generation boundaries
96          """
97          if ((time_steps == 0 and len(times) == 0)
98                  or (time_steps != 0 and len(times) != 0)):
99              raise ValueError(
100                 "should give exactly one of time_steps or times arguments")
101         fig = plt.figure()
102         ax = fig.gca()
103         if len(times) == 0:
104             times = [self.sol.index[i*self.sol.shape[0] //
105                                     max(time_steps-1, 0)] for i in range(time_steps-1)]
106             times.append(self.sol.index[-1])
107         ilocs = [
108             self.sol.index.get_loc(
109                 time,
110                 method='nearest') for time in times]
111         ax.plot(self.sol.T.iloc[:, ilocs], color='k', linewidth=1)
```

```
112         create_labels(times, self.sol, ax)
113         ax.set_xlabel('x')
114         ax.set_ylabel('concentration')
115         ax.set_xlim([min(self.sol.columns), max(self.sol.columns)])
116         if show_generations:
117             show_plot_generations(ax)
118         plt.show()
119 
120     def deposition_fraction(self, offset=0.0):
121         """
122         Get the deposition fractions for a breathing cycle.
123 
124         Parameters
125         ----------
126         offset: float
127             Start calculation at this time step
128 
129         Returns
130         -------
131         dict
132             Dictionary containing the local and total deposition fractions
133         """
134         period = 1/self.breathing_rate
135         t_half = self.sol.loc[offset:offset+period/2].index
136         N0 = self.sol.loc[offset:offset+period/2, 0]
137         offset_idx = self.sol.index.get_loc(offset)
138         v = self.velocity_all[offset_idx:offset_idx+len(N0), 0]
139         total_RT = integrate.trapz(N0 * self.A_A[0] * v, t_half)
140 
141         t = self.sol.loc[offset:offset+period].index
142         local_deposition = []
143         deposition_array = np.array(self.total_deposition)
144         for gen in range(24):
145             local_deposition.append(
146                 integrate.trapz(
147                     deposition_array[offset_idx:+offset_idx+len(t), gen], t)
148             )
149         local_deposition = np.array(local_deposition)
150         total_deposition = sum(local_deposition)
151         return {
152             "total_deposition": total_deposition,
153             "total_RT": total_RT,
154             "deposition_fraction": total_deposition/total_RT,
155             "local_deposition_fraction": local_deposition / total_RT,
156         }
157 
158 
159 def create_labels(times, solution, axis, x_index=-1):
160     """
161     Create solution labels on a particular figure
162 
163     Parameters
164     ----------
165     times: list
166         Times steps for the labels
167     solution: Solution DataFrame
```

```
168         DataFrame containing solution (`Solution.sol`)
169     axis: matplotlib axis type
170         Matplotlib axis to plot on
171     x_index: int
172         The X index for the labels
173 
174     Returns
175     -------
176     None
177     """
178     coords = solution.loc[times, solution.columns[x_index]]
179     # to make the key a string
180     coords = {
181         "t={:.3}".format(key): value for key, value in zip(
182             coords.index.map(str),
183             coords.values)}
184     for key, value in coords.items():
185         axis.text(float(solution.columns[x_index]), value, key,
186                   horizontalalignment='right', verticalalignment='top')
187 
188 
189 def show_plot_generations(axis):
190     """
191     Create generation indicators on a particular figure
192 
193     Parameters
194     ----------
195     axis: matplotlib axis type
196         Matplotlib axis to plot on
197 
198     Returns
199     -------
200     None
201     """
202     y = axis.get_ylim()[1]
203     # Alternate vertical alignment to save space in later generations
204     verticalalignment = ['bottom', 'top']
205     # BUG: cumulative_length is not dynamically obtained, which means that you
206     # cannot show plot generations using Yeh and Schum geometry or when using
207     # rescale_geometry=True (which is the default!)
208     for idx, x in enumerate(cumulative_length):
209         if idx < 8:
210             va = verticalalignment[0]
211         else:
212             va = verticalalignment[idx % 2]
213         axis.axvline(x=x, linestyle=':', color='k', linewidth=0.5)
214         if ((idx > 11) and (idx % 3 != 0)) or (idx == 24):
215             # We only plot some generation labels, to avoid clutter.
216             continue
217         axis.text(x=x, y=y, s="{}".format(idx), fontsize='small',
218                   horizontalalignment='center', verticalalignment=va)
```

## C.4. terms/advection.py

```
"""
This module contains the discretisation matrix for advection of the 1d general
dynamic aerosol equation.
"""


import numpy as np


class Advection:
    """Represents a advection discretisation matrix

    Parameters
    ----------
    x: numpy array
        Grid points
    velocity: numpy array
        fluid velocity
    density: float
        fluid density
    A_A: numpy array
        Cross sectional area of all airways
    left_boundary: tuple
        Boundary condition on the left side
    right_boundary: tuple
        Boundary condition on the right side

    Returns
    -------
    numpy array
        Advection matrix, A, in diagonal ordered form
    numpy array
        Advection vector, b, for 1D advection, using the upwind scheme
    """

    def __init__(self, x, velocity, density, A_A,
                 left_boundary, right_boundary):
        self.x = x
        self.velocity = velocity
        self.density = density
        self.A_A = A_A
        self.left_boundary = left_boundary
        self.right_boundary = right_boundary
        self.A, self.b = self.sparse_advection_matrix()

    def __repr__(self):
        data = {
            'x': self.x,
            'velocity': self.velocity,
            'density': self.density,
            'A_A': self.A_A,
            'left_boundary': self.left_boundary,
            'right_boundary': self.right_boundary,
        }
        return ("Diffusion({x}, {velocity}, {density}, {A_A}, {left_boundary}, "
```

```
56              "{right_boundary})").format(**data)
57
58      def __str__(self):
59          return (self.A, self.b)
60
61      def sparse_advection_matrix(self):
62          N = len(self.x)
63          A = np.zeros([3, N])
64          b = np.zeros(N)
65          # interior points
66          coefficients = self.A_A * self.velocity * self.density
67          # Take only the first element of velocity. We assume that velocity has
68          # the same sign everywhere.
69          if self.velocity[0] == 0:
70              return (A, b)
71          elif self.velocity[0] > 0:
72              # west
73              A[0, :-1] = coefficients
74              # interior
75              A[1, :-1] = -coefficients
76              # The boundary point has no coefficient in the upstream model, so we
77              # just copy the one before that.
78              A[1, -1] = -coefficients[-1]
79          elif self.velocity[0] < 0:
80              coefficients *= -1
81              # east
82              A[2, 1:] = coefficients
83              # interior
84              A[1, 1:] = -coefficients
85              # The boundary point has no coefficient in the upstream model, so we
86              # just copy the one before that.
87              A[1, 0] = -coefficients[0]
88          return (A, b)
```

## C.5. terms/diffusion.py

```
"""
This module contains the discretisation matrix for diffusion of the 1d general
dynamic aerosol equation.
"""


import numpy as np


class Diffusion:
    """Represents a diffusion discretisation matrix

    Parameters
    ----------
    x: numpy array
        Grid points
    D_interface: numpy array
        Diffusion constants for faces between grid points
    A_T: numpy array
        Cross sectional area of all airways
    left_boundary: tuple
        Boundary condition on the left side
    right_boundary: tuple
        Boundary condition on the right side
    grid_spacing: numpy array
        Mesh grid distance

    Returns
    -------
    numpy array
        Diffusion matrix, A, in diagonal ordered form
    numpy array
        Diffusion vector, b, for 1D diffusion
    """

    def __init__(self, x, D_interface, A_T, left_boundary, right_boundary,
                 grid_spacing):
        self.x = x
        self.D_interface = D_interface
        self.A_T = A_T
        self.left_boundary = left_boundary
        self.right_boundary = right_boundary
        self.grid_spacing = grid_spacing
        self.A, self.b = self.sparse_diffusion_matrix()

    def __repr__(self):
        data = {
            'x': self.x,
            'D_interface': self.D_interface,
            'A_T': self.A_T,
            'left_boundary': self.left_boundary,
            'right_boundary': self.right_boundary,
            'grid_spacing': self.grid_spacing,
        }
        return ("Diffusion({x}, {D_interface}, {A_T}, {left_boundary}, "
```

```
56                 "{right_boundary}, {grid_spacing})").format(**data)
57
58     def __str__(self):
59         return (self.A, self.b)
60
61     def sparse_diffusion_matrix(self):
62         N = len(self.x)
63         A = np.zeros([3, N])
64         b = np.zeros(N)
65         # interior points
66         coefficients = self.A_T * self.D_interface / self.grid_spacing
67         # west
68         A[0, :-1] = coefficients
69         # east
70         A[2, 1:] = coefficients
71         # interior
72         A[1, :] = -(A[0, :] + A[2, :])
73         return (A, b)
```

## C.6. terms/deposition.py

```
"""
This module contains methods for the deposition of particles in the aerosol
dynamic equation.
"""


import numpy as np
from constants import gravity, boltzmann_constant, EPS
import warnings


class Deposition:
    """Represents a Deposition discretisation matrix

    Parameters
    ----------
    x: numpy array
        Grid points
    airway_diameter: numpy array
        Diameter of RT at every x
    number_airways_per_generation: numpy array
        Number of airways per generation
    fluid_viscosity: float
        Viscosity of air
    velocity: numpy array
        Velocity of at every x
    fluid_density: float
        Density of air
    branching_angle: numpy array
        Branching angles per generation
    generation_length: numpy array
        Length of the generations
    cumulative_length: numpy array
        Cumulative generation lengths
    grid_spacing: numpy array
        Grid spacing
    gravitational_settling_velocity: numpy array
        Gravitational settling velocity for all x
    particle_relaxation_time: float
        Relaxation time of the particle
    effective_diffusion_coefficient: numpy array
        Diffusion coefficients for all x
    brownian_diffusion_coefficient: float
        Brownian diffusion coefficient for all x

    Returns
    -------
    numpy array
        Deposition matrix, A, in diagonal ordered form
    numpy array
        Deposition vector, b
    """

    def __init__(self, x, airway_diameter, number_airways_per_generation,
                 fluid_viscosity, velocity, fluid_density, branching_angle,
```

```
56                generation_length, cumulative_length, grid_spacing,
57                gravitational_settling_velocity, particle_relaxation_time,
58                effective_diffusion_coefficient,
59                brownian_diffusion_coefficient):
60        self.x = x
61        self.airway_diameter = airway_diameter
62        self.number_airways_per_generation = number_airways_per_generation
63        self.fluid_viscosity = fluid_viscosity
64        self.velocity = velocity
65        self.fluid_density = fluid_density
66        self.branching_angle = branching_angle
67        self.generation_length = generation_length
68        self.cumulative_length = cumulative_length
69        self.grid_spacing = grid_spacing
70        self.gravitational_settling_velocity = gravitational_settling_velocity
71        self.particle_relaxation_time = particle_relaxation_time
72        self.effective_diffusion_coefficient = effective_diffusion_coefficient
73        self.brownian_diffusion_coefficient = brownian_diffusion_coefficient
74        self.wetted_perimeter = self.get_wetted_perimeter()
75        # Brownian diffusion velocity
76        if np.any(abs(self.velocity) < EPS) == 0:
77            # This if statement is to avoid inf values when dividing by
78            # velocity. In that case we assume that the diffusion velocity is
79            # 0.
80            self.reynolds_number = self.get_reynolds_number()
81            self.schmidt_number = self.get_schmidt_number()
82            self.dimensionless_length = self.get_dimensionless_length()
83            self.sherwood_number = self.get_sherwood_number()
84            self.diffusion_velocity = self.get_brownian_diffusion_velocity()
85        else:
86            self.diffusion_velocity = 0
87        # Impaction velocity
88        self.stokes_number = self.get_stokes_number()
89        self.impact_velocity = self.get_impact_velocity()
90        self.deposition_velocity = sum([
91            self.gravitational_settling_velocity,
92            self.diffusion_velocity,
93            self.impact_velocity
94        ])
95        self.A, self.b = self.sparse_deposition_matrix()
96
97    def __repr__(self):
98        data = {
99            'x': self.x,
100           'airway_diameter': self.airway_diameter,
101           'number_airways_per_generation': self.number_airways_per_generation,
102           'fluid_viscosity': self.fluid_viscosity,
103           'velocity': self.velocity,
104           'fluid_density': self.fluid_density,
105           'branching_angle': self.branching_angle,
106           'generation_length': self.generation_length,
107           'cumulative_length': self.cumulative_length,
108           'grid_spacing': self.grid_spacing,
109           'gravitational_settling_velocity': self.gravitational_settling_velocity,
110           'particle_relaxation_time': self.particle_relaxation_time,
111           'effective_diffusion_coefficient': self.effective_diffusion_coefficient,
```

```
            'brownian_diffusion_coefficient': self.brownian_diffusion_coefficient,
        }
        return ("Deposition({x}, {airway_diameter},"
                " {number_airways_per_generation}, {fluid_viscosity},"
                " {velocity}, {fluid_density}, {branching_angle},"
                " {generation_length}, {cumulative_length}, {grid_spacing},"
                " {gravitational_settling_velocity},"
                " {particle_relaxation_time},"
                " {effective_diffusion_coefficient},"
                " {brownian_diffusion_coefficient})").format(**data)

    def __str__(self):
        return (self.A, self.b)

    def get_wetted_perimeter(self):
        """
        Calculate the wetted wetted perimeter.

        Returns
        -------
        numpy array
            Wetted perimeter
        """
        return (self.number_airways_per_generation *
                np.pi * self.airway_diameter)

    def get_brownian_diffusion_velocity(self):
        """
        Calculate the Brownian diffusion velocity.

        Returns
        -------
        numpy array
            Brownian diffusion velocity
        """
        return self.brownian_diffusion_coefficient * \
            self.sherwood_number / self.airway_diameter

    def get_dimensionless_length(self):
        """
        Calculate the dimensionless length.

        Returns
        -------
        numpy array
            Dimensionless length
        """
        # This is different than what the paper says. The paper takes the total
        # x, that is, from the beginning of the RT.
        absolute_distance = self.x - self.cumulative_length
        return (absolute_distance / (self.airway_diameter *
                                     self.reynolds_number * self.schmidt_number))

    def get_sherwood_number(self):
        """
        Calculate Sherwood's number.
```

```
168
169         Returns
170         -------
171         numpy array
172             Sherwood number
173         """
174         sherwood_number = np.zeros(len(self.x))
175         threshold = 0.01
176         smaller_than_indices = np.where(self.dimensionless_length <= threshold)
177         larger_than_indices = np.where(self.dimensionless_length > threshold)
178         with warnings.catch_warnings():
179             # We suppress the warning, because we later replace the inf values
180             # before returning sherwood_number.
181             warnings.filterwarnings(
182                 "ignore", message="divide by zero encountered in power")
183             sherwood_number[smaller_than_indices] = (
184                 1.077 * np.power(
185                     self.dimensionless_length[smaller_than_indices], -1/3) - 0.7)
186         sherwood_number[larger_than_indices] = (
187             3.657 + 6.874
188             * np.power(1000 * self.dimensionless_length[larger_than_indices], -0.488)
189             * np.exp(-57.2 * self.dimensionless_length[larger_than_indices])
190         )
191         # remove inf values (hacky)
192         sherwood_number[sherwood_number > 1E308] = 3.657
193         return sherwood_number
194
195     def get_reynolds_number(self):
196         """
197         Calculate Reynolds number.
198
199         Returns
200         -------
201         numpy array
202             Reynolds number
203         """
204         return abs(self.fluid_density * self.velocity * self.airway_diameter
205                    / self.fluid_viscosity)
206
207     def get_schmidt_number(self):
208         """
209         Calculate particle Schmidt number.
210
211         Returns
212         -------
213         float
214             Schmidt number
215         """
216         return self.fluid_viscosity / \
217             (self.fluid_density * self.brownian_diffusion_coefficient)
218
219     def get_stokes_number(self):
220         """
221         Calculate Stokes' number.
222
223         Returns
```

```
224         -------
225         numpy array
226             Stokes' number
227         """
228         return abs(self.particle_relaxation_time * self.velocity
229                    / self.airway_diameter)
230 
231     def get_impact_velocity(self):
232         """
233         Calculate the impact velocity.
234 
235         Returns
236         -------
237         numpy array
238             impact velocity
239         """
240         relative_distance = ((self.x - self.cumulative_length)
241                              / self.generation_length)
242         truth_array = relative_distance >= 0.8
243         impact_velocity = (truth_array * (
244             self.particle_relaxation_time * self.velocity**2 * self.branching_angle
245             / (0.2 * self.generation_length)
246         ))
247         return impact_velocity
248 
249     def sparse_deposition_matrix(self):
250         N = len(self.x)
251         A = np.zeros([3, N])
252         b = np.zeros(N)
253         # interior points
254         dxs = 0.5*(self.grid_spacing[1:] + self.grid_spacing[:-1])
255         dxs = np.concatenate(([0.5*dxs[0]], dxs, [0.5*dxs[-1]]))
256         coefficients = -(self.deposition_velocity *
257                          self.wetted_perimeter * dxs)
258         A[1, :] = coefficients
259         # vector b is empty because we do not have to linearise the source term
260         return (A, b)
```

## C.7. results.py

```
"""
This module can be used to generate results and figures/images for use in the
thesis.
"""
import os

from constants import Constructor
from model import Airway
from solution import create_labels, show_plot_generations

from tikzplotlib import save as tikz_save
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker


# Directories
DIR = os.path.dirname(os.path.realpath(__file__))
BASE_DIR = os.path.abspath(os.path.join(DIR, '..', '..'))
TEX_IMAGE_DIR = os.path.abspath(os.path.join(BASE_DIR, 'Images', 'Graphs'))
OTHER_IMAGE_DIR = os.path.abspath(
    os.path.join(BASE_DIR, 'Images', 'Unused', 'parametrical_tests'))
LITERATURE_RESULTS_DIR = os.path.abspath(
    os.path.join(BASE_DIR, 'Datasets', 'Literature'))
RESULTS_DIR = os.path.abspath(os.path.join(BASE_DIR, 'Datasets', 'Results'))
# Plot constants to make every plot look consistent
COLOR = 'k'
LINEWIDTH = 1
LINESTYLES = [':', '-.', '--', '-']
TICK_LABEL_STYLE = """ticklabel style={
  /pgf/number format/fixed,
}"""


def _handle_output_file(output_filename, subfigure=False):
    """
    Plot or save a matplotlib figure in .png or .tex format

    Parameters
    ----------
    output_filename: str or None
        Either none for a direct plot, or an output filename to save. Two
        extension are supported: .tex and .png
    subfigure: bool
        Specify if the figure should be a subfigure (only for saving in .tex
        format).

    Returns
    -------
    None
    """
    if output_filename is None:
        plt.show()
    else:
```

```
56          if (ext := os.path.splitext(output_filename)[1]) == '.tex':
57              extra_axis_parameters = [
58                  'clip=false',
59                  'log ticks with fixed point',
60                  TICK_LABEL_STYLE,
61                  'scaled y ticks = false',
62              ]
63              if subfigure:
64                  extra_axis_parameters.append(r'width=\textwidth')
65              output_filename = os.path.join(TEX_IMAGE_DIR, output_filename)
66              tikz_save(
67                  output_filename,
68                  strict=True,
69                  extra_axis_parameters=extra_axis_parameters,
70              )
71          elif ext == '.png':
72              output_filename = os.path.join(OTHER_IMAGE_DIR, output_filename)
73              plt.savefig(output_filename)
74
75
76  def _handle_time_indices(time_indices):
77      """
78      Calculate time indices if not specified.
79
80      Parameters
81      ----------
82      time_indices: None or list
83          Indices of time steps
84      Returns
85      -------
86      list
87          list of time indices
88      """
89      if time_indices is None:
90          max_index = len(T)
91          time_indices = [0, max_index//3, 2*max_index//3, -1]
92      return time_indices
93
94
95  def get_deposition_fractions(output_filename, start=-2, stop=1, number=10,
96                               extra_constructor_properties={}):
97      """Calculate the deposition fractions for a range of particle diameters.
98
99      Particle diameters vary in logspace, and results are saved to a file.
100
101     Parameters
102     ----------
103     output_filename: str
104         output filename
105     start: int
106         start diameter (integer, where 0 is 1e-6, 1 is 1e-5 etc)
107     stop: int
108         stop diameter (like start)
109     number: int
110         number of diameters to calculate
111     extra_constructor_properties: dict
```

```
112         Extra parameters for `Constructor`
113     """
114     # Calculating deposition fractions
115     diameters = np.logspace(start, stop, number) * 1e-6
116     deposition_values = []
117     for diameter in diameters:
118         numerical_solution = Airway(**Constructor(
119             particle_diameter=diameter,
120             **extra_constructor_properties,
121         ).unpack())
122         deposition_values.append(
123             numerical_solution.deposition_fraction()['deposition_fraction'])
124     data = np.array([diameters, deposition_values])
125     df = pd.DataFrame(data.T, columns=['diameter', 'deposition'])
126     df.to_csv(os.path.join(RESULTS_DIR, output_filename), index=False)
127 
128 
129 def create_deposition_plot(results, literature_filename=None,
130                            output_filename=None, subfigure=False):
131     """
132     Create a deposition plot (particle diameter vs deposition fraction)
133 
134     Parameters
135     ----------
136     results: str or list
137         Filename or filenames from `get_deposition_fractions`
138     literature_filename: str or None
139         Filename of literature results to compare to
140     output_filename: str
141         output filename
142     subfigure: bool
143         Specify if the figure should be a subfigure (only for saving in .tex
144         format).
145 
146     See Also
147     --------
148     `get_deposition_fractions`
149     """
150     if isinstance(results, str):
151         results = [results]
152     fig, ax = plt.subplots()
153     if literature_filename is not None:
154         literature_filename = os.path.join(
155             LITERATURE_RESULTS_DIR, literature_filename)
156         literature = pd.read_csv(literature_filename)
157         ax.plot(
158             literature['diameter'],
159             literature['deposition'],
160             color=COLOR,
161             linewidth=LINEWIDTH,
162             linestyle='--')
163     results = [os.path.join(RESULTS_DIR, filename) for filename in results]
164 
165     for idx, result in enumerate(results):
166         df = pd.read_csv(result)
167         ax.plot(
```

```
            df['diameter']*1e6,
            df['deposition'],
            color=COLOR,
            linestyle=LINESTYLES[idx],
            linewidth=LINEWIDTH)

    ax.set_xscale('log')
    ax.xaxis.set_major_formatter(mticker.ScalarFormatter())
    ax.set_xlabel(r'Particle diameter [\si{\micro\metre}]')
    ax.set_ylabel('Deposition fraction')
    ax.set_xlim((0.01, 10))
    ax.set_ylim((0, 1))
    _handle_output_file(output_filename, subfigure=subfigure)


def create_local_deposition_plot(extra_constructor_properties, output_filename=None,
                                 literature_filename=None, subfigure=False,
                                 ):
    """
    Create a local deposition plot (generation vs deposition fraction)

    Parameters
    ----------
    extra_constructor_properties: dict
        Extra parameters for `Constructor`
    output_filename: str
        output filename
    literature_filename: str or None
        Filename of literature results to compare to
    subfigure: bool
        Specify if the figure should be a subfigure (only for saving in .tex
        format).
    """
    if isinstance(extra_constructor_properties, dict):
        extra_constructor_properties = [extra_constructor_properties]
    local_deposition_fraction = []
    fig, ax = plt.subplots()
    for idx, result in enumerate(extra_constructor_properties):
        numerical_solution = Airway(**Constructor(
            **result
        ).unpack())
        ldf = numerical_solution.deposition_fraction()[
            'local_deposition_fraction']
        local_deposition_fraction.append(ldf)
        ax.plot(range(24), ldf, color=COLOR,
                linewidth=LINEWIDTH, linestyle=LINESTYLES[idx])
    if literature_filename is not None:
        literature_filename = os.path.join(
            LITERATURE_RESULTS_DIR, literature_filename)
        literature = pd.read_csv(literature_filename)
        ax.plot(literature['generation'], literature['deposition'],
                color=COLOR, linewidth=LINEWIDTH, linestyle='--')
    ax.set_xlabel('Generation')
    ax.set_ylabel('Deposition fraction')
    ax.yaxis.set_major_locator(
        mticker.MaxNLocator(
```

```
224             min_n_ticks=3, steps=[
225                 1, 2]))
226     ax.xaxis.set_major_locator(mticker.MaxNLocator(steps=[4]))
227     ax.set_ylim(bottom=0)
228     ax.set_xlim((0, 23))
229     _handle_output_file(output_filename, subfigure=subfigure)
230
231
232 def concentration_vs_time_plot(extra_constructor_properties, output_filename=None,
233                                subfigure=False):
234     """
235     Create a concentration plot (concentration vs time)
236
237     Parameters
238     ----------
239     extra_constructor_properties: dict
240         Extra parameters for `Constructor`
241     output_filename: str
242         output filename
243     subfigure: bool
244         Specify if the figure should be a subfigure (only for saving in .tex
245         format).
246     """
247     if isinstance(extra_constructor_properties, dict):
248         extra_constructor_properties = [extra_constructor_properties]
249     fig, ax = plt.subplots()
250     for idx, result in enumerate(extra_constructor_properties):
251         numerical_solution = Airway(**Constructor(
252             **result
253         ).unpack())
254         t = numerical_solution.sol.index
255         deposition_array = np.array(numerical_solution.total_deposition)
256         ax.plot(t, np.cumsum(deposition_array.sum(axis=1)), color=COLOR,
257                 linewidth=LINEWIDTH, linestyle=LINESTYLES[idx])
258     ax.set_xlabel(r'time [\si{\second}]')
259     ax.set_ylabel('Normalised absorption')
260     ax.set_ylim(bottom=0)
261     ax.set_xlim(left=0)
262     _handle_output_file(output_filename, subfigure=subfigure)
263
264
265 def get_dimensions(number_plots):
266     """
267     Return the dimensions for `matplotlib.pyplot.subplots`
268
269     Parameters
270     ----------
271     number_plots: int
272         Number of subplots
273
274     Returns
275     -------
276     Dimensions of subplots
277     """
278     if number_plots < 2:
279         raise ValueError("must have at least 2 plots")
```

```
    if number_plots > 9:
        raise ValueError("cannot have more than 9 plots")
    switcher = {
        2: (2,),
        3: (2, 2),
        4: (2, 2),
        5: (2, 3),
        6: (2, 3),
        7: (3, 3),
        8: (2, 4),
        9: (3, 3),
    }
    return switcher[number_plots]


def time_evolution_plot(time_steps, solutions_bundle, output_filename=None):
    """
    Subplots of concentration vs time for multiple solutions

    Parameters
    ----------
    time_steps: list
        List of time steps
    solutions_bundle: list of pandas DataFrame
        List of `Solution.sol`
    output_filename: str
        output filename

    Notes
    -----
    There is probably some duplication with `single_time_evolution_plot` and
    `compare_multi_solutions_plot`.
    """
    dimensions = get_dimensions(len(solutions_bundle))
    fig, ax = plt.subplots(*dimensions)
    times = [solutions_bundle[0][0].index[
        i*solutions_bundle[0][0].shape[0] // max(time_steps-1, 0)]
        for i in range(time_steps-1)]
    times.append(solutions_bundle[0][0].index[-1])
    for idx, solutions in enumerate(solutions_bundle):
        index = np.unravel_index(idx, dimensions)
        ax[index].set_xlabel('x')
        ax[index].set_ylabel('concentration')
        for idx2, solution in enumerate(solutions):
            solution.columns = map(float, solution.columns)
            ax[index].plot(
                solution.T[times],
                color='k',
                linestyle=LINESTYLES[idx2],
                linewidth=1)
            ax[index].set_xlim([float(solution.columns[0]),
                                float(solution.columns[-1])])
        create_labels(times, solution, ax[index])
    _handle_output_file(output_filename)


```

```
336 def single_time_evolution_plot(
337         time_steps, solutions_bundle, output_filename=None):
338     fig, ax = plt.subplots()
339     times = [solutions_bundle[0].index[
340         i*solutions_bundle[0].shape[0] // max(time_steps-1, 0)]
341         for i in range(time_steps-1)]
342     times.append(solutions_bundle[0].index[-1])
343     ax.set_xlabel('x')
344     ax.set_ylabel('concentration')
345     for idx2, solution in enumerate(solutions_bundle):
346         solution.columns = map(float, solution.columns)
347         ax.plot(
348             solution.T[times],
349             color='k',
350             linestyle=LINESTYLES[idx2],
351             linewidth=1)
352         ax.set_xlim([float(solution.columns[0]), float(solution.columns[-1])])
353     create_labels(times, solution, ax)
354     _handle_output_file(output_filename)
355
356
357 def compare_multi_solutions_plot(number_plots, *solutions):
358     if not all(solution.shape == solutions[0].shape for solution in solutions):
359         raise ValueError("all the input arrays must have same number of"
360                          " dimensions")
361     dimensions = get_dimensions(number_plots)
362     fig, ax = plt.subplots(*dimensions)
363     times = [solutions[0].index[
364         i*solutions[0].shape[0] // max(number_plots-1, 0)]
365         for i in range(number_plots-1)]
366     times.append(solutions[0].index[-1])
367     for idx, time in enumerate(times):
368         index = np.unravel_index(idx, dimensions)
369         for solution in solutions:
370             ax[index].plot(solution.T[time])
371             ax[index].set_ylim([0, 2])
372             ax[index].set_xlabel('x')
373             ax[index].set_ylabel('concentration')
374             ax[index].set_title('{:.2f}%'.format(100*idx/(number_plots-1)))
375     plt.show()
376
377
378 def create_inlet_velocity_plot(output_filename=None):
379     """
380     Plot inlet velocity vs time
381
382     Parameters
383     ----------
384     output_filename: str
385         output filename
386     """
387     constructor = Constructor()
388     inlet_velocity, T = constructor.inlet_velocity, constructor.T
389     fig = plt.figure()
390     ax = fig.gca()
391     ax.plot(T, inlet_velocity, color=COLOR, linewidth=LINEWIDTH)
```

```
392     ax.set_xlabel(r'Time [\si{\second}]')
393     ax.set_ylabel(r'Inlet velocity [\si{\metre\per\second}]')
394     ax.set_xlim((T[0], T[-1]))
395     ax.set_ylim((-1.1*max(abs(inlet_velocity)), 1.1*max(abs(inlet_velocity))))
396     _handle_output_file(output_filename)
397
398
399 def get_velocities_df(time_indices=None):
400     """
401     Get DataFrame with velocity profile
402
403     Parameters
404     ----------
405     time_indices: None or list
406         Indices of time steps
407
408     Returns
409     -------
410     pandas DataFrame
411         velocity for all x and times at time_indices
412     """
413     constructor = Constructor()
414     velocity_all, T = constructor.velocity_all, constructor.T
415     x = constructor.x
416     time_indices = _handle_time_indices(time_indices)
417     velocities = velocity_all[time_indices, :]
418     df = pd.DataFrame(velocities.T, index=x, columns=T[time_indices])
419     return df
420
421
422 def get_reynolds_df(time_indices=None):
423     """
424     Get DataFrame with Reynolds numbers profile
425
426     Parameters
427     ----------
428     time_indices: None or list
429         Indices of time steps
430
431     Returns
432     -------
433     pandas DataFrame
434         Reynolds numbers for all x and times at time_indices
435     """
436     constructor = Constructor()
437     velocity_all, T = constructor.velocity_all, constructor.T
438     x = constructor.x
439     time_indices = _handle_time_indices(time_indices)
440     reynolds = abs(velocity_all[time_indices, :] * constructor.fluid_density
441                    * constructor.get_generation_number(
442                        constructor.airway_diameter_all[time_indices, :])
443                    / constructor.fluid_viscosity)
444     df = pd.DataFrame(reynolds.T, index=x, columns=T[time_indices])
445     return df
446
447
```

```
448 def get_concentration_df(time_indices=None):
449     """
450     Get DataFrame with concentration profile
451 
452     Parameters
453     ----------
454     time_indices: None or list
455         Indices of time steps
456 
457     Returns
458     -------
459     pandas DataFrame
460         Concentration profile for all x and times at time_indices
461     """
462     numerical_solution = Airway(**Constructor().unpack())
463     time_indices = _handle_time_indices(time_indices)
464     df = numerical_solution.sol.T.iloc[:, time_indices]
465     return df
466 
467 
468 def create_multi_plot(df, y_label=None, time_indices=None,
469                       show_generations=True, output_filename=None,
470                       subfigure=False, show_labels=True, x_index=-1):
471     fig, ax = plt.subplots()
472     ax.plot(df, color=COLOR, linewidth=0.4)
473     ax.set_xlabel(r'Distance from trachea [\si{\metre}]')
474     ax.set_ylabel(y_label)
475     ax.set_xlim([min(df.index), max(df.index)])
476     ax.xaxis.set_major_formatter(mticker.ScalarFormatter())
477     if show_labels:
478         create_labels(df.columns, df.T, ax, x_index)
479     if show_generations:
480         show_plot_generations(ax)
481     _handle_output_file(output_filename, subfigure=subfigure)
482 
483 
484 def create_multiple_deposition_fractions(d):
485     for filename, properties in d.items():
486         print(filename)
487         get_deposition_fractions(
488             filename,
489             number=20,
490             extra_constructor_properties=properties)
491 
492 
493 def stability_plot(output_filename=None):
494     def g(z, dx, dt, alpha, beta, gamma, rho):
495         return (alpha - (2*beta*dt/dx**2) - (dt/dx)*gamma - dt*rho) \
496             + (dt/dx**2)*beta*np.exp(1j*z) \
497             + ((dt/dx)*gamma + (dt/dx**2)*beta)*np.exp(-1j*z)
498 
499     fig, ax = plt.subplots()
500     # unit circle
501     t = np.linspace(0, 2*np.pi, 50)
502     ax.plot(np.cos(t), np.sin(t), color=COLOR, linestyle=':')
503 
```

```
    s = np.linspace(-100, 100, 1000)
    w = g(s, 0.07, 0.01, 0.15, 0.1, 1, 0.1)
    ax.plot(w.real, w.imag, color=COLOR)
    lim = 1.2
    ax.set_xlabel(r'\(\operatorname{Re}\)')
    ax.set_ylabel(r'\(\operatorname{Im}\)')
    ax.set_xlim((-lim, lim))
    ax.set_ylim((-lim, lim))
    _handle_output_file(output_filename)


if __name__ == '__main__':
    generate_results = input('Generate results? (y/n): ')
    if generate_results.lower() != 'y':
        exit()
    create_inlet_velocity_plot()
    get_deposition_fractions('deposition_fraction.csv',
                             start=-2, stop=1, number=20)
    create_deposition_plot('deposition_fraction.csv',
                           literature_filename='Eulerian_deposition_fraction.csv',
                           output_filename="deposition_fraction.tex")
    time_indices = [10, 20, 50, 100]
    create_multi_plot(get_velocities_df(time_indices=time_indices),
                      y_label=r'Velocity [\si{\metre\per\second}]',
                      time_indices=time_indices, x_index=35,
                      output_filename='multi_velocity.tex', subfigure=True)
    create_multi_plot(get_reynolds_df(time_indices=time_indices),
                      y_label=r"Reynolds's number", time_indices=time_indices,
                      x_index=35, output_filename='multi_reynolds.tex',
                      subfigure=True)
    time_indices = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) * 10
    create_multi_plot(get_concentration_df(time_indices=time_indices),
                      y_label=r"Concentration", time_indices=time_indices,
                      show_labels=False,
                      output_filename='multi_concentration.tex')
    create_local_deposition_plot(
        {'particle_diameter': 0.01e-6, 'tidal_volume': 0.000625, 'dt': 0.01},
        literature_filename='local_deposition_0.01e-6.csv',
        output_filename='local_deposition_0.01e-6.tex', subfigure=True)
    create_local_deposition_plot(
        {'particle_diameter': 1e-6, 'tidal_volume': 0.000625, 'dt': 0.01},
        literature_filename='local_deposition_1e-6.csv',
        output_filename='local_deposition_1e-6.tex', subfigure=True)
    # Peclet plot
    solutions_bundle = [
        [pd.read_csv(
            f'../../Datasets/Results/simple_peclet/{T}-{P}-peclet.csv',
            index_col=0)
         for T in ['analytical', 'numerical']] for P in [0.003, 0.03, 0.3, 3]
    ]
    for name, bundle in zip([0.003, 0.03, 0.3, 3], solutions_bundle):
        single_time_evolution_plot(
            5, bundle, f'peclet-{name}.tex')

    # Results generation
    # Effect of geometry rescaling:
```

```
560     d = {
561         'deposition_fraction-default.csv': {},
562         'deposition_fraction-rescale_geometry-False.csv': {
563             'rescale_geometry': False,
564         },
565         'deposition_fraction-rescale_geometry-False-FRC-2400.csv': {
566             'rescale_geometry': False,
567             'functional_residual_capacity': 0.0024,
568         },
569     }
570     create_multiple_deposition_fractions(d)
571     create_deposition_plot(
572         [
573             'deposition_fraction-default.csv',
574             'deposition_fraction-rescale_geometry-False.csv',
575             'deposition_fraction-rescale_geometry-False-FRC-2400.csv',
576         ],
577         output_filename='rescale_geometry.tex',
578         subfigure=True)
579     create_local_deposition_plot(
580         [{'particle_diameter': 4e-6, 'rescale_geometry': False},
581          {'particle_diameter': 4e-6}],
582         output_filename='rescale_geometry_local.tex',
583         subfigure=True)
584 
585     # Effect of different geometries
586     from constants import yeh_schum_length, yeh_schum_radius
587     d = {
588         'deposition_fraction-YS.csv': {
589             'generation_length': yeh_schum_length,
590             'generation_radius': yeh_schum_radius,
591             'rescale_geometry': False,
592         },
593     }
594     create_multiple_deposition_fractions(d)
595     create_deposition_plot(
596         [
597             'deposition_fraction-rescale_geometry-False.csv',
598             'deposition_fraction-YS.csv',
599         ],
600         output_filename='weibel_vs_yehschum.tex')
601 
602     # Effect of time-dependent geometry vs fixed
603     d = {
604         'deposition_fraction-fixed_geometry.csv': {
605             'constant_diameter': True,
606         },
607     }
608     create_multiple_deposition_fractions(d)
609     create_deposition_plot(
610         [
611             'deposition_fraction-default.csv',
612             'deposition_fraction-fixed_geometry.csv',
613         ],
614         output_filename='time_dependent_geometry.tex',
615         subfigure=True)
```

```
616     create_local_deposition_plot(
617         [{'particle_diameter': 0.3e-6},
618          {'particle_diameter': 0.3e-6, 'constant_diameter': True}],
619         output_filename='fixed_geometry_local.tex',
620         subfigure=True)
621 
622     # Effect of tidal volume
623     d = {}
624     values = [0.0005, 0.001, 0.002, 0.003]
625     for value in values:
626         d[f'deposition_fraction-tidal_volume-{value}.csv'] = {
627             'tidal_volume': value,
628             'functional_residual_capacity': 0.003,
629         }
630     create_multiple_deposition_fractions(d)
631     create_deposition_plot(
632         [
633             'deposition_fraction-tidal_volume-0.0005.csv',
634             'deposition_fraction-tidal_volume-0.001.csv',
635             'deposition_fraction-tidal_volume-0.002.csv',
636             'deposition_fraction-tidal_volume-0.003.csv',
637         ],
638         output_filename='tidal_volume.tex',
639         subfigure=True)
640 
641     concentration_vs_time_plot(
642         [{'t_final': 60.0, 'particle_diameter': 0.3e-6},
643          {'t_final': 60.0, 'tidal_volume': 0.003, 'particle_diameter': 0.3e-6}],
644         output_filename='concentration_vs_time.tex',
645         subfigure=True)
646 
647     # Effect of particle density
648     d = {}
649     values = np.linspace(500, 2000, 4)
650     for value in values:
651         d[f'deposition_fraction-particle_density-{value}.csv'] = {
652             'particle_density': value,
653             'functional_residual_capacity': 0.003,
654         }
655     create_multiple_deposition_fractions(d)
656     create_deposition_plot(
657         [
658             'deposition_fraction-particle_density-500.csv',
659             'deposition_fraction-particle_density-1000.csv',
660             'deposition_fraction-particle_density-1500.csv',
661             'deposition_fraction-particle_density-2000.csv',
662         ],
663         output_filename='particle_density.tex')
664 
665     stability_plot("stability.tex")
666 
667     # Effect of breathing rate
668     breathing_rate = [10/60, 12/60, 15/60]
669     # NOTE: you would *expect* the following code to work, but it does not
670     # (see: https://stackoverflow.com/a/34021333/7770654). Therefore, we use
671     # the partial function from functools.
```

```
672     #
673     # breathing_pattern = [
674     #     lambda t: -np.cos(t*br*2*np.pi) for br in breathing_rate
675     # ]
676     from functools import partial
677     breathing_pattern = [
678         partial(lambda t, coef: -np.cos(t*coef*2*np.pi), coef=br)
679         for br in breathing_rate
680     ]
681     d = {}
682     for br, bp in zip(breathing_rate, breathing_pattern):
683         d[f'deposition_fraction-breathing_rate-{br}.csv'] = {
684             'functional_residual_capacity': 0.003,
685             'breathing_rate': br,
686             'breathing_pattern': bp,
687             't_final': 1/br + 0.1,
688         }
689     create_multiple_deposition_fractions(d)
690     create_deposition_plot(
691         [
692             'deposition_fraction-breathing_rate-0.16666666666666666.csv',
693             'deposition_fraction-breathing_rate-0.2.csv',
694             'deposition_fraction-breathing_rate-0.25.csv',
695         ],
696         output_filename='breathing_rate.tex')
```

## C.8. exact.py

```
"""
This module computes the exact solution in the case of 1D advection-diffusion,
with a Dirichlet condition on the left, and Neumann on the right.
Solution taken from https://naldc.nal.usda.gov/download/CAT82780278/PDF
"""


import numpy as np
import solution as da
from scipy.optimize import fsolve


class ExactAirway(da.Solution):
    """
    Represent a 1d pipe in a advection/diffusion situation
    """

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.R = 1 # hardcoded value
        self.D = self.D_interface[0]
        self.initial_condition = np.append(self.left_boundary[1],
                                           self.initial_condition)
        self.check_analytical_conditions()
        self.velocity = self.velocity[0]
        self.sol = self.make_df(self.solve())

    def check_analytical_conditions(self):
        if not all([self.left_boundary[0] == "dirichlet",
                    self.right_boundary[0] == "neumann"]):
            raise ValueError("boundaries must be of type Dirichlet and Neumann"
                             " respectively")
        if not max(self.D_interface) == min(self.D_interface):
            raise ValueError("analytical solution does not support nonconstant"
                             " diffusion coefficient")
        if not max(self.velocity) == min(self.velocity):
            raise ValueError("analytical solution does not support nonconstant"
                             " velocity")
        if "deposition" in self.run_flags:
            raise ValueError("analytical solution does not support deposition")
        return True

    def A3_get_eigenvalues(self):
        """
        Get eigenvalues for the 1D advection-diffusion problem.

        Returns
        -------
        numpy array
            Array of eigenvalues
        """
        initial_guess = np.arange(2.5, 2.5+3.1*self.N, 3.1)
        # These hard-coded values come from a graphical analysis of the function
        # below.
        f = lambda m: m/np.tan(m) + self.velocity*self.L/(2*self.D)
```

```
56          return fsolve(f, initial_guess)
57
58      def A3_exact(self, t):
59          """
60          Get solution of the 1D advection-diffusion equation for given t
61
62          Parameters
63          ----------
64          t: float
65              t coordinate
66
67          Returns
68          -------
69          numpy array
70              concentration at the x array, at a time t
71          """
72          x, R, D, v, L, c_i, c_0, N = [self.x, self.R, self.D, self.velocity,
73                                        self.L, self.initial_condition,
74                                        self.left_boundary[1], self.N]
75          # m is an eigenvalue
76          summand = lambda m: (
77              (2*m * np.sin(m*x/L)
78               * np.exp(v*x/(2*D) - v**2*t/(4*D*R) - m**2*D*t/(L**2*R)))
79              / (m**2 + (v*L/(2*D))**2 + v*L/(2*D))
80          )
81          eigenvalues = self.A3_get_eigenvalues()
82          sum_array = np.array([summand(l) for l in eigenvalues])
83          u = c_i + (c_0 - c_i)*(1-np.sum(sum_array, axis=0))
84          return u
85
86      def solve(self):
87          mat = np.array([self.A3_exact(t) for t in self.T[1:]])
88          mat = np.vstack([self.initial_condition, mat])
89          return mat
90
91
92  def exact_steady_state(x, D, v, L, c_0, c_L):
93      """
94      Get steady state solution of 1D convective-diffusion equation for given `x`
95
96      Parameters
97      ----------
98      x: numpy array
99          x coodinates
100     D: float
101         Coefficient of the diffusive term
102     v: float
103         Velocity of the convective flow
104     L: float
105         Length of the tract
106     c_0: float
107         Dirichlet boundary condition on the left side
108     c_L: float
109         Dirichlet boundary condition on the right side
110
111     Returns
```

```
112     -------
113     numpy array
114         Concentrations at positions `x`
115     """
116     P = v*L/D # Peclet number
117     return c_0 + (c_L-c_0)*(np.exp(P*x/L)-1)/(np.exp(P)-1)
118 
119 
120 if __name__ == "__main__":
121     analytical_solution = ExactAirway()
122     analytical_solution.surface_plot()
123     analytical_solution.time_evolution_plot(6)
```