Cite as: Raza, M. A.: Implementation of Lumped Parameter Network Boundary Conditions for the Patient-Specific CFD Simulations of Coronary Arteries in OpenFOAM. In Proceedings of CFD with OpenSource Software, 2024, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2024

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Implementation of Lumped Parameter Network Boundary Conditions for the Patient-Specific CFD Simulations of Coronary Arteries in OpenFOAM

Developed for foam-extend-4.1
Requires: solids4foam-v2.1

Author:
Muhammad Ahmad Raza
University College Dublin
muhammad.a.raza@ucdconnect.ie

Peer reviewed by:
Dr. Malachy J. O'Rourke
Dr. Saeed Salehi
Mélida Andrea Correa Guaña

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like to learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

February 13, 2025

Learning outcomes

The main requirements of a tutorial in the course is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- A brief description of how to set up and run a standard incompressible, laminar CFD case in OpenFOAM and `solids4foam` environments will be given.
- A general description of the base class `fixedValue` covering its usage will be given.

The theory of it:

- The theory on which `fixedValue` boundary condition, typically used to prescribe 0 Pa pressure at the outlet, is based will be discussed.
- A brief description of how the fixed value boundary conditions are unsuitable for the patient-specific simulations and cause reflection at the outlet due to impedance mismatch in FSI simulations will be given.

How it is implemented:

- An in-depth description of how base class for `fixedValue` boundary condition is currently implemented in OpenFOAM and `solids4foam` will be given.
- A description of the proposed implementation of pressure derived from 0D lumped parameter networks will be given.

How to modify it:

- The boundary condition will be modified to apply 0D lumped parameter network (LPN) boundary conditions based on flow coming out of the outlet and values of the network parameters provided by the user.
- A single-element resistive (R) model, two-element windkessel (RC) model, three-element windkessel (RCR) model, and four-element series as well as parallel windkessel (RCRL) models with distal pressure terms will be implemented to prescribe patient-specific outlet pressure boundary conditions for the larger blood vessels, such as aorta.
- A more sophisticated lumped parameter network (LPN) model for coronary boundary condition with arterial and venous capacitive and resistive elements as well as intramyocardial compliance and time-variant pressure term will be implemented for left and right coronary outlets.

- A guide on how to adjust the required parameters for the 0D lumped parameter models will be given.
- CFD tutorial cases to validate the implemented boundary conditions against the standard solution of ODEs describing the respective LPNs obtained in MATLAB.

Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- How to set up and run standard document tutorials using `pimpleFoam` solver for incompressible, laminar cases in `foam-extend-4.1`.
- How to set up and run standard document fluids tutorials like `cylinderInChannel` using `pimpleFluid` solver for incompressible, laminar cases in `solids4foam-v2.1` toolbox [1].
- How to customize and compile boundary conditions and do top-level application programming in OpenFOAM.
- Basic level of bash scripting.
- Basic level MATLAB programming to solve ODEs and plotting results.
- Basic knowledge of windkessel effect and lumped parameter network (LPN) models such as two, three, and four-element windkessel models used in cardiovascular biomechanics [2, 3].
- Basic knowledge of coronary artery physiology and coronary microcirculation in the context of hemodynamic modeling.
- Basic knowledge of lumped parameter network (LPN) models used to set up outlet pressure boundary conditions for coronary artery simulation [4, 5].
- Basic knowledge of cardiovascular simulations, preferably coronary artery simulation in the packages like SimVascular [6] and CRIMSON [7].

Contents

Nomenclature

Acronyms

CABG	Coronary Artery Bypass Graft
CAD	Coronary Artery Disease
CCTA	Coronary Computed Tomography Angiography
CFD	Computational Fluid Dynamics
CHD	Coronary Heart Disease
CVD	Cardiovascular Disease
DIC	Diagonal Incomplete Cholesky
DILU	Diagonal Incomplete LU
FDM	Finite Difference Method
FFR	Fractional Flow Reserve
FFR-CT	Computed Tomography Angiography-derived Fractional Flow Reserve
FSI	Fluid-Structure Interaction
GSV	Great Saphenous Vein
IHD	Ischemic Heart Disease
ITA	Internal Thoracic Artery
LAD	Left Anterior Descending Artery
LCA	Left Coronary Artery
LPN	Lumped Parameter Network
MRI	Magnetic Resonance Imaging
ODE	Ordinary Differential Equation
PBiCG	Preconditioned Bi-Conjugate Gradient
PCG	Preconditioned Conjugate Gradient
PCI	Percutaneous Coronary Intervention
RCA	Right Coronary Artery
WK	Windkessel Model

English symbols

Δt	Time step size	s
C	Vessel compliance	$m^4 \cdot s^2/kg$
C_e	Electric capacitance	F
CO	Cardiac output	m^3/s
D	Diameter	m
f	Frequancy	Hz
g	Accelration of gravity	9.81 (m/s^2)
I	Electric current	A
L	Blood inertance or inductance	kg/m^4
L_e	Electric inductance	H
P/p	Pressure	Pa
Q	Volumetric flow rate	m^3/s
R	Radius	m

R	Vessel resistance	$\mathrm{kg/(m^4 \cdot s)}$
R_e	Electric resistance	Ω
Re	Reynolds number	
T	Time period	s
t	Time	s
U	Velocity	m/s
V	Electric potential difference	V

Greek symbols

μ	Fluid dynamic viscosity	$\mathrm{kg/(m \cdot s)}$
ν	Fluid kinematic viscosity	$\mathrm{m^2/s}$
ω	Angular velocity	rad/s
ρ	Fluid density	$\mathrm{kg/m^3}$

Subscripts

a	arterial vessels
a-μ/am	micro-arterial vessels
d	distal vessels
e	electric
i	intermediate
im	intramyocardial
n	n^{th} or current time step
p	proximal vessels
v	veinous vessels
v-μ/vm	micro-veinous vessels

Chapter 1

Introduction

1.1 Coronary artery disease

Cardiovascular disease (CVD) ranks as the primary cause of mortality worldwide, with coronary artery disease (CAD) being the most common of all cardiovascular diseases. Coronary artery disease (CAD), also known as coronary heart disease (CHD), ischemic heart disease (IHD) or myocardial ischemia, is the build-up of atherosclerotic plaque in the coronary arteries. Plaque is made up of fats, cholesterol, calcium and other substances in the blood. It can develop over time, locally narrowing the diameter in a coronary artery (CA) branch. This is called a CA stenosis and restricts the flow of oxygen-rich blood to the myocardium. If this happens slowly in time, the vascular network can adapt by creating additional arterioles, a mechanism referred to as angiogenesis. Another possible compensation is widening the pressure-regulating arterioles,a phenomenon known as arteriogenesis, which results in increased blood velocity in the stenosis upstream. However, this results in additional load to the plaque which may burst, triggering a blood clot that can cause a myocardial infarction, commonly known as heart attack which is characterized by a sudden blockage of oxygen-rich blood supply to the myocardium downstream a CA branch. Since this condition arises from the narrowing or complete blockage of coronary arteries due to the accumulation of plaque—often comprised of fatty substances—leading to atherosclerosis. The resultant restriction of blood flow to the heart muscle causes ischemia [8, 9].

The current "gold standard" for the diagnosis of the coronary artery disease is an invasive procedure known as fractional flow reserve (FFR) measurement. It is obtained through a standard diagnostic catheter at the time of a coronary angiogram or cardiac catheterization. FFR is defined as the ratio of maximum myocardial blood flow in the presence of stenosis and theoretical maximum myocardial blood flow in the absence of stenosis. However, it is calculated as a ratio of pressure measured distal to the blockage and pressure proximal to the blockage. The FFR value correlates with the degree of blockage in the artery. The normal ratio is expected to be 1 indicating no stenosis. For instance, an FFR value of 0.80 means that the maximum blood flow in the coronary artery being measured is 80% of what it would be if the artery were completely normal. It is useful in the assessment of intermediate blockages to determine the suitable treatment plan. Usually, FFR less than 0.75 indicates ischemia-producing stenosis raising the need for revascularization or surgical intervention. FFR greater than 0.80 indicates non–ischemia-producing stenosis, for which medical therapy adopted as a treatment. FFR value between 0.75 and 0.80 is considered a gray zone where choice is made between revascularization and medical therapy [10, 11, 12].

In cases of severe coronary artery disease, coronary artery bypass graft surgery (CABG) is a common intervention. This surgical procedure involves rerouting blood around arterial blockages using a graft, typically sourced from another part of the patient's body. For instance, the left anterior descending artery (LAD), also known as anterior descending branch, is usually revascularized by sourcing a graft from the internal thoracic artery (ITA), also known as the internal mammary artery, is an artery that supplies the anterior chest wall and the breasts. Similarly, the great saphenous vein (GSV), also known as long saphenous vein, is a large, subcutaneous, superficial vein of the leg. It is

the longest vein in the body, which is usually used for autotransplantation in CABG, when arterial grafts are not available or many grafts are required, such as in a triple bypass or quadruple bypass. Alternatively, a non-surgical option known as percutaneous coronary intervention (PCI) is available. This technique involves the insertion of a balloon catheter into the affected coronary artery, which is then inflated to reopen the vessel [13, 14].

In recent years, computed tomography angiography-derived fractional flow reserve (FFR-CT) has been proposed to replace the invasive FFR measurement. FFR-CT combines coronary CT angiography (CCTA) with computational fluid dynamics (CFD) to estimate the fractional flow reserve (FFR), hence eliminating the need for catheterization and pharmacologic stress by using advanced algorithms to simulate blood flow from CCTA images. This allows for a functional assessment of coronary lesions, helping clinicians determine whether a stenosis significantly impairs blood flow and whether intervention is needed. FFR-CT has been validated in clinical trials and is increasingly used for non-invasive, patient-friendly CAD diagnosis while reducing unnecessary invasive procedures [15, 16, 17].

Apart from enabling non-invasive assessment of coronary stenosis severity by computing fractional flow reserve (FFR-CT), numerical simulations can play a vital role in improving coronary interventions including coronary artery bypass grafting (CABG), stenting, and percutaneous coronary interventions (PCI) by providing detailed insights into blood flow dynamics. Using patient-specific imaging data from computed tomography (CT) or magnetic resonance imaging (MRI) scans, CFD models can simulate hemodynamic parameters like blood velocity, pressure distribution, and wall shear stress, which are critical for assessing the effectiveness of interventions. In CABG, CFD can help optimize graft design by evaluating different configurations, such as sequential and individual grafts, to ensure optimal flow distribution and long-term graft patency. In stenting procedures, CFD can assist in predicting flow disturbances, turbulence, and the risk of restenosis, helping in the development of more effective stent designs that minimize complications like thrombosis. These simulations can allow clinicians to personalize treatment strategies, improving surgical planning and post-operative outcomes. By integrating numerical simulations with medical imaging, researchers and clinicians can enhance patient-specific decision-making, ultimately leading to more effective and durable coronary interventions [18, 19, 20, 21].

All in all, a comprehensive understanding of blood flow dynamics is crucial not only for studying the progression of coronary artery disease but also for diagnosing the severity of stenosis non-invasively, determining the most suitable treatment strategies, predicting the outcomes of proposed intervention a priori, and optimizing the interventions for individual patients.

1.2 Numerical simulations of coronary arteries

Computational simulations have demonstrated significant value in analyzing blood flow within the cardiovascular system [22]. These tools are instrumental in examining the hemodynamics of both healthy and diseased vessels [23, 24], supporting the development and evaluation of vascular medical devices [25, 26], aiding in surgical planning, and forecasting the outcomes of interventions [27, 28]. With ongoing advancements in computational power and numerical methods for blood flow modeling, additional applications are anticipated.

Despite their utility, numerical simulations have been infrequently applied to predict the pulsatile flow and pressure fields within three-dimensional coronary vascular networks. This is largely due to the intricate relationship between coronary vascular dynamics and the interplay between the heart and the arterial system. Unlike other arterial systems, coronary blood flow diminishes during ventricular contraction as intramyocardial pressure rises, exerting a compressive force on coronary vessels [29]. In contrast, coronary flow increases during ventricular relaxation, when intramyocardial pressure decreases, reducing the extravascular compressive force. Consequently, realistic simulation of coronary flow and pressure requires integrated models that account for both the heart and arterial system, as well as their dynamic interactions. Because of these complexities, most three-dimensional computational studies have focused solely on the coronary arteries, often neglecting the interplay between the heart and the arterial system. These studies typically prescribe coronary flow instead

of predicting it and have not incorporated realistic pressure modeling. Additionally, these studies employ traction-free boundary conditions at arterial outlets, regardless of whether the coronary artery walls are considered rigid [25, 24], compliant [30, 31], or subject to cardiac motion [32, 33]. For example, Migliavacca and colleagues [34, 27] analyzed pulsatile coronary flow and pressure in a single coronary artery using a three-dimensional model, but their study relied on an idealized representation and employed low mesh resolution.

1.3 Boundary conditions for the patient-specific CA simulations

Analytic boundary condition (BC) models have often been explicitly coupled, requiring either subiterations within each time step or the use of small time steps dictated by the stability of explicit time integration schemes. To accurately predict physiologically relevant coronary flow rates and pressures in patient-specific arterial networks, computational models must be both robust and stable enough to handle complex flow characteristics while also efficiently integrating different computational scales [35].

In earlier research, Vignon-Clementel *et al.* [36, 37] developed boundary conditions for three-dimensional blood flow models that accommodate subject-specific physiological conditions. For arterial outlet boundaries, excluding coronary vascular beds, simple analytic models such as resistance, impedance, and the three-element windkessel model were used to simulate the arterial system not included in the computational domain. In terms of accuracy and efficiency, this fully implicit approach has proven to be both robust and versatile. In a related study by Kim *et al.* [38], a lumped-parameter heart model was implemented as an inflow boundary condition for a three-dimensional finite-element model of the aorta, enabling the interaction between the ventricle and the arterial system to be considered. This fully implicit coupling produced physiologically realistic flow and pressure fields in a subject-specific aortic model. Additionally, numerical instabilities at boundaries caused by reverse and complex flow structures were addressed by applying an augmented Lagrangian method to enforce constraints on the shape of the velocity profile at the aortic inlet and selected outlets [35].

Kim *et al.* [5] developed a comprehensive method to predict coronary flow and pressure in three-dimensional epicardial coronary arteries by integrating models of the heart and arterial system while accounting for their interactions. This approach aimed to create computational simulations that are both robust and stable, capable of handling complex flow dynamics and coupling across various scales to provide physiologically accurate predictions of flow rates and pressures within patient-specific coronary arterial trees. The method involved coupling a lumped parameter heart model, representing the left and right sides of the heart, to a three-dimensional finite element model of the aorta, enabling simulation of heart-arterial system interactions. For each coronary outlet, a lumped parameter coronary vascular bed model, derived from a simplified version of the model developed by Mantero *et al.* [4], was employed to represent downstream impedance while eliminating coronary venous microcirculation compliance for numerical efficiency. Intramyocardial pressure, influencing coronary flow, was modeled using left or right ventricular pressure based on coronary artery location, with these pressures derived from lumped parameter heart models within a closed-loop system. Boundary conditions included a fully implicit coupling method for the heart, aortic, and coronary vascular bed models, while the three-element windkessel model was applied at non-coronary boundaries to define impedance. Numerical instabilities caused by reverse or complex flow structures were resolved using an augmented Lagrangian method to constrain the velocity profile at the aortic inlet and select outlet boundaries. Simulations were conducted for rest, light, and moderate exercise conditions, as well as varying degrees of stenosis in the left anterior descending coronary artery, capturing coronary and aortic flow and pressure under these scenarios. Blood flow was modeled using the incompressible Navier-Stokes equations under the assumption of a Newtonian fluid, with vessel walls described by elastodynamic equations assuming small displacements and a fixed fluid mesh. The simulations produced realistic coronary flow and pressure waveforms consistent with literature, demonstrating high diastolic and low systolic flow in the left coronary arteries, and the

opposite trend in the right coronary artery due to ventricular pressure differences. These methods enabled the study of various vascular interventions for cardiovascular disease and the effects of changes in cardiac and arterial properties on coronary flow and pressure.

The methodology introduced by Kim *et al.* [5] was later implemented in the finite element method-based tools such as SimVascular [6], an open source pipeline for cardiovascular simulation, and CRIMSON (CardiovasculaR Integrated Modelling and SimulatiON) [7], an advanced simulation environment for subject-specific hemodynamic analysis. A distinctive aspect of coronary blood flow is its "out-of-phase" relationship between flow and pressure. Unlike systemic circulation, where blood flow peaks during systole, the majority of coronary blood flow occurs during diastole. This phenomenon is attributable to the constriction of coronary vascular beds during systole, which increases vascular resistance. When the heart relaxes during diastole, resistance decreases, allowing for maximal coronary flow. The packages like SimVascular and CRIMSON incorporate specialized boundary conditions that effectively simulate this behavior. The left coronary artery flow pattern is typically marked by reduced flow during systole and increased flow during diastole, whereas the right coronary artery flow pattern often exhibits comparable peaks during both systole and diastole.

1.4 OpenFOAM implementations of LPN boundary conditions

Some attempts have been made previously to implement windkessel model boundary condition for cardiovascular flow simulations in OpenFOAM. For instance, in 2016, Piebalgs *et al.* [39] developed a library of solvers and boundary conditions to solve for physiological flow problems in OpenFOAM 4.0. They edited the `pimpleFoam` solver as `windkesselSolver` so that it can implement three-element windkessel model boundary conditions. In 2021, Schenkel *et al.* [40] implemented a haemodynamics simulation framework `haemoFoam` [41] including particle migration model and advanced haemorheology models, as well as three-element windkessel model boundary conditions in OpenFOAM v2206. The implementation of three-element windkessel model boundary condition in `haemoFoam` was based on the earlier implementation by Piebalgs *et al.* [39]. In 2020, Manchester *et al.* [42] implemented three-element windkessel boundary condition on up to 10 outlet boundaries in OpenFOAM v4 [43] for the patient-specific simulation of aorta with aortic valve stenosis. Later on in 2022, they updated their implementation for OpenFOAM v8 [44]. Three-element windkessel model is most commonly used LPN model as an outlet boundary condition for the simulation of larger blood vessels such as aorta. However, no open-source OpenFOAM implementation of more complex windkessel models such as four-element windkessel model which accounts for the inertia of blood apart from the vessel compliance and resistance is available. Similarly, no open-source OpenFOAM implementation of more complex and sophisticated LPN model for downstream coronary vascular beds to evaluate the coronary outlet pressure boundary condition is available.

The aim of this project is to implement and test windkessel and coronary lumped parameter network (LPN) boundary conditions for the patient-specific computational fluid dynamics (CFD) simulations of coronary arteries in OpenFOAM. The project is structured into the following main steps:

1. Developement of differential equations describing the relationship between pressure and flow rate for different lumped parameter network (LPN) models including single-element resistive (R) model, two-element windkessel (RC) model, three-element windkessel (RCR) model, and four-element series as well as parallel windkessel (RCRL) models, and coronary outlet LPN model.

2. Implementation of lumped parameter network (LPN) models including single-element resistive (R) model, two-element windkessel (RC) model, three-element windkessel (RCR) model, and four-element series as well as parallel windkessel (RCRL) models for the evaluation of the outlet pressure boundary condition in larger blood vessels such as aorta in OpenFOAM.

3. Implementation of lumped parameter network (LPN) model for downstream coronary vascular beds for the evaluation of the coronary outlet pressure boundary condition in OpenFOAM.

4. Validation of the implementation against the solution obtained by the numerical treatment of LPNs in MATLAB with the given flow rate and network parameters' values for a pulsatile flow in a straight tube.

The code would be under process and updated. The reader can visit the author's GitHub profile (`https://github.com/MA-Raza`) to make comments or get the latest updates on the ongoing development and implementation of a framework for the patient-specific CFD and FSI simulations of coronary arteries in OpenFOAM and `solids4foam` toolbox.

Chapter 2

LPN Boundary Conditions

2.1 Windkessel effect

The windkessel effect is a crucial physiological mechanism that describes how large elastic arteries moderate pulsatile blood flow generated by the heart to maintain a continuous and steady flow through the circulatory system. The term "windkessel," derived from the German word for "air chamber," historically referred to the air reservoirs in fire engines, which smoothed out intermittent water flow. Analogously, in the human cardiovascular system, the aorta and other large elastic arteries act as a "hydraulic accumulator," damping pressure fluctuations and ensuring consistent blood supply during both systole and diastole. Large arteries, such as the aorta, common carotid, and pulmonary arteries, contain elastin fibers in their walls, which enable them to stretch during systole as blood is ejected from the heart. This stretching stores energy as the arteries accommodate the incoming stroke volume. During diastole, when the heart is at rest, these arteries recoil, releasing the stored energy and maintaining blood flow despite the lack of cardiac ejection. The compliance—or distensibility—of these vessels is analogous to the capacitance in hydraulic systems, highlighting their role as elastic reservoirs [3, 2, 45].

The windkessel effect arises from the interplay between the compliance of large arteries and the resistance of smaller arteries and arterioles. This mechanism reduces pulse pressure by absorbing the systolic surge and maintaining diastolic flow, ensuring stable organ perfusion throughout the cardiac cycle. The arterial compliance effectively buffers the pulsatile output of the heart, converting it into a smoother flow downstream. Since the rate of blood entering large arteries exceeds the rate at which it exits through peripheral resistance, these vessels temporarily store blood during systole and discharge it during diastole [46].

Historically, the concept of the windkessel effect can be traced back to Stephen Hales in the 18th century, who observed the variability of arterial pressure and suggested a connection to arterial elasticity. Otto Frank, a German physiologist, later provided the mathematical foundation for this concept, solidifying its importance in understanding cardiovascular dynamics. The analogy to the air chamber in fire engines, where compliance and resistance combine to produce a constant flow, elegantly captures the function of large arteries as dynamic reservoirs [3].

Understanding the windkessel effect is foundational for comprehending the physiological regulation of blood pressure and flow, as well as the pathological changes that occur with aging or disease. For example, conditions such as arteriosclerosis or hypertension reduce arterial compliance, leading to increased systolic pressure and diminished diastolic flow, which can impair perfusion and increase the workload on the heart. This understanding also informs the development of computational models and diagnostic tools to evaluate arterial function and cardiovascular health [47].

2.2 Elements of a windkessel model

The windkessel model, developed in the late 19th century by German physiologist Otto Frank, represents the heart and systemic arterial system as a closed hydraulic circuit. In this analogy, the circuit consists of a water pump connected to a chamber filled with water and containing a pocket of air. As the pump operates, the water compresses the air, which subsequently forces the water out of the chamber—a mechanism resembling the function of the heart. The windkessel models are frequently employed to simulate the load experienced by the heart during the cardiac cycle, correlating blood pressure and flow in the aorta while accounting for arterial compliance, peripheral resistance, and the inertia of blood flow. A windkessel model can be compared to Poiseuille's law in hydraulics, which describes fluid flow through pipes. Similarly, the windkessel model characterizes blood flow in arteries as analogous to fluid dynamics in a hydraulic system. An explanation for the key parameters of a windkessel model is given below.

2.2.1 Vessel resistance

The resistance describes the resistance encountered by blood as it flows through the network of blood vessel. For electrical resistance R_e, we know

$$V = IR_e \tag{2.1}$$

where I is the current through and V is the potential difference across the resistor R_e. Whereas, in the hydraulic or physiological circuit for the resistor R, we have

$$P(t) = Q(t)R \tag{2.2}$$

where Q is the volume flow rate and P is the pressure drop across the vessel network. Hence the resistor R has the dimensions of $[ML^{-4}T^{-1}]$.

2.2.2 Vessel compliance

The compliance reflects the elasticity and ability of blood vessels to expand and contract during the cardiac cycle. For electrical capacitance C_e, we know

$$I = C_e \frac{\mathrm{d}V}{\mathrm{d}t} \tag{2.3}$$

where I is the current through and V is the potential difference across the capacitor C_e. Whereas, in the hydraulic or physiological circuit for the capacitance or compliance C, we have

$$Q(t) = C \frac{\mathrm{d}P(t)}{\mathrm{d}t} \tag{2.4}$$

where Q is the volume flow rate and P is the pressure. Hence the capacitance or compliance C has the dimensions of $[M^{-1}L^4T^2]$.

2.2.3 Blood inertia

The inertia of blood accounts for the momentum of blood as it moves through the heart and circulatory system. For electrical inductance L_e, we know

$$V = L_e \frac{\mathrm{d}I}{\mathrm{d}t} \tag{2.5}$$

where I is the current through and V is the potential difference across the inductor L_e. Whereas, in the hydraulic or physiological circuit for the inductance or blood inertance L, we have

$$P(t) = L \frac{\mathrm{d}Q(t)}{\mathrm{d}t} \tag{2.6}$$

where Q is the volume flow rate and P is the pressure. Hence the inductance or blood inertance L has the dimensions of $[ML^{-4}]$.

2.3 Discretization schemes

The lumped parameter network (LPN) models such as a windkessel model relates the pressure at an outlet boundary of the 3D domain i.e., pressure at an inlet of the 0D model, with the flow rate coming out of the 3D domain i.e., flow rate going into the 0D model, through an ordinary differential equation (ODE). The time derivative terms for pressure and flow rate in these ODEs can be evaluated numerically using finite difference method (FDM). The finite difference method (FDM) is a numerical technique used to approximate derivatives of a function by replacing them with finite difference approximations. This method is particularly useful in solving differential equations where exact solutions may not be possible or practical. FDM involves discretizing the domain of the function into a grid or mesh and approximating derivatives using the values of the function at discrete points.

The derivative of a function $f(x)$ at the point x is defined as the limit of a difference quotient given by

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \tag{2.7}$$

which in other words states that the difference quotient $\frac{f(x+h)-f(x)}{h}$ is an approximation of the derivative $f'(x)$, and this approximation gets better as h gets smaller. It is well known that Taylor's theorem with remainder gives the Taylor series expansion as

$$f(x+h) = f(x) + hf'(x) + h^2\frac{f''(\xi)}{2!}, \quad \xi \in (x, x+h). \tag{2.8}$$

Rearranging it gives

$$\frac{f(x+h) - f(x)}{h} - f'(x) = h\frac{f''(\xi)}{2}, \tag{2.9}$$

stating that the error is proportional to h, so $\frac{f(x+h)-f(x)}{h}$ is said to be a first-order approximation.

If $h > 0$, say $h = \Delta x$ where Δx is a finite (as opposed to infinitesimal) positive number, then

$$\frac{f(x+h) - f(x)}{h} = \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{2.10}$$

is called the first-order $\mathcal{O}(\Delta x)$ forward difference approximation of $f'(x)$.

If $h < 0$, say $h = -\Delta x$, then

$$\frac{f(x+h) - f(x)}{h} = \frac{f(x) - f(x-\Delta x)}{\Delta x} \tag{2.11}$$

is called the first-order $\mathcal{O}(\Delta x)$ backward difference approximation of $f'(x)$.

By combining different Taylor series expansions, we can obtain approximations of $f'(x)$ of various orders. For instance, by subtracting the Taylor series expansions

$$f(x+\Delta x) = f(x) + \Delta x f'(x) + \Delta x^2\frac{f''(x)}{2!} + \Delta x^3\frac{f'''(\xi_1)}{3!}, \quad \xi_1 \in (x, x+\Delta x), \tag{2.12}$$

$$f(x-\Delta x) = f(x) - \Delta x f'(x) + \Delta x^2\frac{f''(x)}{2!} - \Delta x^3\frac{f'''(\xi_2)}{3!}, \quad \xi_2 \in (x-\Delta x, x), \tag{2.13}$$

we obtain

$$f(x+\Delta x) - f(x-\Delta x) = 2\Delta x f'(x) + \Delta x^3\frac{(f'''(\xi_1) + f'''(\xi_2))}{6}, \tag{2.14}$$

so that

$$\frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} - f'(x) = \Delta x^2\frac{(f'''(\xi_1) + f'''(\xi_2))}{12} \tag{2.15}$$

indicating that $\frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x}$ is an approximation of $f'(x)$ whose error is proportional to Δx^2. It is called the second-order or $\mathcal{O}(\Delta x^2)$ centered difference approximation of $f'(x)$.

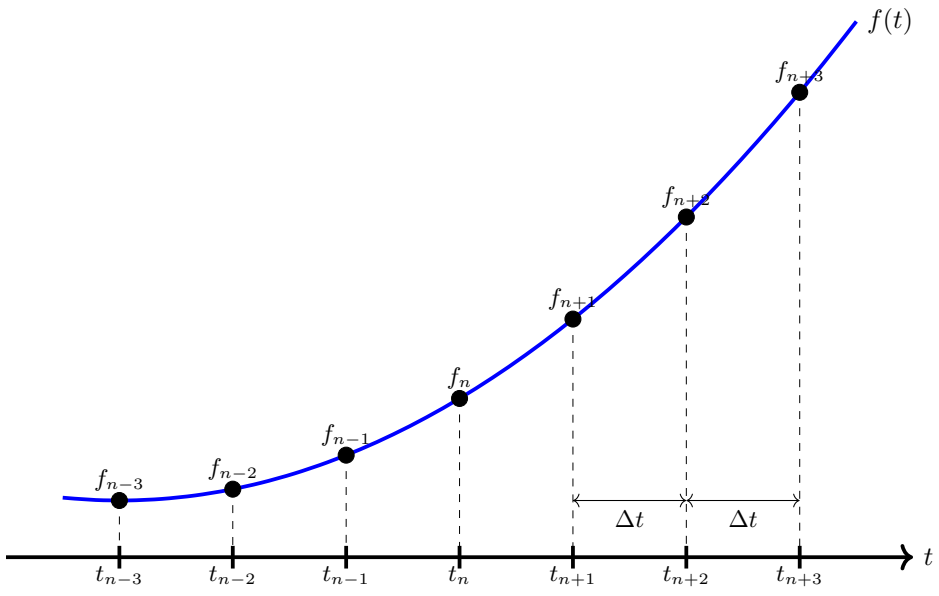


Figure 2.1: Finite difference discretization of a function $f(t)$ with the time-step size of Δt

The ODEs representing different windkessel models and coronary LPN models contains the time derivative terms for flow rate and pressure which needs to discretized in order to numerically evaluate the solution to ODE at any time step. For these parameters only the historical values might be known. Henceforth, backward finite difference approximation can be employed to discretize these terms. In summary, for any function $f(t)$ discretized using FDM with a time-step size of Δt, as shown in Figure 2.1, $\mathcal{O}(\Delta t)$ i.e., first-order backward difference approximation of $\frac{\mathrm{d}f(t)}{\mathrm{d}t}$ at n^{th} time step can be given as

$$\frac{\mathrm{d}f(t)}{\mathrm{d}t}\Big|_n = \frac{f_n - f_{n-1}}{\Delta t} \tag{2.16}$$

and $\mathcal{O}(\Delta t^2)$ i.e., second-order backward difference approximation of $\frac{\mathrm{d}f(t)}{\mathrm{d}t}$ at n^{th} time step can be given as

$$\frac{\mathrm{d}f(t)}{\mathrm{d}t}\Big|_n = \frac{3f_n - 4f_{n-1} + f_{n-2}}{2\Delta t}. \tag{2.17}$$

Whereas, for the second derivative of $f(t)$ i.e., $\frac{\mathrm{d}^2 f(t)}{\mathrm{d}t^2}$, first-order backward difference or $\mathcal{O}(\Delta t)$ approximation can be given as

$$\frac{\mathrm{d}^2 f(t)}{\mathrm{d}t^2}\Big|_n = \frac{f_n - 2f_{n-1} + f_{n-2}}{\Delta t^2} \tag{2.18}$$

and second-order backward difference or $\mathcal{O}(\Delta t^2)$ approximation of $\frac{\mathrm{d}^2 f(t)}{\mathrm{d}t^2}$ can be given as

$$\frac{\mathrm{d}^2 f(t)}{\mathrm{d}t^2}\Big|_n = \frac{2f_n - 5f_{n-1} + 4f_{n-2} - f_{n-3}}{\Delta t^2}. \tag{2.19}$$

2.4 Windkessel models

The original windkessel model proposed by Otto Frank was a two-element model considering only of the compliance of larger blood vessel and total distal or peripheral resistance. However, this model can be improved further by considering the proximal resistance and the inertia of blood. A brief mathematical overview of the LPN models from the simplest single-element resistive (R) model

to two-element windkessel (RC) model, three-element windkessel (RCR) model, and more complex four-element windkessel (RCRL) models is given below. The ordinary differential equations (ODEs) representing each of these LPN model are discretized using the scheme explained so that they can be implemented in OpenFOAM later.

2.4.1 Resistive boundary condition

For a simple single-element lumped parameter network containing only a distal resistance R_d and a distal pressure term P_d, as shown in Figure 2.2, the relationship between flow rate and pressure is given by

$$Q(t) = \frac{P(t) - P_d}{R_d} \tag{2.20}$$

where $Q(t)$ is the flow rate coming out of the 3D domain through the outlet and entering the 0D model. Hence, the pressure at the 3D domain's outlet $P(t)$ can be given by the equation

$$P(t) = R_d Q(t) + P_d \tag{2.21}$$

which can be solved to apply the pressure at the 3D domain's outlet at any n^{th} time step as

$$P_n = R_d Q_n + P_d \tag{2.22}$$

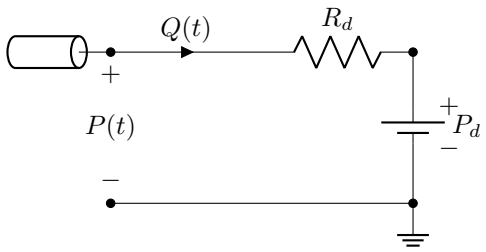


Figure 2.2: Resistive element model with distal pressure term

2.4.2 Two-element windkessel model

A two-element windkessel model contains a distal resistance R_d, a capacitance C, and a distal pressure term P_d, as shown in Figure 2.3. It is the simplest of the windkessel models demonstrating the hemodynamic state. During a cardiac cycle, it takes into account the effect of arterial compliance and total peripheral resistance of distal micro blood vessels. In the electrical analog, the arterial compliance is represented as a capacitor with electric charge storage properties; peripheral resistance of the systemic arterial system is represented as an energy dissipating resistor. The flow of blood from the heart is analogous to that of current flowing in the circuit and the blood pressure in the aorta is modeled as a time-varying electric potential. Hence, the relationship between flow rate and pressure is given by

$$Q(t) = C\frac{\mathrm{d}P(t)}{\mathrm{d}t} + \frac{P(t) - P_d}{R_d} \tag{2.23}$$

where $Q(t)$ is the flow rate coming out of the 3D domain through the outlet and entering the 0D model.

Eq. (2.23) can be rearranged in the form

$$P(t) = R_d Q(t) + P_d - R_d C\frac{\mathrm{d}P(t)}{\mathrm{d}t} \tag{2.24}$$

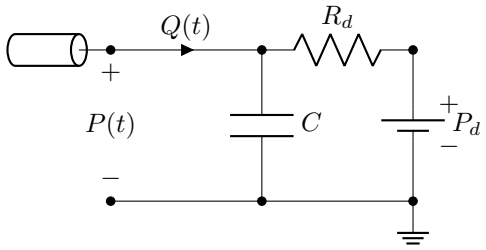


Figure 2.3: Two-element windkessel model with distal pressure term

which can be solved iteratively at each time step using backward finite differencing scheme. Henceforth, using Eq. (2.16) to evaluate the first time derivative term $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ in Eq. (2.24), the first-order finite difference approximation of the pressure at current time step can be given as

$$P_n = \Big(\frac{1}{1+\frac{R_dC}{\Delta t}}\Big)\Big[R_dQ_n + P_d + R_dC\frac{P_{n-1}}{\Delta t}\Big]. \tag{2.25}$$

Similarly, using Eq. (2.17) to evaluate the first time derivative term $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ in Eq. (2.24), the second-order finite difference approximation of the pressure at current time step can be given as

$$P_n = \Big(\frac{1}{1+\frac{3R_dC}{2\Delta t}}\Big)\Big[R_dQ_n + P_d - R_dC\frac{P_{n-2}-4P_{n-1}}{2\Delta t}\Big]. \tag{2.26}$$

2.4.3 Three-element windkessel model

A three-element windkessel model is an improvement in a two-element windkessel model as other than a distal resistance R_d, a capacitance C, and a distal pressure term P_d, it also contains a proximal resistance R_p, as shown in Figure 2.4. This model simulates the characteristic impedance of the proximal aorta. A resistor is added in series to account for this resistance to blood flow due to the aortic valve. The already existing parallel combination of resistor-capacitor represent the total peripheral resistance and aortic compliance in the two-element model as discussed before. Aortic compliance due to pressure variations is achieved by allowing the vessel to undergo volume displacements. The proximal length of the vessel represents the characteristic aortic impedance. Resistance to flow is varied by partial opening and closing of valve. The relationship between flow rate and pressure for the three-element windkessel model is given by

$$(1+\frac{R_p}{R_d})Q(t) + R_pC\frac{\mathrm{d}Q(t)}{\mathrm{d}t} = C\frac{\mathrm{d}P(t)}{\mathrm{d}t} + \frac{P(t)-P_d}{R_d} \tag{2.27}$$

where $Q(t)$ is the flow rate coming out of the 3D domain through the outlet and entering the 0D model.

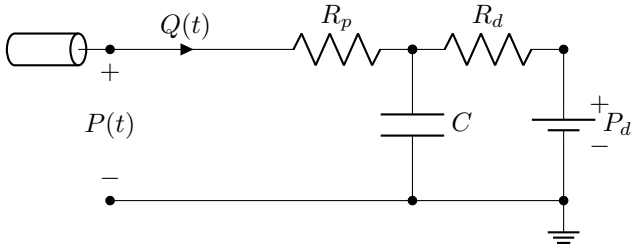


Figure 2.4: Three-element windkessel model with distal pressure term

Eq. (2.27) can be rearranged in the form

$$P(t) = R_p R_d C \frac{\mathrm{d}Q(t)}{\mathrm{d}t} + (R_p + R_d)Q(t) + P_d - R_d C \frac{\mathrm{d}P(t)}{\mathrm{d}t} \tag{2.28}$$

which can be solved iteratively for at each time step using backward finite differencing scheme. Henceforth, using Eq. (2.16) to evaluate the first time derivative terms $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}Q(t)}{\mathrm{d}t}$ in Eq. (2.28), the first-order finite difference approximation of the pressure at current time step can be given as

$$P_n = \Big(\frac{1}{1 + \frac{R_d C}{\Delta t}}\Big)\Big[R_p R_d C \frac{Q_n - Q_{n-1}}{\Delta t} + (R_p + R_d)Q_n + P_d + R_d C \frac{P_{n-1}}{\Delta t}\Big]. \tag{2.29}$$

Similarly, using Eq. (2.17) to evaluate the first time derivative terms $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}Q(t)}{\mathrm{d}t}$ in Eq. (2.28), the second-order finite difference approximation of the pressure at current time step can be given as

$$P_n = \Big(\frac{1}{1 + \frac{3R_d C}{2\Delta t}}\Big)\Big[R_p R_d C \frac{3Q_n - 4Q_{n-1} + Q_{n-2}}{2\Delta t} + (R_p + R_d)Q_n + P_d - R_d C \frac{P_{n-2} - 4P_{n-1}}{2\Delta t}\Big]. \tag{2.30}$$

2.4.4 Four-element windkessel model with inductance in series

The four-element windkessel model includes an inductor in the main branch of the circuit, as shown in Figure 2.5, accounting for the inertia to blood flow in the hydrodynamic model. The pressure drop across the inductor is given as $L\frac{\mathrm{d}Q(t)}{\mathrm{d}t}$. The four-element model gives a more accurate representation of the blood pressure versus cardiac cycle time curve when compared to the two and the three-element windkessel models. For the four-element windkessel model with an inductor L attached in series to the proximal resistance R_p, the relationship between flow rate and pressure is given as

$$(1 + \frac{R_p}{R_d})Q(t) + (\frac{L}{R_d} + R_p C)\frac{\mathrm{d}Q(t)}{\mathrm{d}t} + CL\frac{\mathrm{d}^2Q(t)}{\mathrm{d}t^2} = C\frac{\mathrm{d}P(t)}{\mathrm{d}t} + \frac{P(t) - P_d}{R_d} \tag{2.31}$$

where $Q(t)$ is the flow rate coming out of the 3D domain through the outlet and entering the 0D model.

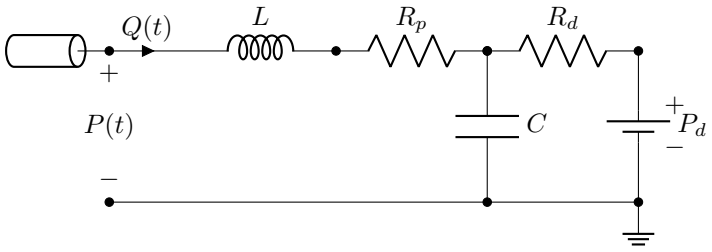


Figure 2.5: Four-element windkessel model with inductance in series and distal pressure term

Eq. (2.31) can be rearranged in the form

$$P(t) = (R_p + R_d)Q(t) + (L + R_p R_d C)\frac{\mathrm{d}Q(t)}{\mathrm{d}t} + R_d CL\frac{\mathrm{d}^2Q(t)}{\mathrm{d}t^2} + P_d - R_d C\frac{\mathrm{d}P(t)}{\mathrm{d}t} \tag{2.32}$$

which can be solved iteratively for at each time step using backward finite differencing scheme. Henceforth, using Eq. (2.16) to evaluate the first time derivative terms $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and Eq. (2.18) to evaluate the second time derivative term $\frac{\mathrm{d}^2Q(t)}{\mathrm{d}t^2}$ in Eq. (2.32), the first-order finite difference

approximation of the pressure at current time step can be given as

$$\begin{aligned} P_n = \Big(\frac{1}{1+\frac{R_dC}{\Delta t}}\Big)\Big[&(R_p+R_d)Q_n \\ &+ (L+R_pR_dC)\frac{Q_n - Q_{n-1}}{\Delta t} \\ &+ R_dCL\frac{Q_n - 2Q_{n-1}+Q_{n-2}}{\Delta t^2} \\ &+ P_d + R_dC\frac{P_{n-1}}{\Delta t}\Big]. \end{aligned} \tag{2.33}$$

Similarly, using Eq. (2.17) to evaluate the first time derivative terms $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and Eq. (2.19) to evaluate the second time derivative term $\frac{\mathrm{d}^2Q(t)}{\mathrm{d}t^2}$ in Eq. (2.32), the second-order finite difference approximation of the pressure at current time step can be given as

$$\begin{aligned} P_n = \Big(\frac{1}{1+\frac{3R_dC}{2\Delta t}}\Big)\Big[&(R_p+R_d)Q_n \\ &+ (L+R_pR_dC)\frac{3Q_n - 4Q_{n-1}+Q_{n-2}}{2\Delta t} \\ &+ R_dCL\frac{2Q_n - 5Q_{n-1}+4Q_{n-2}-Q_{n-3}}{\Delta t^2} \\ &+ P_d - R_dC\frac{P_{n-2}-4P_{n-1}}{2\Delta t}\Big]. \end{aligned} \tag{2.34}$$

2.4.5 Four-element windkessel model with inductance in parallel

For the four-element windkessel model with an inductor L attached in parallel to the proximal resistance R_p, as shown in Figure 2.6, the relationship between flow rate and pressure is given as

$$Q(t) + L(\frac{R_p+R_d}{R_pR_d})\frac{\mathrm{d}Q(t)}{\mathrm{d}t} + CL\frac{\mathrm{d}^2Q(t)}{\mathrm{d}t^2} = \frac{CL}{R_p}\frac{\mathrm{d}^2P(t)}{\mathrm{d}t^2} + (C+\frac{L}{R_pR_d})\frac{\mathrm{d}P(t)}{\mathrm{d}t} + \frac{P(t)-P_d}{R_d} \tag{2.35}$$

where $Q(t)$ is the flow rate coming out of the 3D domain through the outlet and entering the 0D model.

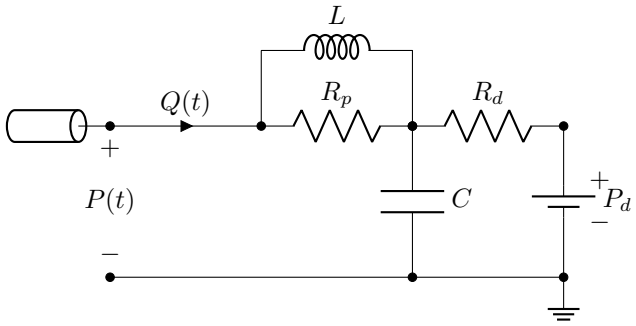


Figure 2.6: Four-element windkessel model with inductance in parallel and distal pressure term

Eq. (2.35) can be rearranged in the form

$$P(t) = R_dQ(t) + L(1+\frac{R_d}{R_p})\frac{\mathrm{d}Q(t)}{\mathrm{d}t} + R_dCL\frac{\mathrm{d}^2Q(t)}{\mathrm{d}t^2} + P_d - \frac{L+R_pR_dC}{R_p}\frac{\mathrm{d}P(t)}{\mathrm{d}t} - \frac{R_dCL}{R_p}\frac{\mathrm{d}^2P(t)}{\mathrm{d}t^2} \tag{2.36}$$

which can be solved iteratively for at each time step using backward finite differencing scheme. Henceforth, using Eq. (2.16) to evaluate the first time derivative terms $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and Eq. (2.18)

to evaluate the second time derivative terms $\frac{\mathrm{d}^2P(t)}{\mathrm{d}t^2}$ and $\frac{\mathrm{d}^2Q(t)}{\mathrm{d}t^2}$ in Eq. (2.36), the first-order finite difference approximation of the pressure at current time step can be given as

$$\begin{aligned} P_n = \Big(\frac{1}{1+\frac{L+R_pR_dC}{R_p\Delta t}+\frac{R_dCL}{R_p\Delta t^2}}\Big)\Big[&R_dQ_n \\ &+L(1+\frac{R_d}{R_p})\frac{Q_n-Q_{n-1}}{\Delta t} \\ &+R_dCL\frac{Q_n-2Q_{n-1}+Q_{n-2}}{\Delta t^2} \\ &+P_d+\frac{L+R_pR_dC}{R_p}\frac{P_{n-1}}{\Delta t} \\ &+\frac{R_dCL}{R_p}\frac{2P_{n-1}-P_{n-2}}{\Delta t^2}\Big]. \end{aligned} \tag{2.37}$$

Similarly, using Eq. (2.17) to evaluate the first time derivative terms $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ and Eq. (2.19) to evaluate the second time derivative terms $\frac{\mathrm{d}^2P(t)}{\mathrm{d}t^2}$ and $\frac{\mathrm{d}^2Q(t)}{\mathrm{d}t^2}$ in Eq. (2.36), the second-order finite difference approximation of the pressure at current time step can be given as

$$\begin{aligned} P_n = \Big(\frac{1}{1+\frac{3(L+R_pR_dC)}{2R_p\Delta t}+\frac{2R_dCL}{R_p\Delta t^2}}\Big)\Big[&R_dQ_n \\ &+L(1+\frac{R_d}{R_p})\frac{3Q_n-4Q_{n-1}+Q_{n-2}}{2\Delta t} \\ &+R_dCL\frac{2Q_n-5Q_{n-1}+4Q_{n-2}-Q_{n-3}}{\Delta t^2} \\ &+P_d-\frac{L+R_pR_dC}{R_p}\frac{P_{n-2}-4P_{n-1}}{2\Delta t} \\ &-\frac{R_dCL}{R_p}\frac{-5P_{n-1}+4P_{n-2}-P_{n-3}}{\Delta t^2}\Big]. \end{aligned} \tag{2.38}$$

2.5 LPN model of downstream coronary vascular beds

The lumped parameter model of coronary vascular beds, as shown in Figure 2.7, is used to represent the impedance of the downstream coronary vascular networks absent in the 3D computational domain. The model has a arterial side, a venous side, and a junction between them modeling the effect of intramyocardial parameters such as compliance and time-varying pressure. The arterial side consists of a coronary arterial resistance R_a which represents the resistance to flow in the coronary arteries, coronary arterial compliance C_a which represents the ability of the coronary arteries to expand and store blood, and coronary arterial microcirculation resistance $R_{a-\mu}$ which represents the resistance to flow in the small vessels of the microcirculation. The venous side consists of a coronary venous microcirculation resistance $R_{v-\mu}$ which represents the resistance to flow in the small veins of the microcirculation, coronary venous compliance C_v which represents the ability of the coronary veins to expand and store blood, and coronary venous resistance R_v which represents the resistance to flow in the coronary veins. Myocardial compliance C_{im} represents the compliance of the myocardium, which affects the flow and pressure in the coronary arteries. Whereas, intramyocardial pressure $P_{im}(t)$ is the pressure exerted by the heart muscle on the coronary vessels, and is time-dependent. It is represented by either the left or right ventricular pressure, depending on the location of the coronary arteries. As a rule of thumb, the left ventricular pressure should be used as $P_{im}(t)$ while modeling the left coronary arteries, whereas the right ventricular pressure should be used as $P_{im}(t)$ while modeling the right coronary arteries. However, the pressure of one ventricle can be used for both left and right coronary branches by multiplying it with an appropriate scaling factor accounting for the fact that right ventricular pressure is typically lower than the left ventricular pressure. For example, if the left ventricular pressure data is available, it can be scaled by a factor of 1.5 and 0.5 for the left and right coronary arteries, respectively [5].

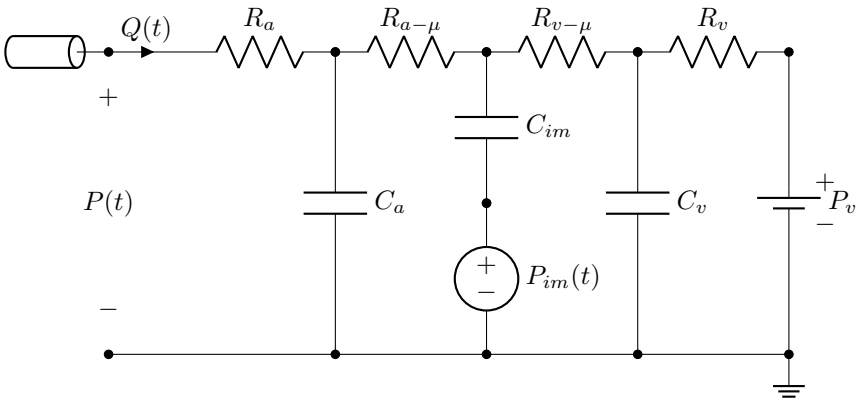


Figure 2.7: Lumped parameter model of downstream coronary vascular beds with intramyocardial pressure term and distal pressure at the venous side

Kim *et al.* [5] used a simplified form of the model developed by Mantero *et al.* [4] by removing the coronary veinous microcirculation compliance C_v from the original model to simplify the numerics. This simplification did not compromise the realistic waveforms of coronary pressure and flow. Hence, the lumped parameter model of coronary vascular beds simplifies the complex downstream coronary circulation into a manageable model, enabling the simulation of coronary flow and pressure while considering the effects of the heart and arterial system. This model can be coupled to the 3D computational domain of the aorta and coronary arteries by replacing the coronary outlet pressure $P(t)$ with an ordinary differential equation derived from the lumped parameter coronary vascular bed model. Considering the given model without venous compliance, the pressure $P(t)$ can be related with the flow rate through a system of linear ordinary differential equations (ODEs) given as

$$Q(t) - C_a \frac{\mathrm{d}P(t)}{\mathrm{d}t} = \frac{P_i(t) - P_v}{R_{v-\mu} + R_v} + C_{im} \frac{\mathrm{d}P_i(t)}{\mathrm{d}t} - C_{im} \frac{\mathrm{d}P_{im}(t)}{\mathrm{d}t} \tag{2.39}$$

and

$$(1 + \frac{R_a}{R_{a-\mu}})Q(t) + R_a C_a \frac{\mathrm{d}Q(t)}{\mathrm{d}t} = \frac{P(t) - P_i(t)}{R_{a-\mu}} + C_a \frac{\mathrm{d}P(t)}{\mathrm{d}t} \tag{2.40}$$

where $P_i(t)$ is an intermediate pressure value taking into account the venous side and intramyocardial parameters and then acting as a time-varient distal pressure term on the arterial side contributing to the evaluation of the overall pressure $P(t)$. Eq. (2.39) and Eq. (2.40) can be rearranged in the form

$$\begin{aligned} P_i(t) = {} & (R_{v-\mu} + R_v)Q(t) \\ & - (R_{v-\mu} + R_v)C_a \frac{\mathrm{d}P(t)}{\mathrm{d}t} \\ & + (R_{v-\mu} + R_v)C_{im} \frac{\mathrm{d}P_{im}(t)}{\mathrm{d}t} \\ & + P_v - (R_{v-\mu} + R_v)C_{im} \frac{\mathrm{d}P_i(t)}{\mathrm{d}t} \end{aligned} \tag{2.41}$$

and

$$\begin{aligned} P(t) = {} & (R_{a-\mu} + R_a)Q(t) \\ & + R_{a-\mu}R_a C_a \frac{\mathrm{d}Q(t)}{\mathrm{d}t} \\ & + P_i(t) - R_{a-\mu}C_a \frac{\mathrm{d}P(t)}{\mathrm{d}t} \end{aligned} \tag{2.42}$$

which can be solved iteratively for at each time step using backward finite differencing scheme. Henceforth, using Eq. (2.16) to evaluate the first time derivative terms $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$, $\frac{\mathrm{d}P_i(t)}{\mathrm{d}t}$, $\frac{\mathrm{d}P_{im}(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ in Eq. (2.41) and Eq. (2.42), the first-order finite difference approximation of the pressures at current time step can be given as

$$\begin{aligned} P_{i,n} = \Big(\frac{1}{1+\frac{(R_{v-\mu}+R_v)C_{im}}{\Delta t}}\Big)\Big[&(R_{v-\mu}+R_v)Q_n \\ &- (R_{v-\mu}+R_v)C_a\frac{P_n - P_{n-1}}{\Delta t} \\ &+ (R_{v-\mu}+R_v)C_{im}\frac{P_{im,n}-P_{im,n-1}}{\Delta t} \\ &+ P_v + (R_{v-\mu}+R_v)C_{im}\frac{P_{i,n-1}}{\Delta t}\Big] \end{aligned} \tag{2.43}$$

and

$$\begin{aligned} P_n = \Big(\frac{1}{1+\frac{R_{a-\mu}C_a}{\Delta t}}\Big)\Big[&(R_{a-\mu}+R_a)Q_n \\ &+ R_{a-\mu}R_aC_a\frac{Q_n - Q_{n-1}}{\Delta t} \\ &+ P_{i,n} + R_{a-\mu}C_a\frac{P_{n-1}}{\Delta t}\Big]. \end{aligned} \tag{2.44}$$

Similarly, using Eq. (2.17) to evaluate the first time derivative terms $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$, $\frac{\mathrm{d}P_i(t)}{\mathrm{d}t}$, $\frac{\mathrm{d}P_{im}(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}P(t)}{\mathrm{d}t}$ in Eq. (2.41) and Eq. (2.42), the second-order finite difference approximation of the pressures at current time step can be given as

$$\begin{aligned} P_{i,n} = \Big(\frac{1}{1+\frac{3(R_{v-\mu}+R_v)C_{im}}{2\Delta t}}\Big)\Big[&(R_{v-\mu}+R_v)Q_n \\ &- (R_{v-\mu}+R_v)C_a\frac{3P_n - 4P_{n-1}+P_{n-2}}{2\Delta t} \\ &+ (R_{v-\mu}+R_v)C_{im}\frac{3P_{im,n}-4P_{im,n-1}+P_{im,n-2}}{2\Delta t} \\ &+ P_v - (R_{v-\mu}+R_v)C_{im}\frac{(P_{i,n-2}-4P_{i,n-1})}{\Delta t}\Big] \end{aligned} \tag{2.45}$$

and

$$\begin{aligned} P_n = \Big(\frac{1}{1+\frac{3R_{a-\mu}C_a}{2\Delta t}}\Big)\Big[&(R_{a-\mu}+R_a)Q_n \\ &+ R_{a-\mu}R_aC_a\frac{3Q_n - 4Q_{n-1}+Q_{n-2}}{2\Delta t} \\ &+ P_{i,n} - R_{a-\mu}C_a\frac{(P_{n-2}-4P_{n-1})}{2\Delta t}\Big]. \end{aligned} \tag{2.46}$$

Chapter 3

Implementation of Windkessel BCs

3.1 Solution methodology

The implementation of the windkessel model for an outlet pressure boundary condition involves a methodical approach to model the dynamics of pressure and flow at the boundary of a computational domain, based on historical pressure, flow rates, and model parameters data. The algorithm is designed to work with different types of so-called lumped parameter models, including resistive and various forms of windkessel models with varying complexities. These models are used to simulate the behavior of arteries or other vessels in fluid dynamics, where resistance of the vessels, their compliance, and inductance or inertance of blood influence the flow and pressure relationship. The algorithm operates in a time-stepping manner and involves several stages to initialize, update, and finalize the boundary conditions, as shown in Algorithm 1.

Initially, the boundary condition (BC) is defined as an object and initialized with default parameter values. Parameters are then read from a dictionary, and the algorithm looks up the necessary models and differencing schemes specified in the dictionary. After initializing the fields and setting up the required properties, the algorithm proceeds to perform the main simulation steps.

During each time-step, the algorithm retrieves the time step size and other necessary data, including boundary flux and pressure fields. It then calculates the patch area required for further calculations. If the time index has changed, the algorithm updates the flow and pressure history variables to ensure the system's state is correctly tracked. For time indices greater than one, the algorithm switches to the specified windkessel model (such as `Resistive`, `WK2`, `WK3`, `WK4Series`, and `WK4Parallel`), applying the appropriate differencing scheme (first-order or second-order) to calculate the new pressure values. The mathematical relationships used to calculate the pressure value in each case are based on the ordinary differential equations (ODE) described in the previous chapter. These updates are crucial for maintaining accurate pressure simulations. After updating the pressure, the algorithm finalizes the boundary condition by calling the base class function to update the coefficients for the next iteration.

During the simulation, depending on the specified `writeTime`, the algorithm writes out the updated boundary field properties. Once the `writeTime` is reached, it logs the parameters for the windkessel model, including resistance, compliance, and inductance, as well as the differencing scheme used. Moreover, the flow rate and pressure histories are also written to ensure historical data is preserved which is necessary for a perfect restart. Finally, the algorithm finalizes the output by writing all boundary field values using a specified function.

3.2 Implementation in OpenFOAM

The resistive and windkessel model lumped parameter network boundary conditions are implemented in `foam-extend 4.1` using a header and a source file described below in detail.

Algorithm 1 Windkessel Outlet Pressure Boundary Condition

```
Define the BC as an object and initialize with default parameter values
Read the parameters, lookup for the model and differencing scheme, and initialize from dictionary
for each time-step do
    Retrieve time step size (dt) from database
    Retrieve the boundary flux (phi) and pressure (p) fields from database
    Calculate patch area (area)
    if time index has changed then
        Update flow history variables (Qo, Qoo, Qo, Qn)
        Update pressure history variables (Po, Poo, Po, Pn)
    end if
    if time index > 1 then
        switch (Windkessel model)
          case Resistive:
            Calculate new pressure (Pn) based on resistive model
          case WK2, WK3, WK4Series or WK4Parallel:
            switch (differencing scheme)
              case firstOrder:
                Calculate new pressure (Pn)
              case secondOrder:
                Calculate new pressure (Pn)
              default:
                Handle unsupported differencing scheme
            end switch
          default:
            Handle unsupported Windkessel model
        end switch
    end if
    Apply implicit pressure update (Pn / (rho + SMALL))
    Finalize boundary condition update (call base class updateCoeffs)
end for
for each writeTime do
    Write common boundary field properties
    Write Windkessel model parameters (resistance, compliance, inductance)
    Write differencing scheme (firstOrder or secondOrder)
    Write flow history values (Qooo, Qoo, Qo, Qn)
    Write pressure history values (Pooo, Poo, Po, Pn)
    Finalize writing of boundary field values using writeEntry("value")
end for
```

3.2.1 Header file

The header file `windkesselOutletPressureFvPatchScalarField.H` contains the description of the implemented boundary condition and all the declarations necessary for the implementation of the numerics in the source file.

3.2.1.1 Description

The description added at the start of the header file give a brief overview of five different lumped parameter network (LPN) models implemented in the source file incldluing

- **`Resistive` (single-element, R)**: A simple resistive boundary where the pressure drop is proportional to flow. A distal pressure term (P_d) is added to this model after the distal

resistance (R_d), making it applicable to cases where distal resistance (R_d) is not grounded to zero pressure. The model is based on the Eq. (2.22).

- **`WK2` (2-element windkessel, RC)**: A model with a distal resistance (R_d) and capacitance (C), commonly used to represent the compliance of arteries without considering proximal resistance. The model is described by the ordinary differential equation (ODE) given in Eq. (2.24). The first- and second-order discretized forms of the ODE given by Eq. (2.25) and Eq. (2.26), respectively are used in the implementation to calculate the pressure.
- **`WK3` (3-element windkessel, RCR)**: The classic 3-element model includes proximal resistance (R_p), distal resistance (R_d), and capacitance (C). It is represented by the ODE given in Eq. (2.28). The first- and second-order discretized forms of the ODE given by Eq. (2.29) and Eq. (2.30), respectively are used in the implementation to calculate the pressure.
- **`WK4Series` (4-element windkessel, RCRL, with inductance in series)**: This model extends the 3-element windkessel by adding inductance (L), accounting for the inertial effects of blood in series with the proximal resistance (R_p). The model is described by the ODE given in Eq. (2.32). The first- and second-order discretized forms of the ODE given by Eq. (2.33) and Eq. (2.34), respectively are used in the implementation to calculate the pressure.
- **`WK4Parallel` (4-element windkessel, RCRL, with inductance in parallel)**: This model extends the 3-element windkessel by placing inductance (L) in parallel with the proximal resistance (R_p), capturing additional dynamics related to flow fluctuations and inertial effects. The model is represented by the ODE given in Eq. (2.36). The first- and second-order discretized forms of the ODE given by Eq. (2.37) and Eq. (2.38), respectively are used in the implementation to calculate the pressure.

A distal pressure term (P_d) is added to all the models after the distal resistance (R_d), making them applicable to cases where distal resistance (R_d) is not grounded to zero pressure. This boundary condition calculates the outlet pressure based on volumetric flow rates and the windkessel parameters using either a second-order backward differencing scheme or a simpler first-order backward differencing approach depending upon the scheme specified by the user. It updates the pressure (P_n) at each time step based on historical values of pressure and flow rates, ensuring the boundary accurately models physiological behavior.

The method to specify the boundary condition for any patch in the `0/p` dictionary is shown in the Listing 3.1.

Listing 3.1: Specification of `windkesselOutletPressure` BC in the `0/p` dictionary

```
<patchName>
{
type                windkesselOutletPressure;
windkesselModel     <Resistive, WK2, WK3, WK4Series or WK4Parallel>; // Type of model, Default: WK3
Rp                  <value>; // Proximal resistance in [kgm^-4s^-1], Default: 1
Rd                  <value>; // Distal resistance in [kgm^-4s^-1], Default: 1
C                   <value>; // Capacitance or compliance in [m^4s^2kg^-1], Default: 1
L                   <value>; // Inductance or inertance in [kgm^-4], Default: 1
Pd                  <value>; // Distal pressure in [Pa], Default: 0
rho                 <value>; // Fluid density in [kgm^-3], Default: 1
diffScheme          <firstOrder or secondOrder>; // Differencing scheme, Default: secondOrder
value               uniform 0;
}
```

3.2.1.2 Header guard and includes

The header file begins with the definition of a header guard using the preprocessor directive `#ifndef windkesselOutletPressureFvPatchScalarField_H`, ensuring that the file content is included only once during the compilation process. The guard is established with the corresponding `#define`

`windkesselOutletPressureFvPatchScalarField_H` directive. Following this, the base class is included through the `#include "fixedValueFvPatchFields.H"` directive, allowing the derived class `windkesselOutletPressureFvPatchScalarField` to inherit its functionalities and definitions.

Listing 3.2: Header guard and includes in header file

```
231 #ifndef windkesselOutletPressureFvPatchScalarField_H
232 #define windkesselOutletPressureFvPatchScalarField_H
233
234 #include "fixedValueFvPatchFields.H"
```

3.2.1.3 Namespace declaration

The header file `windkesselOutletPressureFvPatchScalarField.H` containing relevant declarations for the `windkesselOutletPressureFvPatchScalarField` class resides in the `Foam` namespace, which encapsulates all OpenFOAM-specific classes and functions.

3.2.1.4 Enumerations definition

Before the class declaration, the code defines two enumerations. The `WindkesselModel` enumeration is used to specify the various types of windkessel models available for simulating outlet pressure conditions. These models include the simple resistive model (R), the 2-element windkessel model (RC), the 3-element windkessel model (RCR), and two 4-element windkessel models, one with inductance in series (RCRL) and the other with inductance in parallel. Each model is assigned a unique integer value for identification. The `DifferencingScheme` enumeration defines the numerical schemes used for approximating time derivatives in the ordinary differential equations (ODEs) representing the dynamics of the windkessel models. The first-order backward differencing scheme and the second-order backward differencing scheme are included, each of which is used to discretize time derivatives in the model simulations.

Listing 3.3: Enumerations definition in header file

```
241 //- Enumerator for supported windkessel model types
242 enum WindkesselModel
243 {
244     Resistive = 0,      // Simple resistive model (R)
245     WK2 = 1,            // 2-element Windkessel model (RC)
246     WK3 = 2,            // 3-element Windkessel model (RCR)
247     WK4Series = 3,      // 4-element Windkessel with inductance in series (RCRL)
248     WK4Parallel = 4     // 4-element Windkessel with inductance in parallel
249 };
250
251 //- Enumerator for for time differencing schemes
252 enum DifferencingScheme
253 {
254     firstOrder = 0,     // First-order backward differencing scheme
255     secondOrder = 1     // Second-order backward differencing scheme
256 };
```

3.2.1.5 Class declaration

The class `windkesselOutletPressureFvPatchScalarField` is declared as shown in the Listing 3.4, as a subclass of `fixedValueFvPatchScalarField`, which is used to represent a boundary condition for outlet pressure in fluid simulations. This class encapsulates the properties and methods required for the windkessel model, a model used to simulate outlet pressure based on flow dynamics. By inheriting from `fixedValueFvPatchScalarField`, it benefits from predefined functionality for handling scalar fields and boundary conditions in the finite volume method framework used by OpenFOAM. The class declaration establishes the structure of the object, including both the public and private sections, and facilitates interaction with other components of the simulation environment.

Listing 3.4: `windkesselOutletPressureFvPatchScalarField` class definition in header file

```
class windkesselOutletPressureFvPatchScalarField
:
    public fixedValueFvPatchScalarField
```

3.2.1.6 Private data

The private data members of the `windkesselOutletPressureFvPatchScalarField` class are essential for the internal workings of the windkessel model. These include various physical parameters and state variables. Henceforth, in the private section, as shown in the Listing 3.5, the class defines several windkessel model parameters, such as proximal resistance (`Rp_`), distal resistance (`Rd_`), compliance (`C_`), inductance (`L_`), and distal pressure (`Pd_`). Additionally, fluid density (`rho_`) is stored for use in pressure and flow calculations. The state variables keep track of pressures and flow rates at different time steps, including pressures three time steps ago (`Pooo_`), two time steps ago (`Poo_`), the previous time-step pressure (`Po_`), and the current pressure (`Pn_`). Flow rates are similarly tracked through `Qooo_`, `Qoo_`, `Qo_`, and `Qn_`. The model type and time differencing scheme are specified by `windkesselModel_` and `diffScheme_`, respectively, and the time index for historical data updates is stored in `timeIndex_`.

Listing 3.5: Declaration Private data members in header file

```
private:

    // Private data

        //- Windkessel parameters
        scalar Rp_;     // Proximal resistance (Rp)
        scalar Rd_;     // Distal resistance (Rd)
        scalar C_;      // Compliance or capacitance (C)
        scalar L_;      // Inductance (L)
        scalar Pd_;     // Distal pressure (Pd)
        scalar rho_;    // Fluid density (rho)

        //- Windkessel state variables
        scalar Pooo_;   // Pressure three time steps ago
        scalar Poo_;    // Pressure two time steps ago
        scalar Po_;     // Previous time-step pressure
        scalar Pn_;     // Current pressure
        scalar Qooo_;   // Flow rate three time steps ago
        scalar Qoo_;    // Flow rate two time steps ago
        scalar Qo_;     // Previous time-step flow rate
        scalar Qn_;     // Current flow rate

        //- Windkessel LPN configuration
        WindkesselModel windkesselModel_;   // Selected Windkessel model type
        DifferencingScheme diffScheme_;     // Selected time differencing scheme

        //- Time index
        label timeIndex_;   // Tracks the simulation time index
```

3.2.1.7 Runtime type

The public data members of the `windkesselOutletPressureFvPatchScalarField` class allow access to key functions and methods. The class is equipped with runtime type information through the `TypeName("windkesselOutletPressure")` declaration in the public section, as shown in the Listing 3.6, which allows OpenFOAM's runtime system to correctly handle the object type during simulations.

Listing 3.6: Declaration of runtime type in header file

```
public:

```

```
318     //- Runtime type information
319     TypeName("windkesselOutletPressure");
```

3.2.1.8 Constructors

The `windkesselOutletPressureFvPatchScalarField` class provides several constructors to support different initialization methods, which enable flexibility in how the boundary condition object is created and managed. These constructors allow the class to be instantiated from a patch and an internal field, from a patch, an internal field, and a dictionary, or by copying an existing object.

The first constructor, as shown in the Listing 3.7, accepts a patch and an internal field as input, allowing for the basic setup of the boundary condition.

Listing 3.7: Declaration of a default constructor in header file

```
321     // Constructors
322
323         //- Construct from patch and internal field
324         windkesselOutletPressureFvPatchScalarField
325         (
326             const fvPatch&,
327             const DimensionedField<scalar, volMesh>&
328         );
```

The second constructor, as shown in the Listing 3.8, provides additional functionality by enabling the initialization of the class with a dictionary, which can contain additional parameters for configuring the boundary condition.

Listing 3.8: Declaration of a dictionary constructor in header file

```
330         //- Construct from patch, internal field, and dictionary
331         windkesselOutletPressureFvPatchScalarField
332         (
333             const fvPatch&,
334             const DimensionedField<scalar, volMesh>&,
335             const dictionary&
336         );
```

Another constructor, as shown in the Listing 3.9, supports the mapping of an existing object onto a new patch, which is useful when the simulation mesh is modified.

Listing 3.9: Declaration of a mapping constructor in header file

```
338         //- Map existing object onto a new patch
339         windkesselOutletPressureFvPatchScalarField
340         (
341             const windkesselOutletPressureFvPatchScalarField&,
342             const fvPatch&,
343             const DimensionedField<scalar, volMesh>&,
344             const fvPatchFieldMapper&
345         );
```

Additionally, a copy constructor, as shown in the Listing 3.10, is provided which allows the creation of a new object based on an existing instance. The class also includes a method to construct and return a clone, ensuring that an identical copy of the object can be created when needed.

Listing 3.10: Declaration of a copy constructor and a method to construct and return a clone in header file

```
347         //- Copy constructor
348         windkesselOutletPressureFvPatchScalarField
349         (
350             const windkesselOutletPressureFvPatchScalarField&
351         );
352
353         //- Construct and return a clone
```

```
354         virtual tmp<fvPatchScalarField> clone() const
355         {
356             return tmp<fvPatchScalarField>
357             (
358                 new windkesselOutletPressureFvPatchScalarField(*this)
359             );
360         }
```

Furthermore, another constructor, as shown in the Listing 3.11, facilitates the creation of an object by setting a reference to an internal field, ensuring proper data linkage.

Listing 3.11: Declaration of a constructor setting internal field reference in header file

```
362         //- Construct as copy setting internal field reference
363         windkesselOutletPressureFvPatchScalarField
364         (
365             const windkesselOutletPressureFvPatchScalarField&,
366             const DimensionedField<scalar, volMesh>&
367         );
368
369         //- Construct and return a clone setting internal field reference
370         virtual tmp<fvPatchScalarField> clone
371         (
372             const DimensionedField<scalar, volMesh>& iF
373         ) const
374         {
375             return tmp<fvPatchScalarField>
376             (
377                 new windkesselOutletPressureFvPatchScalarField(*this, iF)
378             );
379         }
```

3.2.1.9 Member functions

The `windkesselOutletPressureFvPatchScalarField` class contains two important member functions, as shown in the Listing 3.12, that manage the boundary condition's behavior during simulations. These member functions include both evaluation and utility functions. The `updateCoeffs()` function is a key method responsible for updating the coefficients of the boundary condition based on the flow rate. This function ensures that the boundary condition remains consistent with the physical dynamics of the windkessel model. Additionally, the class contains the `write()` function which is responsible for outputting the boundary condition data to an output stream. This functionality is crucial for logging simulation results or exporting data for further analysis. These member functions enable the boundary condition object to interact seamlessly within the OpenFOAM framework, allowing for accurate simulation of outlet pressure dynamics in fluid simulations.

Listing 3.12: Declaration of member functions in header file

```
381     // Member functions
382
383         //- Evaluation function to update coefficients for the boundary
384         //  condition based on flow rate and model parameters
385         virtual void updateCoeffs();
386
387         //- Write boundary condition data to output stream
388         virtual void write(Ostream&) const;
```

3.2.2 Source file

This implementation defines the `windkesselOutletPressureFvPatchScalarField` boundary condition using constructors for initialization, an `updateCoeffs()` function for dynamic updates, and a `write()` function for data output. It supports both first-order and second-order backward differencing schemes to calculate time derivatives of pressures and flow rates for the available windkessel models and offers flexibility for user customization.

3.2.2.1 Includes and static data members

The `windkesselOutletPressureFvPatchScalarField.C` file includes several essential header files, as shown in the Listing 3.13, to provide necessary functionality and facilitate integration with the OpenFOAM framework. The `windkesselOutletPressureFvPatchScalarField.H` header defines the custom boundary condition class for windkessel outlet pressure, encapsulating the relevant parameters and methods for implementing the windkessel models. The registration of custom classes with OpenFOAM's runtime selection table, allowing dynamic selection of boundary conditions is enabled by including the `addToRunTimeSelectionTable.H` header file. `fvPatchFieldMapper.H` supports data mapping between patches, ensuring proper data transfer during mesh modifications. The `volFields.H` and `surfaceFields.H` headers handle volumetric and surface fields, respectively, allowing the management of physical quantities like pressure, velocity, and fluxes. The Backward differencing scheme for time integration, is defined by including `backwardDdtScheme.H` header file. Lastly, `word.H` provides the `word` class for handling string operations, which are crucial for input/output processing and data parsing in OpenFOAM simulations.

No static data members are declared.

Listing 3.13: Includes and static data members in source file

```
26 #include "windkesselOutletPressureFvPatchScalarField.H"
27 #include "addToRunTimeSelectionTable.H"
28 #include "fvPatchFieldMapper.H"
29 #include "volFields.H"
30 #include "surfaceFields.H"
31 #include "backwardDdtScheme.H"
32 #include "word.H"
33
34 // * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * * //
35
36 // (No additional details required for static members here)
```

3.2.2.2 Constructors

This section defines multiple constructors including a default constructor, a dictionary constructor, a mapping constructor, and two copy constructors.

A default constructor `windkesselOutletPressureFvPatchScalarField`, as shown in the Listing 3.14, that initializes the object with default values is defined to assign predefined values to the class variables. The constructor accepts a `fvPatch` and a `DimensionedField` as input and then initializes various parameters, including proximal resistance (`Rp_`), distal resistance (`Rd_`), compliance (`C_`), inductance (`L_`), distal pressure (`Pd_`), and fluid density (`rho_`). Additionally, it sets the historical pressure and flow rate variables to zero. A default Windkessel model (`WK3`) and second-order differencing scheme are also selected, while the time index is initialized to -1.

Listing 3.14: A default constructor in source file

```
38 // * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * //
39
40 //- Constructor: Initialize with default values
41 Foam::windkesselOutletPressureFvPatchScalarField::
42 windkesselOutletPressureFvPatchScalarField
43 (
44     const fvPatch& p,
45     const DimensionedField<scalar, volMesh>& iF
46 )
47 :
48     fixedValueFvPatchScalarField(p, iF), // Call base class constructor
49     Rp_(1),     // Proximal resistance
50     Rd_(1),     // Distal resistance
51     C_(1),      // Compliance
52     L_(1),      // Inductance
53     Pd_(0),     // Distal pressure
54     rho_(1),    // Fluid density
```

```
    Pooo_(0), Poo_(0), Po_(0), Pn_(0), // Initialize pressures to zero
    Qooo_(0), Qoo_(0), Qo_(0), Qn_(0), // Initialize flow rates to zero
    windkesselModel_(WK3),     // Default windkessel model
    diffScheme_(secondOrder),  // Default differencing scheme
    timeIndex_(-1)  // Initialize time index to invalid
{}
```

A dictionary constructor `windkesselOutletPressureFvPatchScalarField`, as shown in the Listing 3.15, that initializes the object from a dictionary allows for more flexible parameter initialization by extracting values directly from a dictionary, typically used for user input. This constructor accepts a `fvPatch`, `DimensionedField`, and `dictionary` as inputs. It retrieves parameters such as `Rp_`, `Rd_`, `C_`, `L_`, `Pd_`, and `rho_` from the dictionary, providing default values if the entries are not found. It sets the time index to -1.

Listing 3.15: A dictionary constructor in source file

```
//- Constructor: Initialize from dictionary (typically used in user input)
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict),      // Call base class constructor
    Rp_(dict.lookupOrDefault<scalar>("Rp", 1)),     // Proximal resistance
    Rd_(dict.lookupOrDefault<scalar>("Rd", 1)),     // Distal resistance
    C_(dict.lookupOrDefault<scalar>("C", 1)),       // Compliance
    L_(dict.lookupOrDefault<scalar>("L", 1)),       // Inductance
    Pd_(dict.lookupOrDefault<scalar>("Pd", 0)),     // Distal pressure
    rho_(dict.lookupOrDefault<scalar>("rho", 1)),   // Fluid density
    Pooo_(dict.lookupOrDefault<scalar>("Pooo", 0)),
    Poo_(dict.lookupOrDefault<scalar>("Poo", 0)),
    Po_(dict.lookupOrDefault<scalar>("Po", 0)),
    Pn_(dict.lookupOrDefault<scalar>("Pn", 0)),
    Qooo_(dict.lookupOrDefault<scalar>("Qooo", 0)),
    Qoo_(dict.lookupOrDefault<scalar>("Qoo", 0)),
    Qo_(dict.lookupOrDefault<scalar>("Qo", 0)),
    Qn_(dict.lookupOrDefault<scalar>("Qn", 0)),
    timeIndex_(-1)  // Time index reset
```

Moreover, it retrieves and maps the specified windkessel model type (e.g., `"Resistive"`, `"WK2"`, `"WK3"`, etc.) and differencing scheme (e.g., `"firstOrder"` or `"secondOrder"`) from the dictionary, as shown in the Listing 3.16. After retrieving these values, it prints relevant information about the selected model for debugging purposes.

Listing 3.16: Retrieval and mapping of model type and differencing scheme from dictionary

```
{

    Info << "\n\nApplying windkesselOutletPressure BC on patch: " << patch().name() << endl;

    //- Retrieve the model as a string and map to the enumerator
    word WKModelStr = dict.lookupOrDefault<word>("windkesselModel", "WK3");

    if (WKModelStr == "Resistive")
    {
        windkesselModel_ = Resistive;

        //- Print Windkessel model properties for debugging/verification
        Info << "\n\nProperties of Windkessel Model: \n"
             << "Model Type: 1-Element (Resistive) \n"
             << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
             << "Distal Pressure: Pd = " << Pd_ << " Pa \n"
             << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
    }
```

```
105     else if (WKModelStr == "WK2")
106     {
107         windkesselModel_ = WK2;
108 
109         //- Print Windkessel model properties for debugging/verification
110         Info << "\n\nProperties of Windkessel Model: \n"
111              << "Model Type: 2-Element (RC) \n"
112              << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
113              << "Compliance: C = " << Rd_ << " m^4s^2kg^-1 \n"
114              << "Distal Pressure: Pd = " << C_ << " Pa \n"
115              << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
116     }
117     else if (WKModelStr == "WK3")
118     {
119         windkesselModel_ = WK3;
120 
121         //- Print Windkessel model properties for debugging/verification
122         Info << "\n\nProperties of Windkessel Model: \n"
123              << "Model Type: 3-Element (RCR) \n"
124              << "Proximal Resistance: Rp = " << Rp_ << " kgm^-4s^-1 \n"
125              << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
126              << "Compliance: C = " << C_ << " m^4s^2kg^-1 \n"
127              << "Distal Pressure: Pd = " << Pd_ << " Pa \n"
128              << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
129     }
130     else if (WKModelStr == "WK4Series")
131     {
132         windkesselModel_ = WK4Series;
133 
134         //- Print Windkessel model properties for debugging/verification
135         Info << "\n\nProperties of Windkessel Model: \n"
136              << "Model Type: 4-Element (RCRL with L in Series to Rp) \n"
137              << "Proximal Resistance: Rp = " << Rp_ << " kgm^-4s^-1 \n"
138              << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
139              << "Compliance: C = " << C_ << " m^4s^2kg^-1 \n"
140              << "Inertance: L = " << L_ << " kgm^-4 \n"
141              << "Distal Pressure: Pd = " << Pd_ << " Pa \n"
142              << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
143     }
144     else if (WKModelStr == "WK4Parallel")
145     {
146         windkesselModel_ = WK4Parallel;
147 
148         //- Print Windkessel model properties for debugging/verification
149         Info << "\n\nProperties of Windkessel Model: \n"
150              << "Model Type: 4-Element (RCRL with L in Parallel to Rp) \n"
151              << "Proximal Resistance: Rp = " << Rp_ << " kgm^-4s^-1 \n"
152              << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
153              << "Compliance: C = " << C_ << " m^4s^2kg^-1 \n"
154              << "Inertance: L = " << L_ << " kgm^-4 \n"
155              << "Distal Pressure: Pd = " << Pd_ << " Pa \n"
156              << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
157     }
158     else
159     {
160         FatalErrorInFunction << "\n\nUnknown Windkessel Model: " << WKModelStr
161                              << "\nValid Windkessel Models (windkesselModel) are : \n\n"
162                              << " 5 \n ( \n Resistive \n WK2 \n WK3 \n WK4Series \n WK4Parallel \n ) \
    n"
163                              << exit(FatalError);
164     }
165 
166     //- Retrieve the scheme as a string and map to the enumerator
167     word schemeStr = dict.lookupOrDefault<word>("diffScheme", "secondOrder");
168 
169     Info << "Differencing Scheme for Windkessel Model: " << schemeStr << endl;
170 
171     if (schemeStr == "firstOrder")
```

```
    {
        diffScheme_ = firstOrder;
    }
    else if (schemeStr == "secondOrder")
    {
        diffScheme_ = secondOrder;
    }
    else
    {
        FatalErrorInFunction << "\n\nUnknown Differencing Scheme: " << schemeStr
                             << "\nValid Differencing Schemes (diffScheme) are : \n\n"
                             << " 2 \n ( \n firstOrder \n secondOrder \n ) \n"
                             << exit(FatalError);
    }

}
```

A mapping constructor `windkesselOutletPressureFvPatchScalarField`, as shown in the Listing 3.17, that maps an existing field onto a new patch facilitates the copying of the boundary condition from one patch to another, with an optional field mapping operation. This constructor accepts an existing `windkesselOutletPressureFvPatchScalarField` object, a new `fvPatch`, a `DimensionedField`, and a `fvPatchFieldMapper`. It copies the relevant properties, such as the Windkessel parameters, historical pressure and flow rate variables, and the model selection. The constructor allows the boundary condition to be reassigned to a new patch, ensuring that all the properties are appropriately transferred.

Listing 3.17: A mapping constructor in source file

```
//- Constructor: Map an existing field onto a new patch
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const windkesselOutletPressureFvPatchScalarField& ptf, // Existing field
    const fvPatch& p, // New patch to map onto
    const DimensionedField<scalar, volMesh>& iF, // Internal field reference
    const fvPatchFieldMapper& mapper // Field mapper for mapping operations
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper), // Map base class properties
    Rp_(ptf.Rp_),     // Copy proximal resistance parameter
    Rd_(ptf.Rd_),     // Copy distal resistance parameter
    C_(ptf.C_),       // Copy compliance parameter
    L_(ptf.L_),       // Copy inertance parameter
    Pd_(ptf.Pd_),     // Copy distal pressure parameter
    rho_(ptf.rho_),   // Copy fluid density
    Pooo_(ptf.Pooo_), // Copy previous state pressure variables
    Poo_(ptf.Poo_),
    Po_(ptf.Po_),
    Pn_(ptf.Pn_),
    Qooo_(ptf.Qooo_), // Copy previous state flow rate variables
    Qoo_(ptf.Qoo_),
    Qo_(ptf.Qo_),
    Qn_(ptf.Qn_),
    windkesselModel_(ptf.windkesselModel_), // Copy Windkessel model type
    diffScheme_(ptf.diffScheme_), // Copy numerical differencing scheme
    timeIndex_(ptf.timeIndex_) // Copy time index
{}
```

Two variations of copy constructors are defined. The first copy constructor, as shown in the Listing 3.18, of the `windkesselOutletPressureFvPatchScalarField` class is used to create a duplicate field, acceptind an existing `windkesselOutletPressureFvPatchScalarField` object as input and copies all its properties, including windkessel parameters, pressure and flow rate history, and model type. It is essentially a direct duplication of the object's state, enabling the creation of identical copies when necessary.

Listing 3.18: A copy constructor in source file

```
//- Copy constructor: Create a duplicate field with all properties
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const windkesselOutletPressureFvPatchScalarField& wkpsf // Source field
)
:
    fixedValueFvPatchScalarField(wkpsf), // Copy base class properties
    Rp_(wkpsf.Rp_),     // Copy proximal resistance
    Rd_(wkpsf.Rd_),     // Copy distal resistance
    C_(wkpsf.C_),       // Copy compliance
    L_(wkpsf.L_),       // Copy inertance
    Pd_(wkpsf.Pd_),     // Copy distal pressure
    rho_(wkpsf.rho_),   // Copy fluid density
    Pooo_(wkpsf.Pooo_), // Copy pressure variables
    Poo_(wkpsf.Poo_),
    Po_(wkpsf.Po_),
    Pn_(wkpsf.Pn_),
    Qooo_(wkpsf.Qooo_), // Copy flow rate variables
    Qoo_(wkpsf.Qoo_),
    Qo_(wkpsf.Qo_),
    Qn_(wkpsf.Qn_),
    windkesselModel_(wkpsf.windkesselModel_), // Copy Windkessel model type
    diffScheme_(wkpsf.diffScheme_), // Copy differencing scheme
    timeIndex_(wkpsf.timeIndex_) // Copy time index
{}
```

Another copy constructor with a new internal field reference, as shown in the Listing 3.19, allows the creation of a duplicate `windkesselOutletPressureFvPatchScalarField` object, but with a different internal field reference. It takes an existing `windkesselOutletPressureFvPatchScalarField` object and a new `DimensionedField` as input. While it copies all the class properties, it ensures that the internal field reference is updated, making it suitable for scenarios where the internal field needs to be modified or reassigned without altering other properties of the object.

Listing 3.19: Second copy constructor in source file

```
//- Copy constructor with new internal field reference
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const windkesselOutletPressureFvPatchScalarField& wkpsf, // Source field
    const DimensionedField<scalar, volMesh>& iF // New internal field
)
:
    fixedValueFvPatchScalarField(wkpsf, iF), // Map base class properties
    Rp_(wkpsf.Rp_),     // Copy proximal resistance
    Rd_(wkpsf.Rd_),     // Copy distal resistance
    C_(wkpsf.C_),       // Copy compliance
    L_(wkpsf.L_),       // Copy inertance
    Pd_(wkpsf.Pd_),     // Copy distal pressure
    rho_(wkpsf.rho_),   // Copy fluid density
    Pooo_(wkpsf.Pooo_), // Copy pressure variables
    Poo_(wkpsf.Poo_),
    Po_(wkpsf.Po_),
    Pn_(wkpsf.Pn_),
    Qooo_(wkpsf.Qooo_), // Copy flow rate variables
    Qoo_(wkpsf.Qoo_),
    Qo_(wkpsf.Qo_),
    Qn_(wkpsf.Qn_),
    windkesselModel_(wkpsf.windkesselModel_), // Copy Windkessel model type
    diffScheme_(wkpsf.diffScheme_), // Copy differencing scheme
    timeIndex_(wkpsf.timeIndex_) // Copy time index
{}
```

3.2.2.3 Member functions

There are two member functions in the source file including `updateCoeffs()` which updates the pressure and `write()` which writes the current state of the boundary condition to an output stream.

The `updateCoeffs()` function, as shown in the Listing 3.20, is a member function of the `Foam::windkesselOutletPressureFvPatchScalarField` class, responsible for updating the coefficients associated with the boundary condition for a windkessel model in the CFD simulations. The function checks if the coefficients have already been updated for the current time step and performs necessary calculations to update pressure and flow rate histories as required by the specific windkessel model. It handles different models (`Resistive`, `WK2`, `WK3`, `WK4Series`, `WK4Parallel`) and different differencing schemes (`firstOrder`, `secondOrder`) to compute pressure values based on flow rates and other parameters.

Listing 3.20: `updateCoeffs()` member function in source file

```
275 // * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * * //
276
277 //- Update coefficients for boundary condition
278 void Foam::windkesselOutletPressureFvPatchScalarField::updateCoeffs()
```

Initially, the function checks if the coefficients have been updated by calling the `updated()` method, as shown in the Listing 3.21. If the coefficients have already been updated for the current time step, the function returns early, avoiding redundant calculations. If not, it proceeds to retrieve essential data from the simulation database, including the time step size (`dt`), flux field (`phi`), and boundary pressure field (`p`). The total area of the patch is then calculated using the `patch().magSf()` method, which returns the magnitude of the surface vector of the patch.

Listing 3.21: Checking the coefficients update and the retrieval of essential data from the simulation database within `updateCoeffs()` function

```
280     //- Check if coefficients have already been updated for this time step
281     if (updated())
282     {
283         return;
284     }
285
286     //- Retrieve the time step size from the database
287     const scalar dt = db().time().deltaTValue();
288
289     //- Retrieve the flux field (phi) from the database
290     const surfaceScalarField& phi =
291         db().lookupObject<surfaceScalarField>("phi");
292
293     //- Retrieve the boundary pressure field from the database
294     const fvPatchField<scalar>& p =
295         db().lookupObject<volScalarField>("p").boundaryField()[this->patch().index()];
296
297     //- Calculate the total area of the patch
298     scalar area = gSum(patch().magSf());
```

Next, the pressure and flow rate histories are updated if the time index has advanced, as shown in the Listing 3.22. This involves shifting the historical pressure and flow rate values (e.g., `Pooo_`, `Poo_`, `Po_`, `Pn_`, and `Qooo_`, `Qoo_`, `Qo_`, `Qn_`) to make space for the latest values.

Listing 3.22: Updating pressure and flow rate histories within `updateCoeffs()` function

```
300     //- Update pressure and flow rate histories if the time index has advanced
301     if (db().time().timeIndex() != timeIndex_)
302     {
303         timeIndex_ = db().time().timeIndex();
304
305         //- Update pressure history variables
306         Pooo_ = Poo_;
307         Poo_ = Po_;
308         Po_ = Pn_;
```

```
309         Pn_ = gSum(p * patch().magSf()) / area; // Compute mean pressure over the patch
310
311         //- Update flow rate history variables
312         Qooo_ = Qoo_;
313         Qoo_ = Qo_;
314         Qo_ = Qn_;
315         Qn_ = gSum(phi.boundaryField()[patch().index()]); // Compute total flux
316     }
```

After updating the history, the function proceeds to calculate the new pressure value (`Pn_`), as shown in the Listing 3.23. The pressure update depends on the windkessel model selected and the differencing scheme applied. Each windkessel model (`Resistive`, `WK2`, `WK3`, `WK4Series`, `WK4Parallel`) employs a different formula for pressure computation, which can either use a backward Euler scheme or an explicit scheme, depending on the model and settings. For each model, the appropriate formula for pressure calculation is executed. The `Rd_`, `C_`, `Pd_`, and other parameters are used to calculate the new pressure based on the flow rate (`Qn_`) and its change over time. The function also handles both first-order and second-order differencing schemes, adjusting the pressure computation accordingly. If an unknown model or differencing scheme is encountered, an error is raised using `FatalErrorInFunction`.

Listing 3.23: Calculating the new pressure value (`Pn_`) within `updateCoeffs()` function

```
318     //- Calculate new pressure using backward differencing scheme
319     if
320     (
321         db().time().timeIndex()>1 // Perform calculations only if the time index is greater than 1
322     )
323     {
324         //- Switch to the specified windkessel model
325         switch (windkesselModel_)
326         {
327             case Resistive: // Single-element resistive model (R)
328
329                 Pn_ = Rd_ * Qn_ + Pd_;
330
331                 break;
332
333             case WK2: // 2-element windkessel model (RC)
334
335                 //- Switch to the specified differencing scheme
336                 switch (diffScheme_)
337                 {
338                     case firstOrder:
339
340                         Pn_ = Rd_ * Qn_
341                             + Pd_
342                             + Rd_ * C_ * Po_ / dt;
343
344                         Pn_ /= (1.0 + Rd_ * C_ / dt) + SMALL;
345
346                         break;
347
348                     case secondOrder:
349
350                         Pn_ = Rd_ * Qn_
351                             + Pd_
352                             - Rd_ * C_ * ((Poo_ - 4 * Po_) / (2 * dt));
353
354                         Pn_ /= (1.0 + 3 * Rd_ * C_ / (2 * dt)) + SMALL;
355
356                         break;
357
358                     default:
359
360                         FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
361                 }
```

```
362
363                 break;
364
365             case WK3: // 3-element windkessel model (RCR)
366
367                 //- Switch to the specified differencing scheme
368                 switch (diffScheme_)
369                 {
370                     case firstOrder:
371
372                         Pn_ = Rp_ * Rd_ * C_ * ((Qn_ - Qo_) / dt)
373                             + (Rp_ + Rd_) * Qn_
374                             + Pd_
375                             + Rd_ * C_ * (Po_ / dt);
376
377                         Pn_ /= (1.0 + Rd_ * C_ / dt) + SMALL;
378
379                         break;
380
381                     case secondOrder:
382
383                         Pn_ = Rp_ * Rd_ * C_ * ((3 * Qn_ - 4 * Qo_ + Qoo_) / (2 * dt))
384                             + (Rp_ + Rd_) * Qn_
385                             + Pd_
386                             - Rd_ * C_ * ((Poo_ - 4 * Po_) / (2 * dt));
387
388                         Pn_ /= (1.0 + 3 * Rd_ * C_  / (2 * dt)) + SMALL;
389
390                         break;
391
392                     default:
393
394                         FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
395                 }
396
397                 break;
398
399             case WK4Series: // 4-element series windkessel model (RCRL-Series)
400
401                 //- Switch to the specified differencing scheme
402                 switch (diffScheme_)
403                 {
404                     case firstOrder:
405
406                         Pn_ = (Rp_ + Rd_) * Qn_
407                             + (L_ + Rp_ * Rd_ * C_) * ((Qn_ - Qo_) / dt)
408                             + Rd_ * C_ * L_ * ((Qn_ - 2 * Qo_ + Qoo_) / (pow (dt, 2)))
409                             + Pd_
410                             + Rd_ * C_ * (Po_ / dt);
411
412                         Pn_ /= (1.0 + Rd_ * C_ / dt) + SMALL;
413
414                         break;
415
416                     case secondOrder:
417
418                         Pn_ = (Rp_ + Rd_) * Qn_
419                             + (L_ + Rp_ * Rd_ * C_) * ((3 * Qn_ - 4 * Qo_ + Qoo_) / (2 * dt))
420                             + Rd_ * C_ * L_ * ((2 * Qn_ - 5 * Qo_ + 4 * Qoo_ - Qooo_) / (pow (dt, 2)))
421                             + Pd_
422                             - Rd_ * C_ * ((Poo_ - 4 * Po_) / (2 * dt));
423
424                         Pn_ /= (1.0 + 3 * Rd_ * C_  / (2 * dt)) + SMALL;
425
426                         break;
427
428                     default:
429
```

```
                        FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
                }

                break;

            case WK4Parallel: // 4-element parallel windkessel model (RCRL-Parallel)

                //- Switch to the specified differencing scheme
                switch (diffScheme_)
                {
                    case firstOrder:

                        Pn_ = Rd_ * Qn_
                            + L_ * (1 + Rd_ / Rp_ ) * ((Qn_ - Qo_) / dt)
                            + Rd_ * C_ * L_ * ((Qn_ - 2 * Qo_ + Qoo_) / (pow (dt, 2)))
                            + Pd_
                            + ((L_ + Rp_ * Rd_ * C_) / Rp_) * (Po_ / dt)
                            + (Rd_ * C_ * L_ / Rp_) * ((2 * Po_ - Poo_) / (pow (dt, 2)));

                        Pn_ /= (1.0 + ((L_ + Rp_ * Rd_ * C_) / (Rp_ * dt)) + (Rd_ * C_ * L_ / (Rp_ *
    pow (dt, 2)))) + SMALL;

                        break;

                    case secondOrder:

                        Pn_ = Rd_ * Qn_
                            + L_ * (1 + Rd_ / Rp_ ) * ((3 * Qn_ - 4 * Qo_ + Qoo_) / (2 * dt))
                            + Rd_ * C_ * L_ * ((2 * Qn_ - 5 * Qo_ + 4 * Qoo_ - Qooo_) / (pow (dt, 2)))
                            + Pd_
                            - ((L_ + Rp_ * Rd_ * C_) / Rp_) * ((Poo_ - 4 * Po_) / (2 * dt))
                            - (Rd_ * C_ * L_ / Rp_) * ((-5 * Po_ + 4 * Poo_ - Pooo_) / (pow (dt, 2)));

                        Pn_ /= (1.0 + (3 * (L_ + Rp_ * Rd_ * C_) / (2 * Rp_ * dt)) + (2 * Rd_ * C_ *
    L_ / (Rp_ * pow (dt, 2)))) + SMALL;

                        break;

                    default:

                        FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
                }

                break;

            default:

                FatalErrorInFunction << "Unknown Windkessel Model!" << exit(FatalError);
        }

    }
```

Finally, the pressure update is applied to the boundary field, as shown in the Listing 3.24. The pressure (`Pn_`) is directly updated using the formula `Pn_ / (rho_ + SMALL)`. The function then calls the base class's `updateCoeffs()` method to finalize the update.

Listing 3.24: Applying the pressure to the boundary field within `updateCoeffs()` function

```
    //- Apply implicit pressure update to the boundary field
    operator==(Pn_ / (rho_ + SMALL));

    //- Call base class function to finalize the update
    fixedValueFvPatchScalarField::updateCoeffs();
```

The `updateCoeffs()` function ensures that the boundary conditions related to the windkessel model are correctly updated for each time step, accounting for various models and differencing schemes. The function is critical in maintaining the accuracy and stability of the simulation by

applying the appropriate windkessel model and calculation scheme for pressure updates based on the flow rates and other parameters.

The second function, `write()` function, as shown in the Listing 3.25, is a member of the `Foam::windkesselOutletPressureFvPatchScalarField` class, which is responsible for writing the boundary condition properties of the windkessel model to an output stream. The function gathers various parameters that define the model and its configuration, such as resistance values, pressure values, and the explicit/implicit solver flag, and outputs them to the specified stream. This function ensures that all relevant model details, including history parameters and the chosen differencing scheme, are captured and written to the output for further analysis or post-processing.

Listing 3.25: `write()` member function in source file

```
487 //- Write the boundary condition properties to an output stream
488 void Foam::windkesselOutletPressureFvPatchScalarField::write(Ostream& os) const
```

At the beginning of the function, the base class function `fvPatchScalarField::write(os)` is called to write common boundary field properties, as shown in the Listing 3.26. This serves as a starting point to ensure that all necessary boundary-related information is included in the output. Moreover, the details including distal resistance (`Rd_`), distal pressure (`Pd_`) and fluid density (`rho_`) which are common across all LPN models are written to the output stream.

Listing 3.26: Calling the class to write common boundary field properties and writing common windkessel model details within `write()` function

```
490     //- Call the base class function to write common boundary field properties
491     fvPatchScalarField::write(os);
492
493     //- Write the common windkessel model details to the output stream
494     os.writeKeyword("Rd") << Rd_ << token::END_STATEMENT << nl; // Resistance parameter
495     os.writeKeyword("Pd") << Pd_ << token::END_STATEMENT << nl; // Prescribed pressure
496     os.writeKeyword("rho") << rho_ << token::END_STATEMENT << nl; // Fluid density
```

The function then proceeds to write the specific details of the windkessel model being used, as shown in the Listing 3.27. Depending on the value of the `windkesselModel_` variable, different properties are written to the output stream. For all the models, details including distal resistance (`Rd_`), distal pressure (`Pd_`) and fluid density (`rho_`) have been written before. Henceforth, for the `Resistive` model, the string for `windkesselModel_` is written in addition to the common details. For the `WK2` model, in addition to the common details, the compliance parameter (`C_`) is included. For the `WK3` model, details, apart from the common parameters, include the proximal resistance (`Rp_`) and compliance (`C_`). For the `WK4Series` and `WK4Parallel` models, the additional inductance (`L_`) parameter is included, along with the proximal resistance (`Rp_`), compliance (`C_`), and common parameters. If an unknown windkessel model is encountered, an error is raised using `FatalErrorInFunction`, which should never be reached as wrong entries have already been checked in constructors.

Listing 3.27: Writing specific windkessel model details within `write()` function

```
498     //- Write the specific windkessel model details to the output stream
499     switch (windkesselModel_)
500     {
501         case Resistive:
502             os.writeKeyword("windkesselModel") << "Resistive" << token::END_STATEMENT << nl;
503             break;
504
505         case WK2:
506             os.writeKeyword("windkesselModel") << "WK2" << token::END_STATEMENT << nl;
507             os.writeKeyword("C") << C_ << token::END_STATEMENT << nl; // Compliance
508             break;
509
510         case WK3:
511             os.writeKeyword("windkesselModel") << "WK3" << token::END_STATEMENT << nl;
512             os.writeKeyword("Rp") << Rp_ << token::END_STATEMENT << nl; // Proximal resistance
513             os.writeKeyword("C") << C_ << token::END_STATEMENT << nl; // Compliance
```

```
514             break;
515
516         case WK4Series:
517             os.writeKeyword("windkesselModel") << "WK4Series" << token::END_STATEMENT << nl;
518             os.writeKeyword("Rp") << Rp_ << token::END_STATEMENT << nl;
519             os.writeKeyword("C") << C_ << token::END_STATEMENT << nl;
520             os.writeKeyword("L") << L_ << token::END_STATEMENT << nl; // Inductance
521             break;
522
523         case WK4Parallel:
524             os.writeKeyword("windkesselModel") << "WK4Parallel" << token::END_STATEMENT << nl;
525             os.writeKeyword("Rp") << Rp_ << token::END_STATEMENT << nl;
526             os.writeKeyword("C") << C_ << token::END_STATEMENT << nl;
527             os.writeKeyword("L") << L_ << token::END_STATEMENT << nl;
528             break;
529
530         default:
531             FatalErrorInFunction << "Unknown Windkessel Model: " << windkesselModel_ << exit(
    FatalError);
532     }
```

Next, the function writes the differencing scheme used for calculations. The `diffScheme_` variable is checked, and the corresponding scheme—either `firstOrder` or `secondOrder`—is written to the output, as shown in the Listing 3.28. If an unknown differencing scheme is encountered, an error is raised, which should never be reached as wrong entries have already been checked in constructors.

Listing 3.28: Writing differencing scheme used for calculations within `write()` function

```
534     //- Write the differencing scheme used for calculations
535     switch (diffScheme_)
536     {
537         case firstOrder:
538             os.writeKeyword("diffScheme") << "firstOrder" << token::END_STATEMENT << nl;
539             break;
540
541         case secondOrder:
542             os.writeKeyword("diffScheme") << "secondOrder" << token::END_STATEMENT << nl;
543             break;
544
545         default:
546             FatalErrorInFunction << "Unknown differencing scheme: " << diffScheme_ << exit(FatalError)
    ;
547     }
```

Next, the function writes the flow rate and pressure histories, as shown in the Listing 3.29. This includes the third past flow rate (`Qooo_`) and pressure (`Pooo_`), the second past flow rate (`Qoo_`) and pressure (`Poo_`), the previous flow rate (`Qo_`) and pressure (`Po_`), and the current flow rate (`Qn_`) and pressure (`Pn_`). These flow rate and pressure histories are necessary for a perfect restart of the simulation. Finally, the function calls the `writeEntry()` function to write the boundary field value entry to the output stream. This ensures that the boundary field value is included in the output along with the model parameters.

Listing 3.29: Writing flow rate and pressure histories and boundary field entry within `write()` function

```
549     //- Write flow history parameters
550     os.writeKeyword("Qooo") << Qooo_ << token::END_STATEMENT << nl; // Third past flow rate
551     os.writeKeyword("Qoo") << Qoo_ << token::END_STATEMENT << nl; // Second past flow rate
552     os.writeKeyword("Qo") << Qo_ << token::END_STATEMENT << nl; // Previous flow rate
553     os.writeKeyword("Qn") << Qn_ << token::END_STATEMENT << nl; // Current flow rate
554
555     //- Write pressure history parameters
556     os.writeKeyword("Pooo") << Pooo_ << token::END_STATEMENT << nl; // Third past pressure
557     os.writeKeyword("Poo") << Poo_ << token::END_STATEMENT << nl; // Second past pressure
558     os.writeKeyword("Po") << Po_ << token::END_STATEMENT << nl; // Previous pressure
559     os.writeKeyword("Pn") << Pn_ << token::END_STATEMENT << nl; // Current pressure
```

```
560
561     //- Write the boundary field value entry to the output stream
562     writeEntry("value", os);
```

The `write()` function serves to export the essential properties and configuration details of the windkessel model, along with its flow and pressure history, to an output stream. It ensures that all necessary information for later analysis is made available in a standardized format. This is crucial for simulations where boundary conditions and their histories play a significant role in the overall model behavior.

3.2.2.4 Runtime type selection

The final part of the source file, as shown in Listing 3.30, defines the boundary field type for the `windkesselOutletPressureFvPatchScalarField` class within the `Foam` namespace.

The `makePatchTypeField` macro is used to establish the type of the boundary field, associating the class `fvPatchScalarField` with `windkesselOutletPressureFvPatchScalarField`. This step is crucial as it registers the specific boundary field type, enabling it to be used within the broader framework of the code. This declaration ensures that the `windkesselOutletPressureFvPatchScalarField` class will be recognized as a valid type for boundary fields in simulations, particularly for those involving windkessel models. The macro call effectively links the class with the appropriate functionalities of the `fvPatchScalarField`, allowing for seamless integration within the computational framework.

Listing 3.30: Runtime type selection in source file

```
567 namespace Foam
568 {
569     //- Define the boundary field type for this class
570     makePatchTypeField
571     (
572         fvPatchScalarField,                         // Base class
573         windkesselOutletPressureFvPatchScalarField  // Derived class
574     );
575 }
```

Chapter 4

Implementation of Coronary LPN BCs

4.1 Solution methodology

The implementation of the coronary lumped parameter network (LPN) model boundary condition (BC) follows a systematic approach, virtually same as that followed for the implementation of wind-kessel model boundary conditions. However, the ordinary differential equations (ODE) solved for this BC are based on the LPN of downstream coronary vascular beds with intramyocardial pressure term, as formulated previously. The boundary condition is designed to account for complex interactions between pressure, flow rate, and different LPN parameters. The methodology encompasses the modeling of pressure dynamics at the coronary outlet, incorporating historical and current flow and pressure data as well as intramyocardial pressure time-series data provided through a file, while ensuring that the boundary conditions are updated at each time step, as shown in the Algorithm 2.

Initially, the boundary condition is established with a set of predefined parameters, including resistances and compliance values, as well as a scaling factor for intramyocardial pressure (P_{im}). The boundary condition relies on a time-dependent numerical scheme to update the coefficients governing the system's behavior. This is achieved through the `updateCoeffs` method, where the time derivatives for flow rate and pressure are recalculated using either a first-order or second-order backward differencing scheme, depending on the user's preference. For complex problems, the choice between first-order and second-order differencing might be critical for controlling the numerical stability and accuracy of the solution. The second-order backward differencing shceme is generally consider more accurate than the first-order scheme to calculate first or higher time or spatial derivatives.

To ensure consistency and physical relevance, immediate flow rate and pressure histories are maintained throughout the simulation. These histories are updated whenever the time step changes. The current flow rate is obtained by summing the flux across the boundary faces, while the pressure is computed by taking the area-weighted average of the pressure field at the boundary. These historical values are then used in conjunction with the chosen differencing scheme to compute updated pressure values by solving system of two linear first-order ODEs obtained for coronary LPN model at each time step. The calculations incorporate arterial and veinous-side resistances and arterial compliance, intramyocardial compliance and intramyocardial pressure data, which modulate the pressure based on the instantaneous flow rates and historical pressure values.

The intramyocardial pressure, denoted as P_{im}, plays a crucial role in the boundary condition. It is determined by interpolating from a dataset provided by the user, which is loaded from an external file. The `loadPimData` method facilitates the extraction of this data, while the `interpolatePim` method uses linear interpolation to compute the P_{im} value corresponding to the current simulation time. This is done to accounts for the periodic nature of the data, ensuring that the P_{im} values are appropriately adjusted over the course of the simulation.

Algorithm 2 Coronary Outlet Pressure Boundary Condition

```
Define the BC as an object and initialize with default parameter values
Read parameters, lookup for the PimFile, differencing scheme, and initialize from dictionary
for each time-step do
    Retrieve time step size (dt) from database
    Retrieve the boundary flux (phi) and pressure (p) fields from database
    Calculate patch area (area)
    if time index has changed then
        Update flow history variables (Qo, Qoo, Qo, Qn)
        Update pressure history variables (Po, Poo, Po, Pn)
        Update intramyocardial pressure history variables (Pimoo, Pimo, Pimn)
        Update intermediate pressure history variables (Pioo, Pio, Pin)
    end if
    if time index > 1 then
        switch (differencing scheme)
          case firstOrder:
            Calculate new intermediate pressure (Pin) using first-order scheme
            Calculate new outlet pressure (Pn) using first-order scheme
          case secondOrder:
            Calculate new intermediate pressure (Pin) using second-order scheme
            Calculate new outlet pressure (Pn) using second-order scheme
          default:
            Handle unsupported differencing scheme
        end switch
    end if
    Apply implicit pressure update (Pn / (rho + SMALL))
    Finalize boundary condition update (call base class updateCoeffs)
end for
for each writeTime do
    Write common boundary field properties
    Write LPN model parameters (resistances, compliances)
    Write differencing scheme (firstOrder or secondOrder)
    Write flow history values (Qoo, Qo, Qn)
    Write pressure history values (Pimoo, Pimo, Pimn, Pioo, Pio, Pin, Poo, Po, Pn)
    Finalize writing of boundary field values using writeEntry("value")
end for
```

It should be noted that the ability of the linear interpolation process for P_{im} to handle the potential periodicity in the data and to ensure that the time-varying nature of the intramyocardial pressure is accurately captured depends on the resolution of the provided data and the time-step size for the simulation.

In addition to the core functionality, the boundary condition provides detailed output capabilities, allowing the user to examine the evolution of the flow rate and pressure history throughout the simulation. During the simulation, depending on the specified `writeTime`, the `write` method outputs the current values of various parameters, including resistance, compliance, flow rate, and pressure histories. This information is valuable for post-processing and analysis, enabling the user to monitor the behavior of the boundary condition over time.

Finally, the BC is registered within the OpenFOAM framework using the `makePatchTypeField` macro, which ensures that the model can be selected and applied within OpenFOAM simulations. This integration with the OpenFOAM framework allows the boundary condition to be used in conjunction with other solvers and models, facilitating comprehensive simulations of cardiovascular systems.

4.2 Implementation in OpenFOAM

The coronary lumped parameter network (LPN) boundary condition is implemented in `foam-extend 4.1` using a header and a source file described below in detail.

4.2.1 Header file

The header file `coronaryOutletPressureFvPatchScalarField.H` contains the description of the implemented boundary condition and all the declarations necessary for the implementation of the numerics in the source file.

4.2.1.1 Description

The description added at the start of the header file gives a brief overview of coronary lumped parameter network (LPN) model implemented in the source file. The coronary LPN boundary condition is designed to model outlet pressure using the lumped parameter network (LPN) model, which represents downstream coronary vascular beds. This approach is particularly suited for simulating hemodynamics in coronary artery systems. The LPN model is widely employed in cardiovascular simulations due to its ability to capture the intricate interactions between the arterial system and various lumped components representing resistance, compliance, and pressure dynamics.

The LPN model relies on a system of ordinary differential equations (ODEs) to relate pressure (P) and flow rate (Q) at the boundary. Key parameters incorporated in this model include arterial resistance (R_a), micro-arterial resistance (R_{am}), arterial compliance (C_a), intramyocardial compliance (C_{im}), intramyocardial pressure (P_{im}), venous resistance (R_v), and micro-venous resistance (R_{vm}). Additionally, a distal pressure term (P_v) is included distal to R_v on the venous side. To simplify numerical computations, venous compliance (C_v) is excluded. The LPN model has been shown in the literature to efficiently reproduce characteristic coronary flow and pressure curves.

The system of ODEs governing the LPN model for downstream coronary vascular beds consists of two equation given by Eq. (2.41) and Eq. (2.42) which give the intermediate and coronary outlet pressure values, respectively. Both of these equations should be solved simultaneously to calculates the coronary outlet pressure $P(t)$ based on volumetric flow rate $Q(t)$ and LPN model parameters. The boundary condition computes the pressure (P_n) at each time step using either the first-order discretized form of the ODEs given by Eq. (2.43) and Eq. (2.44) or the second-order discretized form of the ODEs given by Eq. (2.45) and Eq. (2.46), ensuring that the physiological behavior is modeled accurately.

The method to specify the boundary condition for any patch in the `0/p` dictionary is shown in the Listing 4.1.

Listing 4.1: Specification of `coronaryOutletPressure` BC in the `0/p` dictionary

```
<patchName>
{
type                coronaryOutletPressure;
Ra                  <value>;        //Arterial resistance in [kgm^-4s^-1], Default: 1
Ram                 <value>;        //Micro-arterial resistance in [kgm^-4s^-1], Default: 1
Rv                  <value>;        //Veinous resistance in [kgm^-4s^-1], Default: 1
Rvm                 <value>;        //Micro-venous resistance in [kgm^-4s^-1], Default: 0
Ca                  <value>;        //Arterial compliance in [m^4s^2kg^-1], Default: 1
Cim                 <value>;        //Intramyocardial compliance in in [m^4s^2kg^-1], Default: 1
PimFile             "<Path to Pim Data File>"; //Path to Pim time-series data file, Default: PimData
PimScaling          <value>;        // Intramyocardial pressure scaling factor, Default: 1
Pv                  <value>;        //Pressure distal to Rv in [Pa], Default: 0
rho                 <value>;        // Fluid density in [kgm^-3], Default: 1
diffScheme          <firstOrder or secondOrder>; // Differencing scheme, Default: secondOrder
value               uniform 0;
}
```

4.2.1.2 Header guard and includes

The header file starts with preprocessor directives, as shown in the Listing 4.2, to ensure that the header file is not included multiple times during compilation.

The directive `#ifndef coronaryOutletPressureFvPatchScalarField_H` checks whether the macro `coronaryOutletPressureFvPatchScalarField_H` has already been defined. If it has not been defined, the macro is defined using `#define`, effectively marking the file as included. This prevents potential redefinition errors and ensures that the file's content is processed only once. Moreover, the file includes several essential header files and libraries to support the implementation of the boundary condition. The inclusion of `"fixedValueFvPatchFields.H"` provides access to functionalities related to fixed-value boundary conditions in finite volume patch fields, forming the foundation for implementing the coronary outlet pressure boundary condition. The `"scalar.H"` file is included to allow operations involving scalar values, which are fundamental in numerical calculations for pressure and flow. The `"word.H"` header is used to manage strings or keywords, which are necessary for handling dictionary entries and parameter configurations. Moreover, standard C++ libraries are included to enhance the functionality of the code. The inclusion of `<vector>` ensures that the `std::vector` container can be utilized. Similarly, the inclusion of `<utility>` provides access to the `std::pair` template, which is useful for storing and manipulating paired data.

Listing 4.2: Header guard and includes in header file

```
149 #ifndef coronaryOutletPressureFvPatchScalarField_H
150 #define coronaryOutletPressureFvPatchScalarField_H
151
152 #include "fixedValueFvPatchFields.H"
153 #include "scalar.H"
154 #include "word.H"
155 #include <vector>  // Required for std::vector
156 #include <utility> // Required for std::pair
```

4.2.1.3 Namespace declaration

The header file `coronaryOutletPressureFvPatchScalarField.H` containing relevant declarations for the `coronaryOutletPressureFvPatchScalarField` class resides in the `Foam` namespace, which encapsulates all OpenFOAM-specific classes and functions.

4.2.1.4 Enumerations definition

Before the class declaration, the code includes one enumerations `DifferencingScheme`, as shown in the Listing 4.3, which defines the numerical schemes used for approximating time derivatives in the system of ordinary differential equations (ODEs) representing the coronary LPN model. The first-order backward differencing scheme and the second-order backward differencing scheme are included, each of which is used to discretize time derivatives in the model simulations.

Listing 4.3: Enumerations definition within `Foam` namespace in header file

```
163 // Enumerator for numerical differencing schemes used in the LPN model
164 enum DifferencingScheme
165 {
166     firstOrder = 0, // First-order backward differencing scheme
167     secondOrder = 1 // Second-order backward differencing scheme
168 };
```

4.2.1.5 Class declaration

The class `coronaryOutletPressureFvPatchScalarField` is declared as shown in the Listing 4.4, as a subclass of `fixedValueFvPatchScalarField`, which is used to represent a boundary condition for outlet pressure in fluid simulations. This class encapsulates the properties and methods required for the coronary LPN model BC. By inheriting from `fixedValueFvPatchScalarField`, it benefits

from predefined functionality for handling scalar fields and boundary conditions in the finite volume method framework used by OpenFOAM. The class declaration establishes the structure of the object, including both the public and private sections, and facilitates interaction with other components of the simulation environment.

Listing 4.4: `coronaryOutletPressureFvPatchScalarField` class definition in header file

```
class coronaryOutletPressureFvPatchScalarField
:
    public fixedValueFvPatchScalarField
```

4.2.1.6 Private data

The private data members of the `coronaryOutletPressureFvPatchScalarField` class are essential for the internal workings of the coronary LPN model. These include various physical parameters and state variables. Henceforth, in the private section, as shown in the Listing 4.5, the class defines several LPN model parameters including resistances (`Ra_, Ram_, Rv_, Rvm_`), compliances (`Ca_, Cim_`), distal pressure (`Pv_`), fluid density (`rho_`), intramyocardial pressure scaling factor (`PimScaling_`). The state variables keep track of pressures and flow rates at different time steps. The time differencing scheme is specified by `diffScheme_`. For intramyocardial pressure time series data handling, `PimFile_` is defined as `Foam::fileName` to store the path to the data file specified in the dictionary. A vector `PimData_` is defined to store the P_{im} time series data interpolated from the given file. The time index for historical data updates is stored in `timeIndex_`.

Listing 4.5: Declaration of private data members in header file

```
private:

    // Private data

        //- Model parameters
        scalar Ra_;          // Arterial resistance
        scalar Ram_;         // Micro-arterial resistance
        scalar Rv_;          // Veinous resistance
        scalar Rvm_;         // Micro-veinous resistance
        scalar Ca_;          // Arterial compliance
        scalar Cim_;         // Intramyocardial compliance
        scalar PimScaling_;  // Scaling factor for intramyocardial pressure
        scalar Pv_;          // Pressure distal to Rv
        scalar rho_;         // Fluid density

        //- State variables for flow rates and pressures
        scalar Qoo_, Qo_, Qn_;       // Historical flow rate values
        scalar Poo_, Po_, Pn_;       // Historical pressure values
        scalar Pioo_, Pio_, Pin_;    // Historical intermediate pressure values
        scalar Pimoo_, Pimo_, Pimn_; // Historical intramyocardial pressure values

        //- LPN model configuration
        DifferencingScheme diffScheme_; // Numerical differencing scheme
        Foam::fileName PimFile_;      // File containing intramyocardial pressure data

        //- Intramyocardial pressure data
        std::vector<std::pair<scalar, scalar>> PimData_; // Time-series data for (time, Pim)

        //- Time index
        label timeIndex_;             // Current time index for updates
```

4.2.1.7 Helper methods

To read and interpolate the intramyocardial pressure time series data, few helper methods are defined in the private section of the class, as shown in the Listing 4.6.

A helper method `loadPimData` is defined to load the data from the file placed at the given path. Another helper method `expandEnvironmentVariables` is defined to expand the any environment variables that the data file path may contain. While `interpolatePim` is defined to interpolate the intramyocardial pressure data during the simulation for the flow time at each time-step.

Listing 4.6: Declaration of helper methods in header file

```
225     // Helper methods
226
227         //- Function to interpolate intramyocardial pressure (Pim) data at a given time
228         scalar interpolatePim(const scalar& time) const;
229
230         //- Function to load intramyocardial pressure (Pim) data from a specified file
231         void loadPimData(const word& fileName);
232
233         //- Function to expands environment variables in a given string
234         static std::string expandEnvironmentVariables(const std::string& input);
```

4.2.1.8 Runtime type

The public data members of the `coronaryOutletPressureFvPatchScalarField` class allow access to key functions and methods. The class is equipped with runtime type information through the `TypeName("coronaryOutletPressure")` declaration in the public section, as shown in the Listing 4.7, which allows OpenFOAM's runtime system to correctly handle the object type during simulations.

Listing 4.7: Declaration of runtime type in header file

```
236 public:
237
238     //- Runtime type information
239     TypeName("coronaryOutletPressure");
```

4.2.1.9 Constructors

The `coronaryOutletPressureFvPatchScalarField` class provides several constructors to support different initialization methods, which enable flexibility in how the boundary condition object is created and managed. These constructors allow the class to be instantiated from a patch and an internal field, from a patch, an internal field, and a dictionary, or by copying an existing object.

The first constructor, as shown in the Listing 4.8, accepts a patch and an internal field as input, allowing for the basic setup of the boundary condition.

Listing 4.8: Declaration of a default constructor in header file

```
241     // Constructors
242
243         //- Construct from patch and internal field
244         coronaryOutletPressureFvPatchScalarField
245         (
246             const fvPatch& patch,
247             const DimensionedField<scalar, volMesh>& field
248         );
```

The second constructor, as shown in the Listing 4.9, provides additional functionality by enabling the initialization of the class with a dictionary, which can contain additional parameters for configuring the boundary condition.

Listing 4.9: Declaration of a dictionary constructor in header file

```
250         //- Construct from patch, internal field, and dictionary
251         coronaryOutletPressureFvPatchScalarField
252         (
```

```
253             const fvPatch& patch,
254             const DimensionedField<scalar, volMesh>& field,
255             const dictionary& dict
256         );
```

Another constructor, as shown in the Listing 4.10, supports the mapping of an existing object onto a new patch, which is useful when the simulation mesh is modified.

Listing 4.10: Declaration of a mapping constructor in header file

```
258         //- Map existing object onto a new patch
259         coronaryOutletPressureFvPatchScalarField
260         (
261             const coronaryOutletPressureFvPatchScalarField& other,
262             const fvPatch& patch,
263             const DimensionedField<scalar, volMesh>& field,
264             const fvPatchFieldMapper& mapper
265         );
```

Additionally, a copy constructor is provided, as shown in the Listing 4.11, allowing the creation of a new object based on an existing instance. The class also includes a method to construct and return a clone, ensuring that an identical copy of the object can be created when needed.

Listing 4.11: Declaration of a copy constructor and a method to construct and return a clone in header file

```
267         //- Copy constructor
268         coronaryOutletPressureFvPatchScalarField
269         (
270             const coronaryOutletPressureFvPatchScalarField& other
271         );
272
273         //- Construct and return a clone
274         virtual tmp<fvPatchScalarField> clone() const
275         {
276             return tmp<fvPatchScalarField>
277             (
278                 new coronaryOutletPressureFvPatchScalarField(*this)
279             );
280         }
```

Furthermore, another constructor, as shown in the Listing 4.12, facilitates the creation of an object by setting a reference to an internal field, ensuring proper data linkage.

Listing 4.12: Declaration of a constructor setting internal field reference in header file

```
282         //- Construct as copy setting internal field reference
283         coronaryOutletPressureFvPatchScalarField
284         (
285             const coronaryOutletPressureFvPatchScalarField& other,
286             const DimensionedField<scalar, volMesh>& field
287         );
288
289         //- Construct and return a clone setting internal field reference
290         virtual tmp<fvPatchScalarField> clone
291         (
292             const DimensionedField<scalar, volMesh>& iF
293         ) const
294         {
295             return tmp<fvPatchScalarField>
296             (
297                 new coronaryOutletPressureFvPatchScalarField(*this, iF)
298             );
299         }
```

4.2.1.10 Member functions

The `coronaryOutletPressureFvPatchScalarField` class contains two important member functions, as shown in the Listing 4.13, that manage the boundary condition's behavior during simulations. These member functions include both evaluation and utility functions. The `updateCoeffs()` function is a key method responsible for updating the coefficients of the boundary condition based on the flow rate. This function ensures that the boundary condition remains consistent with the physical dynamics of the coronary LPN model. Additionally, the class contains the `write()` function, which is responsible for outputting the boundary condition data to an output stream. This functionality is crucial for logging simulation results or exporting data for further analysis. These member functions enable the boundary condition object to interact seamlessly within the OpenFOAM framework, allowing for accurate simulation of outlet pressure dynamics in fluid simulations.

Listing 4.13: Declaration of member functions in header file

```
301     // Member functions
302
303         //- Evaluation function to update coefficients for the boundary
304         //  condition based on flow rate and model parameters
305         virtual void updateCoeffs();
306
307         //- Write boundary condition data to output stream
308         virtual void write(Ostream& os) const;
```

4.2.2 Source file

This implementation defines the `coronaryOutletPressureFvPatchScalarField` boundary condition using various constructors for initialization, mapping and copying, an `updateCoeffs()` function for dynamic updates, and a `write()` function for data output. It supports both first-order and second-order backward differencing schemes to calculate time derivatives of pressures and flow rates for the coronary LPN model and offers flexibility for user customization.

4.2.2.1 Includes and static data members

The `coronaryOutletPressureFvPatchScalarField.C` file includes several essential header files, as shown in the Listing 4.14, to provide necessary functionality and facilitate integration with the OpenFOAM framework.

Listing 4.14: Includes and static data members in source file

```
27 // Include the header file defining the class and related dependencies
28 #include "coronaryOutletPressureFvPatchScalarField.H"
29 #include "addToRunTimeSelectionTable.H" // Macro for runtime selection table
30 #include "fvPatchFieldMapper.H"         // Field mapper utility
31 #include "volFields.H"                  // Volume fields
32 #include "surfaceFields.H"              // Surface fields
33 #include "scalar.H"                     // Scalar type definition
34 #include <fstream>                      // File handling for reading/writing
35 #include <string.h>                    // String manipulation utilities
36 #include "IOdictionary.H"              // Dictionary I/O operations
37 #include "fileName.H"                  // FileName handling
38 #include "OSspecific.H"                // OS-specific operations (e.g., paths)
39 #include <cstdlib>                     // Environment variable handling
40
41
42 // * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * * //
43
44 // (None in this section)
```

The header file, `coronaryOutletPressureFvPatchScalarField.H`, defining the class and its related dependencies is included at the beginning of the implementation file. To facilitate runtime selection table functionality, `addToRunTimeSelectionTable.H` macro is used. Utilities for

field mapping are provided through `fvPatchFieldMapper.H`, while the definitions for volume and surface fields are included via `volFields.H` and `surfaceFields.H`, respectively. Scalar type definitions are introduced through `scalar.H`. File handling capabilities are enabled with the inclusion of `<fstream>`, and string manipulation utilities are made available using `<string.h>`. Dictionary input/output operations are supported by `IOdictionary.H`, while file path handling is managed by `fileName.H`. Moreover, operating system-specific operations are accessed through `OSspecific.H`, and environment variable handling is provided by `<cstdlib>`.

It should be noted that no static data members are defined in this section.

4.2.2.2 Constructors

This section defines multiple constructors including a default constructor, a dictionary constructor, a mapping constructor, and two copy constructors.

A default constructor `coronaryOutletPressureFvPatchScalarField`, as shown in the Listing 4.15, that initializes the object with default values is defined to assign predefined values to the class variables. The constructor accepts a `fvPatch` and a `DimensionedField` as input and then initializes various model parameters. Additionally, it sets the historical pressure and flow rate variables to zero. By default second-order differencing scheme is selected, while the time index is initialized to -1.

Listing 4.15: A default constructor in source file

```
46 // * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * * //
47
48 //- Default constructor: Initializes the class with default values for all parameters.
49 Foam::coronaryOutletPressureFvPatchScalarField::
50 coronaryOutletPressureFvPatchScalarField
51 (
52     const fvPatch& p,                                   // Patch reference
53     const DimensionedField<Foam::scalar, volMesh>& iF // Internal field reference
54 )
55 :
56     fixedValueFvPatchScalarField(p, iF), // Call the base class constructor
57     Ra_(1),                              // Initialize default arterial resistance
58     Ram_(1),                             // Initialize micro-arterial resistance
59     Rv_(1),                              // Initialize venous resistance
60     Rvm_(1),                             // Initialize micro-venous resistance
61     Ca_(1),                              // Initialize arterial compliance
62     Cim_(1),                             // Initialize intramyocardial compliance
63     PimScaling_(1),                      // Initialize scaling factor for intramyocardial pressure
64     Pv_(0),                              // Default distal pressure
65     rho_(1),                             // Default fluid density
66     Qoo_(0), Qo_(0), Qn_(0),             // Initialize flow rate history
67     Poo_(0), Po_(0), Pn_(0),             // Initialize pressure history
68     Pioo_(0), Pio_(0), Pin_(0),          // Initialize intermediate pressure history
69     Pimoo_(0), Pimo_(0), Pimn_(0),       // Initialize intramyocardial pressure history
70     diffScheme_(secondOrder),            // Use second-order differencing by default
71     PimFile_("PimData"),                   //Default name of the Pim data file
72     timeIndex_(-1)                       // Set an invalid time index as the initial state
73 {}
```

A dictionary constructor `coronaryOutletPressureFvPatchScalarField`, as shown in the Listing 4.16, that initializes the object from a dictionary allows for more flexible parameter initialization by extracting values directly from a dictionary, typically used for user input. This constructor accepts a `fvPatch`, `DimensionedField`, and `dictionary` as inputs. It retrieves the values of various LPN model parameters from the dictionary, providing default values if the entries are not found.

Listing 4.16: A dictionary constructor in source file

```
75 //- Constructor from a dictionary, typically used for user input configuration.
76 Foam::coronaryOutletPressureFvPatchScalarField::
77 coronaryOutletPressureFvPatchScalarField
78 (
```

```
    const fvPatch& p,                                   // Patch reference
    const DimensionedField<Foam::scalar, volMesh>& iF, // Internal field reference
    const dictionary& dict                             // Dictionary containing user-provided values
)
:
    fixedValueFvPatchScalarField(p, iF, dict), // Call the base class constructor
    Ra_(dict.lookupOrDefault<Foam::scalar>("Ra", 1)),       // Arterial resistance
    Ram_(dict.lookupOrDefault<Foam::scalar>("Ram", 1)),     // Micro-arterial resistance
    Rv_(dict.lookupOrDefault<Foam::scalar>("Rv", 1)),       // Venous resistance
    Rvm_(dict.lookupOrDefault<Foam::scalar>("Rvm", 1)),     // Micro-venous resistance
    Ca_(dict.lookupOrDefault<Foam::scalar>("Ca", 1)),       // Arterial compliance
    Cim_(dict.lookupOrDefault<Foam::scalar>("Cim", 1)),     // Intramyocardial compliance
    PimScaling_(dict.lookupOrDefault<Foam::scalar>("PimScaling", 1)), // Pim scaling factor
    Pv_(dict.lookupOrDefault<Foam::scalar>("Pv", 0)),       // Distal pressure
    rho_(dict.lookupOrDefault<Foam::scalar>("rho", 1)),     // Fluid density
    Qoo_(dict.lookupOrDefault<Foam::scalar>("Qoo", 0)),     // Historical flow rate values
    Qo_(dict.lookupOrDefault<Foam::scalar>("Qo", 0)),
    Qn_(dict.lookupOrDefault<Foam::scalar>("Qn", 0)),
    Poo_(dict.lookupOrDefault<Foam::scalar>("Poo", 0)),     // Historical pressure values
    Po_(dict.lookupOrDefault<Foam::scalar>("Po", 0)),
    Pn_(dict.lookupOrDefault<Foam::scalar>("Pn", 0)),
    Pioo_(dict.lookupOrDefault<Foam::scalar>("Pioo", 0)),   // Historical intermediate pressure values
    Pio_(dict.lookupOrDefault<Foam::scalar>("Pio", 0)),
    Pin_(dict.lookupOrDefault<Foam::scalar>("Pin", 0)),
    Pimoo_(dict.lookupOrDefault<Foam::scalar>("Pimoo", 0)), // Historical intramyocardial pressure
    Pimo_(dict.lookupOrDefault<Foam::scalar>("Pimo", 0)),
    Pimn_(dict.lookupOrDefault<Foam::scalar>("Pimn", 0)),
    timeIndex_(-1) // Initialize time index
```

Moreover, the dictionary constructor maps the selected differencing scheme (e.g., `firstOrder` or `secondOrder`) from the dictionary and prints the relevant information for debugging and verification purposes, as shown in the Listing 4.17.

Listing 4.17: Extracting differencing shceme from dictionary within constructor in source file

```
    Info << "\n\nApplying coronaryOutletPressure BC on patch: " << patch().name() << endl;

    //- Extract differencing scheme from dictionary and validate
    word schemeStr = dict.lookupOrDefault<word>("diffScheme", "secondOrder");

    Info << "Differencing Scheme for Coronary LPN Model: " << schemeStr << endl;

    if (schemeStr == "firstOrder")
    {
        diffScheme_ = firstOrder; // First-order scheme
    }
    else if (schemeStr == "secondOrder")
    {
        diffScheme_ = secondOrder; // Second-order scheme
    }
    else
    {
        FatalErrorInFunction << "\n\nUnknown Differencing Scheme: " << schemeStr
                             << "\nValid Differencing Schemes (diffScheme) are : \n\n"
                             << " 2 \n ( \n firstOrder \n secondOrder \n ) \n"
                             << exit(FatalError);
    }
```

Next, the dictionary constructor lookup the path to intramyocardial pressure time-series data file specified in the dictionary, and invoke the `loadPimData` function to load the time-series data and store it in the `PimData_` vector, as shown in the Listing 4.18. After retrieving the values, it prints relevant information about the LPN model properties for debugging and verification purposes.

Listing 4.18: Extracting intramyocardial pressure data and printing LPN properties within constructor in source file

```
    //- Extract file name for intramyocardial pressure (Pim) data
```

```
    word PimFileStr = dict.lookupOrDefault<fileName>("PimFile", "PimData");

    PimFile_ = PimFileStr;

    //- Load the Pim data from the specified file
    loadPimData(PimFile_);

    //- Print coronary LPN model properties for debugging/verification
    Info << "\n\nProperties of Coronary LPN Model: \n"
            << "Differencig Scheme: " << diffScheme_ << " \n"
            << "Arterial Resistance: Ra = " << Ra_ << " kgm^-4s^-1 \n"
            << "Micro-arterial Resistance: Ram = " << Ram_ << " kgm^-4s^-1 \n"
            << "Veinous Resistance: Rv = " << Rv_ << " kgm^-4s^-1 \n"
            << "Micro-veinous Resistance: Rvm = " << Rvm_ << " kgm^-4s^-1 \n"
            << "Arterial Compliance: Ca = " << Ca_ << " m^4s^2kg^-1 \n"
            << "Intramyocardial Compliance: Cim = " << Cim_ << " m^4s^2kg^-1 \n"
            << "Path to Pim Data File: " << PimFile_ << " \n"
            << "Pim Scaling Factor: PimScaling: " << PimScaling_ << " \n"
            << "Pressure Distal to Rv: Pv = " << Pv_ << " Pa \n"
            << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
```

A mapping constructor `coronaryOutletPressureFvPatchScalarField`, as shown in the Listing 4.19, that maps an existing field onto a new patch facilitates the copying of the boundary condition from one patch to another, with an optional field mapping operation. This constructor accepts an existing `coronaryOutletPressureFvPatchScalarField` object, a new `fvPatch`, a `DimensionedField`, and a `fvPatchFieldMapper`. It copies the relevant properties, such as LPN model parameters, historical pressure and flow rate variables, and intramydocardial pressure data. The constructor allows the boundary condition to be reassigned to a new patch, ensuring that all the properties are appropriately transferred.

Listing 4.19: A mapping constructor in source file

```
//- Constructor: Map an existing field onto a new patch
Foam::coronaryOutletPressureFvPatchScalarField::
coronaryOutletPressureFvPatchScalarField
(
    const coronaryOutletPressureFvPatchScalarField& ptf, // Existing field to map from
    const fvPatch& p,                                    // New patch
    const DimensionedField<Foam::scalar, volMesh>& iF,  // Internal field
    const fvPatchFieldMapper& mapper                     // Mapper for field data
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper),    // Call base class mapping constructor
    Ra_(ptf.Ra_),                                        // Copy resistance parameter Ra
    Ram_(ptf.Ram_),                                      // Copy modified resistance Ram
    Rv_(ptf.Rv_),                                        // Copy venous resistance Rv
    Rvm_(ptf.Rvm_),                                      // Copy modified venous resistance Rvm
    Ca_(ptf.Ca_),                                        // Copy arterial compliance Ca
    Cim_(ptf.Cim_),                                      // Copy impedance compliance Cim
    PimScaling_(ptf.PimScaling_),                        // Copy scaling for Pim values
    Pv_(ptf.Pv_),                                        // Copy distal pressure Pv
    rho_(ptf.rho_),                                      // Copy density rho
    Qoo_(ptf.Qoo_), Qo_(ptf.Qo_), Qn_(ptf.Qn_),          // Copy flow rates Qoo, Qo, Qn
    Poo_(ptf.Poo_), Po_(ptf.Po_), Pn_(ptf.Pn_),          // Copy pressures Poo, Po, Pn
    Pioo_(ptf.Pioo_), Pio_(ptf.Pio_), Pin_(ptf.Pin_),    // Copy intermediate pressures Pioo, Pio, Pin
    Pimoo_(ptf.Pimoo_), Pimo_(ptf.Pimo_), Pimn_(ptf.Pimn_), // Copy Pim intermediate pressures
    diffScheme_(ptf.diffScheme_),                        // Copy differencing scheme
    PimFile_(ptf.PimFile_),                              // Copy Pim data file path
    PimData_(ptf.PimData_),                              // Copy Pim data storage
    timeIndex_(ptf.timeIndex_)                          // Copy current time index
{}
```

Two variations of copy constructors are defined. The first copy constructor, as shown in the Listing 4.20, of the `coronaryOutletPressureFvPatchScalarField` class is used to create a duplicate field from an existing instance. An existing `coronaryOutletPressureFvPatchScalarField` object is accepted as input and all its properties, including LPN model parameters, pressure and flow rate

histories, etc. are copied. It is essentially a direct duplication of the object's state, enabling the creation of identical copies when necessary.

Listing 4.20: A copy constructor in source file

```
184 //- Copy constructor: Create a copy of an existing field
185 Foam::coronaryOutletPressureFvPatchScalarField::
186 coronaryOutletPressureFvPatchScalarField
187 (
188     const coronaryOutletPressureFvPatchScalarField& copsf // Field to copy
189 )
190 :
191     fixedValueFvPatchScalarField(copsf),              // Call base class copy constructor
192     Ra_(copsf.Ra_),                                   // Copy resistance parameter Ra
193     Ram_(copsf.Ram_),                                 // Copy modified resistance Ram
194     Rv_(copsf.Rv_),                                   // Copy venous resistance Rv
195     Rvm_(copsf.Rvm_),                                 // Copy modified venous resistance Rvm
196     Ca_(copsf.Ca_),                                   // Copy arterial compliance Ca
197     Cim_(copsf.Cim_),                                 // Copy impedance compliance Cim
198     PimScaling_(copsf.PimScaling_),                   // Copy scaling for Pim values
199     Pv_(copsf.Pv_),                                   // Copy distal pressure Pv
200     rho_(copsf.rho_),                                 // Copy density rho
201     Qoo_(copsf.Qoo_), Qo_(copsf.Qo_), Qn_(copsf.Qn_),  // Copy flow rates Qoo, Qo, Qn
202     Poo_(copsf.Poo_), Po_(copsf.Po_), Pn_(copsf.Pn_),  // Copy pressures Poo, Po, Pn
203     Pioo_(copsf.Pioo_), Pio_(copsf.Pio_), Pin_(copsf.Pin_), // Copy intermediate pressures
204     Pimoo_(copsf.Pimoo_), Pimo_(copsf.Pimo_), Pimn_(copsf.Pimn_), // Copy Pim intermediate pressures
205     diffScheme_(copsf.diffScheme_),                    // Copy differencing scheme
206     PimFile_(copsf.PimFile_),                          // Copy Pim data file path
207     PimData_(copsf.PimData_),                          // Copy Pim data storage
208     timeIndex_(copsf.timeIndex_)                       // Copy current time index
209 {}
```

Another copy constructor with a new internal field reference, as shown in the Listing 4.21, allows the creation of a duplicate `coronaryOutletPressureFvPatchScalarField` object, but with a different internal field reference. It takes an existing `coronaryOutletPressureFvPatchScalarField` object and a new `DimensionedField` as input. While it copies all the class properties, it ensures that the internal field reference is updated, making it suitable for scenarios where the internal field needs to be modified or reassigned without altering other properties of the object.

Listing 4.21: Second copy constructor in source file

```
211 //- Copy constructor with new internal field reference
212 Foam::coronaryOutletPressureFvPatchScalarField::
213 coronaryOutletPressureFvPatchScalarField
214 (
215     const coronaryOutletPressureFvPatchScalarField& copsf, // Field to copy
216     const DimensionedField<scalar, volMesh>& iF            // New internal field reference
217 )
218 :
219     fixedValueFvPatchScalarField(copsf, iF),          // Call base class copy with new field
220     Ra_(copsf.Ra_),                                   // Copy resistance parameter Ra
221     Ram_(copsf.Ram_),                                 // Copy modified resistance Ram
222     Rv_(copsf.Rv_),                                   // Copy venous resistance Rv
223     Rvm_(copsf.Rvm_),                                 // Copy modified venous resistance Rvm
224     Ca_(copsf.Ca_),                                   // Copy arterial compliance Ca
225     Cim_(copsf.Cim_),                                 // Copy impedance compliance Cim
226     PimScaling_(copsf.PimScaling_),                   // Copy scaling for Pim values
227     Pv_(copsf.Pv_),                                   // Copy distal pressure Pv
228     rho_(copsf.rho_),                                 // Copy density rho
229     Qoo_(copsf.Qoo_), Qo_(copsf.Qo_), Qn_(copsf.Qn_),  // Copy flow rates Qoo, Qo, Qn
230     Poo_(copsf.Poo_), Po_(copsf.Po_), Pn_(copsf.Pn_),  // Copy pressures Poo, Po, Pn
231     Pioo_(copsf.Pioo_), Pio_(copsf.Pio_), Pin_(copsf.Pin_), // Copy intermediate pressures
232     Pimoo_(copsf.Pimoo_), Pimo_(copsf.Pimo_), Pimn_(copsf.Pimn_), // Copy Pim intermediate pressures
233     diffScheme_(copsf.diffScheme_),                   // Copy differencing scheme
234     PimFile_(copsf.PimFile_),                         // Copy Pim data file path
235     PimData_(copsf.PimData_),                         // Copy Pim data storage
236     timeIndex_(copsf.timeIndex_)                     // Copy current time index
```

```
{}
```

4.2.2.3 Member functions

There are two important member functions in the source file including `updateCoeffs()` which updates the pressure and `write()` which writes the current state of the boundary condition to an output stream. The `updateCoeffs()` function uses a helper function `interpolatePim` to get the value of intramyocardial pressure at each time-step. The `interpolatePim` in turns uses the vector `PimData_` populated by invoking `loadPimData` function to load the data from the file provided by the user at the specified path during the initialization. If the file path contains any environment variables, it uses `expandEnvironmentVariables` helper function to expand those.

The `updateCoeffs()` function, as shown in the Listing 4.22, is a member function of the `Foam::coronaryOutletPressureFvPatchScalarField` class, responsible for updating the coefficients associated with the boundary condition for a coronary LPN model in the CFD simulations. The function checks if the coefficients have already been updated for the current time step and performs necessary calculations to update pressure and flow rate histories as required by the LPN model. It handles different differencing schemes (`firstOrder`, `secondOrder`) to compute pressure values based on flow rates, pressures, and other parameters.

Listing 4.22: `updateCoeffs()` member function in source file

```
// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * * //

//- Update coefficients for boundary condition
void Foam::coronaryOutletPressureFvPatchScalarField::updateCoeffs()
```

Initially, the function checks if the coefficients have been updated by calling the `updated()` method, as shown in the Listing 4.23. If the coefficients have already been updated for the current time step, the function returns early, avoiding redundant calculations. If not, it proceeds to retrieve essential data from the simulation database, including the time step size (`dt`), current time (`currentTime`), flux field (`phi`), and boundary pressure field (`p`). The total area of the patch is then calculated using the `patch().magSf()` method, which returns the magnitude of the surface vector of the patch.

Listing 4.23: Checking the coefficients update and the retrieval of essential data from the simulation database within `updateCoeffs()` function

```
    //- Check if coefficients are already updated
    if (updated())
    {
        return;
    }

    //- Get time step size and current simulation time
    const scalar dt = db().time().deltaTValue(); // Time step size
    const scalar currentTime = db().time().value(); // Current time

    //- Retrieve the flux field (phi) from the database
    const surfaceScalarField& phi =
        db().lookupObject<surfaceScalarField>("phi");

    //- Retrieve the boundary pressure field from the database
    const fvPatchField<scalar>& p =
        db().lookupObject<volScalarField>("p").boundaryField()[this->patch().index()];

    //- Calculate total patch area
    scalar area = gSum(patch().magSf());
```

Next, the pressure and flow rate histories are updated if the time index has advanced, as shown in the Listing 4.24. This involves shifting the historical flow rate and pressure values (e.g., `Qoo_`, `Qo_`, `Qn_`, `Poo_`, `Po_`, `Pn_`, `Pimoo_`, `Pimo_`, `Pimn_` and `Pioo_`, `Pio_`, `Pin_`) to make space for the

latest values. The current value of flow rate is obtained by summing the flux through the patch. The current value of pressure is computed using area-weighted average, this value will be updated next in the time loop. The value of the intramyocardial pressure at current time is obtained by invoking `interpolatePim` helper function.

Listing 4.24: Updating pressure and flow rate histories within `updateCoeffs()` function

```
265     //- Update pressure and flow rate histories if time index has advanced
266     if (db().time().timeIndex() != timeIndex_)
267     {
268         timeIndex_ = db().time().timeIndex(); // Update time index
269 
270         //- Update flow rate history
271         Qoo_ = Qo_;
272         Qo_ = Qn_;
273         Qn_ = gSum(phi.boundaryField()[patch().index()]); // Compute current flow rate
274 
275         //- Update pressure history
276         Poo_ = Po_;
277         Po_ = Pn_;
278         Pn_ = gSum(p * patch().magSf()) / area; // Compute current pressure (area-weighted average)
279 
280         //- Update Pim history
281         Pimoo_ = Pimo_;
282         Pimo_ = Pimn_;
283         Pimn_ = interpolatePim(currentTime); // Interpolate to compute current Pim
284 
285         //- Update intermediate pressure history
286         Pioo_ = Pio_;
287         Pio_ = Pin_;
288     }
```

After updating the history, the function proceeds to calculate the new pressure value (`Pn_`), as shown in the Listing 4.25. The pressure update depends on the selected differencing scheme. In both cases, the value of intermediate pressure `Pin_` is computed first which is then used to compute the pressure for the coronary outlet boundary `Pn_` using the values of LPN model parameters, flow rate (`Qn_`), intramyocardial pressure `Pimn_` and their change over time. If an unknown differencing scheme is encountered, an error is raised using `FatalErrorInFunction`.

Listing 4.25: Calculating the new pressure value (`Pn_`) within `updateCoeffs()` function

```
290     //- Perform calculations only if time index exceeds 1
291     if (db().time().timeIndex() > 1)
292     {
293         //- Choose differencing scheme
294         switch (diffScheme_)
295         {
296             case firstOrder: // First-order scheme
297 
298                 //- Compute intermediate pressure at current time
299                 Pin_ = (Rvm_ + Rv_) * Qn_
300                     - (Rvm_ + Rv_) * Ca_ * ((Pn_ - Po_) / dt)
301                     + (Rvm_ + Rv_) * Cim_ * ((Pimn_ - Pimo_) / dt)
302                     + Pv_
303                     + (Rvm_ + Rv_) * Cim_ * Pio_ / dt;
304 
305                 Pin_ /= (1.0 + (Rvm_ + Rv_) * Cim_ / dt) + SMALL;
306 
307                 //- Compute boundary pressure at current time
308                 Pn_ = (Ram_ + Ra_) * Qn_
309                     + Ram_ * Ra_ * Ca_ * ((Qn_ - Qo_) / dt)
310                     + Pin_
311                     + Ram_ * Ca_ * (Po_ / dt);
312 
313                 Pn_ /= (1.0 + Ram_ * Ca_ / dt) + SMALL;
314 
315                 break;
```

```
316
317             case secondOrder: // Second-order scheme
318
319                 //- Compute intermediate pressure at current time
320                 Pin_ = (Rvm_ + Rv_) * Qn_
321                     - (Rvm_ + Rv_) * Ca_ * ((3 * Pn_ - 4 * Po_ + Poo_) / (2 * dt))
322                     + (Rvm_ + Rv_) * Cim_ * ((3 * Pimn_ - 4 * Pimo_ + Pimoo_) / (2 * dt))
323                     + Pv_
324                     - (Rvm_ + Rv_) * Cim_ * (Pioo_ - 4 * Pio_) / (2 * dt);
325
326                 Pin_ /= (1.0 + 3 * (Rvm_ + Rv_) * Cim_ /(2 * dt)) + SMALL;
327
328                 //- Compute boundary pressure at current time
329                 Pn_ = (Ram_ + Ra_) * Qn_
330                     + Ram_ * Ra_ * Ca_ * ((3 * Qn_ - 4 * Qo_ + Qoo_) / (2 * dt))
331                     + Pin_
332                     - Ram_ * Ca_ * (Poo_ - 4 * Po_) / (2 * dt);
333
334                 Pn_ /= (1.0 + 3 * Ram_ * Ca_ / (2 * dt)) + SMALL;
335
336                 break;
337
338             default:
339                 //- Handle unknown schemes
340                 FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
341         }
342     }
```

Finally, the pressure update is applied to the boundary field, as shown in the Listing 4.26. The pressure (`Pn_`) is directly updated using the formula `Pn_ / (rho_ + SMALL)`. The function then calls the base class's `updateCoeffs()` method to finalize the update.

Listing 4.26: Applying the pressure to the boundary field within `updateCoeffs()` function

```
244     //- Apply implicit update to the boundary field
245     operator==(Pn_ / (rho_ + SMALL));
246
247     //- Call base class function to finalize the update
248     fixedValueFvPatchScalarField::updateCoeffs();
```

The `updateCoeffs()` function ensures that the boundary conditions related to the coronary LPN model are correctly updated for each time step, accounting for all the parameters involved in its differential equations and differencing schemes. The function is critical in maintaining the accuracy and stability of the simulation by applying the coronary LPN model calculations based on the flow rates and other parameters.

The interpolation of intramyocardial pressure is handled by `interpolatePim` helper function, as shown in the Listing 4.27. It is designed to compute the interpolated value of intramyocardial pressure `Pimn_` at current time. The interpolation is achieved using a dataset, `PimData_`, which must be pre-populated with time-pressure pairs.

Listing 4.27: `interpolatePim` helper function in source file

```
351 // Helper methods
352
353 //- Interpolates the intramyocardial pressure (Pim) at a given time
354 Foam::scalar Foam::coronaryOutletPressureFvPatchScalarField::interpolatePim(const scalar& time) const
```

If `PimData_` is empty, an error is triggered, terminating the operation, as shown in the Listing 4.28.

Listing 4.28: Ensuring `PimData_` is populated within `interpolatePim` helper function

```
356     //- Ensure PimData_ is populated
357     if (PimData_.empty())
358     {
359         FatalErrorInFunction << "PimData_ is empty. Cannot interpolate!" << exit(FatalError);
```

```
360     }
```

Within this function, the start and end times of the dataset are extracted. The input time is adjusted using modulo arithmetic to ensure periodicity, and corrections are applied for negative modulo results, as shown in the Listing 4.29.

Listing 4.29: Calculating the effective time within `interpolatePim` helper function

```
362     //- Get the start and end times of the Pim data
363     const scalar tStart = PimData_.front().first; // Start time of the dataset
364     const scalar tEnd = PimData_.back().first;   // End time of the dataset
365
366     //- Calculate the effective time using modulo to handle periodicity
367     scalar effectiveTime = tStart + std::fmod(time - tStart, tEnd - tStart);
368     if (effectiveTime < tStart)
369     {
370         effectiveTime += (tEnd - tStart); // Adjust for negative modulo results
371     }
```

Linear interpolation is performed by iterating over the dataset to locate the appropriate interval containing the adjusted time, as shown in the Listing 4.30. The corresponding pressure value is calculated by scaling the interpolated pressure with scaling factor `PimScaling_` specified by the user. If no valid interval is found, the last pressure value in the dataset is returned as a fallback. The fallback should never be reaches, as the algorithm have been implemented to repeat the dataset after each cycle. This ensures that a value is always provided, even though the fallback should rarely be needed.

Listing 4.30: Performing linear interpolation to find the corresponding intramyocardial pressure value within `interpolatePim` helper function

```
373     //- Perform linear interpolation to find the corresponding Pim value
374     for (size_t i = 1; i < PimData_.size(); ++i)
375     {
376         if (PimData_[i - 1].first <= effectiveTime && PimData_[i].first >= effectiveTime)
377         {
378             scalar t1 = PimData_[i - 1].first;      // Previous time point
379             scalar t2 = PimData_[i].first;          // Current time point
380             scalar Pim1 = PimData_[i - 1].second;   // Pim value at previous time
381             scalar Pim2 = PimData_[i].second;       // Pim value at current time
382
383             //- Linear interpolation formula
384             return PimScaling_ * (Pim1 + (Pim2 - Pim1) * (effectiveTime - t1) / (t2 - t1));
385         }
386     }
387
388     //- Fallback case: return the last Pim value (should not typically be reached)
389     return PimData_.back().second;
```

The `interpolatePim` helper function uses the dataset contained in the vector `PimData_`, populated during the initialization using `loadPimData` helper function, as shown in the Listing 4.31. It is done by reading a file containing time-pressure pairs.

Listing 4.31: `loadPimData` helper function in source file

```
392 //- Loads intramyocardial pressure data (PimData_) from a file
393 void Foam::coronaryOutletPressureFvPatchScalarField::loadPimData(const word& fileName)
```

Any existing data in `PimData_` is cleared before new data is loaded. The file name provided as input is processed to expand any environment variables that might be present in the file path using another helper function `expandEnvironmentVariables`. An attempt is then made to open the corresponding file, as shown in the Listing 4.32. If the file cannot be opened, an error is raised, halting execution.

The file content is read line by line, and each line is processed to remove parentheses and check for emptiness. Valid lines are parsed into time and pressure values, which are stored as pairs in

`PimData_`, as shown in the Listing 4.33. The file is closed upon completion, ensuring that resources are released.

Listing 4.32: Opening the file containing intramyocardial pressure data within `loadPimData` helper function

```
395     //- Clear existing data
396     PimData_.clear();
397 
398     //- Expand environment variables in the file name
399     std::string expandedFileName = expandEnvironmentVariables(fileName);
400 
401     //- Open the file
402     std::ifstream file(expandedFileName.c_str());
403 
404     //- Check if the file was successfully opened
405     if (!file.is_open())
406     {
407         FatalErrorInFunction << "Cannot open intramyocardial pressure file: " << fileName << exit(
        FatalError);
408     }
```

Listing 4.33: Reading the file containing intramyocardial pressure data within `loadPimData` helper function

```
410     std::string line;
411     while (std::getline(file, line)) // Read the file line by line
412     {
413         //- Remove parentheses from the line
414         line.erase(std::remove(line.begin(), line.end(), '('), line.end());
415         line.erase(std::remove(line.begin(), line.end(), ')'), line.end());
416 
417         //- Skip empty or invalid lines
418         if (line.empty()) continue;
419 
420         //- Parse the line into time and Pim values
421         std::istringstream iss(line);
422         scalar time, Pim;
423         if (iss >> time >> Pim) // If parsing is successful
424         {
425             PimData_.emplace_back(time, Pim); // Store the time and Pim values
426         }
427     }
428 
429     file.close(); // Close the file
```

The function `expandEnvironmentVariables`, shown in the Listing 4.34, is used by `loadPimData` function in order to replace any environment variable patterns in a string with their corresponding values.

Listing 4.34: `expandEnvironmentVariables` helper function in source file

```
432 //- Expands environment variables in the given string
433 std::string Foam::coronaryOutletPressureFvPatchScalarField::expandEnvironmentVariables(const std::
     string& input)
434 {
435     std::string result = input; // Start with the input string
436     size_t pos = 0;
437 
438     //- Look for occurrences of "$" followed by an alphabetic character (environment variable pattern)
439     while ((pos = result.find("$", pos)) != std::string::npos)
440     {
441         size_t endPos = result.find_first_not_of("ABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789", pos + 1);
442         if (endPos == std::string::npos)
443         {
444             //- No valid environment variable found after "$"
445             break;
```

```
        }

        //- Extract the environment variable name
        std::string varName = result.substr(pos + 1, endPos - pos - 1);

        //- Get the environment variable value from the system
        const char* envValue = std::getenv(varName.c_str());

        if (envValue) // If the environment variable exists
        {
            //- Replace the variable with its value
            result.replace(pos, endPos - pos, envValue);
            pos += std::string(envValue).size(); // Move past the replaced value
        }
        else
        {
            //- If the environment variable is not found, skip to the next "$"
            pos = endPos;
        }
    }
    return result; // Return the string with expanded variables
}
```

The process of expansion for the input string begins with the occurrences of the dollar sign ($) followed by the identification of an alphanumeric or underscore character. For each occurrence of the dollar sign ($) in the file path, the variable name is extracted, and its value is retrieved using the system environment. If the variable exists, it is replaced in the input string with its value. If no corresponding environment variable is found, the occurrence is skipped. The modified string, containing the expanded values, is returned. This ensures that environment variables in file paths or other inputs are correctly resolved.

After the pressure update is applied, `write()` function, shown in the Listing 4.35, is invoked. It is a member of the `Foam::coronaryOutletPressureFvPatchScalarField` class, which is responsible for writing the boundary condition properties of the coronary LPN model boundary condition to an output stream. The function gathers various parameters that define the model and its configuration, such as resistance values, compliance values, pressure values, and outputs them to the specified stream. This function ensures that all relevant model details, including history parameters and the chosen differencing scheme, are captured and written to the output for further analysis or post-processing.

Listing 4.35: `write()` member function in source file

```
//- Write the boundary condition properties to an output stream
void Foam::coronaryOutletPressureFvPatchScalarField::write(Ostream& os) const
```

At the beginning of the function, the base class function `fvPatchScalarField::write(os)` is called to write common boundary field properties which includes all the LPN model parameters, as shown in the Listing 4.36. This serves as a starting point to ensure that all necessary boundary-related information is included in the output.

Listing 4.36: Calling the base class function and writing LPN model parameters to an output stream within `write()` function

```
    //- Call base class method to write common properties
    fvPatchScalarField::write(os);

    //- Write each parameter with its keyword and value
    os.writeKeyword("Ra") << Ra_ << token::END_STATEMENT << nl;
    os.writeKeyword("Ram") << Ram_ << token::END_STATEMENT << nl;
    os.writeKeyword("Rv") << Rv_ << token::END_STATEMENT << nl;
    os.writeKeyword("Rvm") << Rvm_ << token::END_STATEMENT << nl;
    os.writeKeyword("Ca") << Ca_ << token::END_STATEMENT << nl;
    os.writeKeyword("Cim") << Cim_ << token::END_STATEMENT << nl;
    os.writeKeyword("PimFile") << PimFile_ << token::END_STATEMENT << nl;
    os.writeKeyword("PimScaling") << PimScaling_ << token::END_STATEMENT << nl;
```

```
484     os.writeKeyword("Pv") << Pv_ << token::END_STATEMENT << nl;
485     os.writeKeyword("rho") << rho_ << token::END_STATEMENT << nl;
```

Next, the function writes the differencing scheme used for calculations. The `diffScheme_` variable is checked, and the corresponding scheme—either `firstOrder` or `secondOrder`—is written to the output, as shown in the Listing 4.37. If an unknown differencing scheme is encountered, an error is raised.

Listing 4.37: Writing the differencing scheme to an output stream within `write()` function

```
487     //- Write the differencing scheme
488     switch (diffScheme_)
489     {
490         case firstOrder:
491             os.writeKeyword("diffScheme") << "firstOrder" << token::END_STATEMENT << nl;
492             break;
493         case secondOrder:
494             os.writeKeyword("diffScheme") << "secondOrder" << token::END_STATEMENT << nl;
495             break;
496         default:
497             FatalErrorInFunction << "Unknown differencing scheme: " << diffScheme_ << exit(FatalError)
      ;
498     }
```

Next, the function writes the history of flow rates. This includes the second past flow rate (`Qoo_`), the previous flow rate (`Qo_`), and the current flow rate (`Qn_`), as shown in the Listing 4.38. Similarly, the function writes the pressure history. The parameters include the second past, the previous, and the current values for intramyocardial pressure, intermediate pressure value, and coronary outlet pressure. These flow rate and pressure histories are necessary for a perfect restart of the simulation. Finally, the function calls the `writeEntry()` function to write the boundary field value entry to the output stream. This ensures that the boundary field value is included in the output along with the model parameters.

Listing 4.38: Writing flow rate and pressure histories and boundary field entry within `write()` function

```
500     //- Write flow history parameters
501     os.writeKeyword("Qoo") << Qoo_ << token::END_STATEMENT << nl;
502     os.writeKeyword("Qo") << Qo_ << token::END_STATEMENT << nl;
503     os.writeKeyword("Qn") << Qn_ << token::END_STATEMENT << nl;
504
505     //- Write pressure history parameters
506     os.writeKeyword("Pimoo") << Pimoo_ << token::END_STATEMENT << nl;
507     os.writeKeyword("Pimo") << Pimo_ << token::END_STATEMENT << nl;
508     os.writeKeyword("Pimn") << Pimn_ << token::END_STATEMENT << nl;
509     os.writeKeyword("Pioo") << Pioo_ << token::END_STATEMENT << nl;
510     os.writeKeyword("Pio") << Pio_ << token::END_STATEMENT << nl;
511     os.writeKeyword("Pin") << Pin_ << token::END_STATEMENT << nl;
512     os.writeKeyword("Poo") << Poo_ << token::END_STATEMENT << nl;
513     os.writeKeyword("Po") << Po_ << token::END_STATEMENT << nl;
514     os.writeKeyword("Pn") << Pn_ << token::END_STATEMENT << nl;
515
516     //- Write the boundary field value entry to the output stream
517     writeEntry("value", os);
```

The `write()` function serves to export the essential properties and configuration details of the coronary LPN model, along with its flow and pressure history, to an output stream. It ensures that all necessary information for later analysis is made available in a standardized format. This is crucial for simulations where boundary conditions and their histories play a significant role in the overall model behavior.

4.2.2.4 Runtime type selection

The final part of the source file, as shown in Listing 4.39, defines the boundary field type for the `coronaryOutletPressureFvPatchScalarField` class within the `Foam` namespace.

Listing 4.39: Runtime type selection in source file

```
522 //- Registration
523 namespace Foam
524 {
525     //- Define the boundary field type for this class
526     makePatchTypeField
527     (
528         fvPatchScalarField,                       // Base class
529         coronaryOutletPressureFvPatchScalarField  // Derived class
530     );
531 }
```

The `makePatchTypeField` macro is used to establish the type of the boundary field, associating the class `fvPatchScalarField` with `coronaryOutletPressureFvPatchScalarField`. This step is crucial as it registers the specific boundary field type, enabling it to be used within the broader framework of the code.

This declaration ensures that the `coronaryOutletPressureFvPatchScalarField` class will be recognized as a valid type for boundary fields in simulations, particularly for those involving coronary LPN model. The macro call effectively links the class with the appropriate functionalities of the `fvPatchScalarField`, allowing for seamless integration within the computational framework.

Chapter 5

Test Cases

5.1 Description

A comprehensive test simulation with a simple cylindrical geometry is prepared to test each of variations of the boundary condition. This is done by preparing a master test case, which is then used to create a directory structure for each of the simulation cases through a bash script. And then all the cases are run one by one. The initial directory structure tree of the test case folder `LumpedParameterNetworkBCTestCase/` containing LPN BC directory `lumpedParameterNetworkBC/`, master test case `LPNBCTestCase/`, their relevant bash scripts for running and cleaning, and main bash scripts for running or cleaning all the cases as well as creating and deleting directory structure is shown in Figure 5.1.

```
LumpedParameterNetworkBCTestCase/
├── Allclean, Allrun, CreateDirStruc, DeleteDirStruc
├── LPNBCTestCase/
│   ├── 0.orig/
│   │   ├── U.{Coronary, WK}
│   │   ├── p.{Resistive}
│   │   └── p.{WK2, WK3, WK4Parallel, WK4Series, Coronary}.{firstOrder,
│   │       secondOrder}
│   ├── Allclean.{Coronary, WK}
│   ├── Allrun.{Coronary, WK}
│   ├── DataFiles/
│   │   └── AorticInletFlowRate, CoronaryInletFlowRate, PimData
│   ├── constant/
│   │   ├── RASProperties, dynamicMeshDict, fluidProperties,
│   │   │   physicsProperties, transportProperties, turbulenceProperties
│   │   └── polyMesh/
│   │       └── blockMeshDict.m4.{Coronary, WK}
│   └── system/
│       └── controlDict, decomposeParDict, fvSchemes, fvSolution
├── MATLABPostprocessing/
│   └── LPNBCTestCasePostprocessing.m
└── lumpedParameterNetworkBC/
    ├── Allwclean, Allwmake
    ├── Make/
    │   └── files, options
    ├── coronaryOutletPressure/
    │   └── coronaryOutletPressureFvPatchScalarField.{C, H}
    └── windkesselOutletPressure/
        └── windkesselOutletPressureFvPatchScalarField.{C, H}
```

Figure 5.1: Initial directory structure of `LumpedParameterNetworkBCTestCase`

To run the test cases, first a complete directory structure is created by running `CreateDirStruc` bash script, then the main `Allrun` bash script is run which in turn runs the `Allwmake` script in the LPN BC directory, and `Allrun` scripts in all case directories. It should be noted that `foam-extend-4.1` with a working installation of `solids4foam-v2.1` is used to run all the cases. The `windkesselOutletPressure` BC is tested using straight cylindrical geometry of aorta, while `coronaryOutletPressure` BC is tested using straight cylindrical geometry of left coronary artery (LCA).

5.2 Geometry and mesh

A `.m4` script is used to create simple straight cylindrical geometries for both aorta and left coronary artery (LCA). The diameter of $D = 30\ mm$ is used for aorta while the diameter of $D = 3\ mm$ is used for LCA. The length of both domains is kept equal to four diameters, $L = 4D$. For both cases, 60 cells are defined along the length. The values of control parameters used to create mesh for aorta are shown in the Listing 5.1, while those used for LCA are shown in the Listing 5.2.

Listing 5.1: Control paremeters in `.m4` script to create mesh for aorta

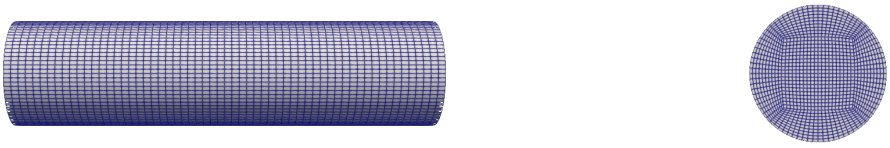
```
28 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
29
30 convertToMeters 0.001;
31
32 // Inner square side half
33 define(s, 7.50)
34
35 // Inner square side curvature
36 define(sc, 8)
37
38 // cylinder radius
39 define(r, 15)
40
41 // Height of cylinder
42 define(z, 120.0)
43
44 // Base z
45 define(Zb, 0)
46
47 // Outlet z
48 define(Zt, calc(Zb + z))
49
50 // Number of cells at inner square
51 define(Ns, 20)
52
53 // Number of cells between inner square and circle
54 define(Ni, 10)
55
56 // Number of cells in the cylinder height
57 define(Nz, 60)
58
59 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Listing 5.2: Control paremeters in `.m4` script to create mesh for LCA

```
28 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
29
30 convertToMeters 0.001;
31
32 // Inner square side half
33 define(s, 0.750)
34
35 // Inner square side curvature
36 define(sc, 0.8)
37
38 // cylinder radius
```

```
define(r, 1.50)

// Height of cylinder
define(z, 12.0)

// Base z
define(Zb, 0)

// Outlet z
define(Zt, calc(Zb + z))

// Number of cells at inner square
define(Ns, 20)

// Number of cells between inner square and circle
define(Ni, 10)

// Number of cells in the cylinder height
define(Nz, 60)

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Both the resulting meshes are same, as shown in Figure 5.2, consisting of 72000 hexahedra cells with maximum aspect ratio of 7.05029367, maximum non-orthogonality of 29.075478, and maximum skewness of 1.01487831, and a difference only in the scale or size of the domain.



(a) Mesh along the length

(b) Mesh on inlet/outlet patches

Figure 5.2: Mesh on straight cylindrical geometry representing aorta and LCA

5.3 Boundary conditions: the 0 directory

5.3.1 U boundary field

For both, aorta and LCA, `timeVaryingFlowRateInletVelocity` boundary condition available in `foam-extend-4.1` as well as `solids4foam-v2.1` is used to specify the inlet volume flow rate, as shown in Listing 5.3 for aorta and Listing 5.5 for LCA respectively from the files, placed in the `DataFiles` directory containing the time-series data for the volume flow rate generated for either aorta or coronary artery.

Listing 5.3: `U` field boundary condition at `inlet` patch for aorta

```
    inlet
    {
        type                timeVaryingFlowRateInletVelocity;
        flowRate            0;              // Volumetric/mass flow rate [m3/s or kg/s]
        value               uniform (0 0 0); // placeholder
        "file|fileName"     "$FOAM_CASE/DataFiles/AorticInletFlowRate";
        outOfBounds         repeat;         // (error|warn|clamp|repeat)
    }
```

The `AorticInletFlowRate` file, shown in the Listing 5.4, contains the volume flow rate data for aorta as a time-series i.e., consisting of 1001 uniformly spaced time-flow rate value pairs with time starting from 0.0 s and ending at 1.0 s, resulting in a period of 1.0 s. The flow rate is going

into the domain, hence the values are positive as required by `timeVaryingFlowRateInletVelocity` boundary condition.

Listing 5.4: Flow rate time-series data in `AorticInletFlowRate` file

```
(
( 0.00000 5.25792000E-05 )
( 0.00100 5.51130000E-05 )
( 0.00200 5.76468200E-05 )
( 0.00300 6.01806300E-05 )
.
.
.
( 0.99800 5.04552600E-05 )
( 0.99900 5.15172200E-05 )
( 1.00000 5.25792000E-05 )
);
```

Listing 5.5: `U` field boundary condition at `inlet` patch for LCA

```
23      inlet
24      {
25          type                timeVaryingFlowRateInletVelocity;
26          flowRate            0;              // Volumetric/mass flow rate [m3/s or kg/s]
27          value               uniform (0 0 0); // placeholder
28          "file|fileName"     "$FOAM_CASE/DataFiles/CoronaryInletFlowRate";
29          outOfBounds         repeat;         // (error|warn|clamp|repeat)
30      }
```

Similarly, the `coronaryInletFlowRate` file, shown in the Listing 5.6, contains the volume flow rate data for LCA as a time-series i.e., consisting of 1001 uniformly spaced time-flow rate value pairs with time starting from 0.0 s and ending at 1.0 s, resulting in a period of 1.0 s. The flow rate is going into the domain, hence the values are positive as required by `timeVaryingFlowRateInletVelocity` boundary condition. The flow rate for LCA is around 2 % of that for aorta.

Listing 5.6: Flow rate time-series data in `CoronaryInletFlowRate` file

```
(
( 0.00000 3.56788000E-07 )
( 0.00100 3.57129822E-07 )
( 0.00200 3.57471644E-07 )
( 0.00300 3.57813465E-07 )
.
.
.
( 0.99710 3.51495783E-07 )
( 0.99810 3.51964391E-07 )
( 0.99910 3.52433000E-07 )
);
```

For the `walls` patch no-slip condition is applied by `fixedValue` of `uniform (0 0 0)`. At the outlet `zeroGradient` condition is used.

5.3.2 `p` boundary field

On the `walls` and `inlet` boundaries, `zeroGradient` condition is applied for `p` field, for all the cases. At the `outlet`, newly developed lumped parameter network (LPN) boundary condition is applied. Henceforth, in total 11 cases need to be solved. For `windkesselOutletPressure` BC, we have 9 subcases including one with `Resistive` BC, two with each of `WK2`, `WK3`, `WK4Series`, and `WK4Parallel` BCs, having one with `firstOrder` and another with `secondOrder` differencing scheme.

The values of parameters used for `Resistive` boundary condition at the aortic outlet are shown in the Listing 5.7.

Listing 5.7: p field boundary condition at outlet patch for aorta for Resistive BC

```
28    outlet
29    {
30        type                windkesselOutletPressure;
31        windkesselModel     Resistive;
32        Rd                  17690000;   //Distal resistance in [kgm^-4s^-1]
33        Pd                  9.7363e+03; // Distal pressure in [Pa]
34        value               uniform 0;
35    }
```

The values of windkessel parameters used for WK2 boundary condition with firstOrder differencing scheme at the aortic outlet are shown in the Listing 5.8. The values of windkessel parameters for another case with secondOrder differencing scheme are the same.

Listing 5.8: p field boundary condition at outlet patch for aorta for WK2 BC

```
28    outlet
29    {
30        type                windkesselOutletPressure;
31        windkesselModel     WK2;
32        diffScheme          firstOrder;
33        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
34        C                   8.3333e-09; // Compliance in [m^4s^2kg^-1]
35        Pd                  0;          // Distal pressure in [Pa]
36        value               uniform 0;
37    }
```

The values of windkessel parameters used for WK3 boundary condition with firstOrder differencing scheme at the aortic outlet are shown in the Listing 5.9. The values of windkessel parameters for another case with secondOrder differencing scheme are the same.

Listing 5.9: p field boundary condition at outlet patch for aorta for WK3 BC

```
28    outlet
29    {
30        type                windkesselOutletPressure;
31        windkesselModel     WK3;
32        diffScheme          firstOrder;
33        Rp                  13997000;   // Proximal resistance in [kgm^-4s^-1]
34        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
35        C                   1.0000e-08; // Compliance in [m^4s^2kg^-1]
36        Pd                  0;          // Distal pressure in [Pa]
37        value               uniform 0;
38    }
```

The values of windkessel parameters used for WK4Series boundary condition with firstOrder differencing scheme at the aortic outlet are shown in the Listing 5.10. The values of windkessel parameters for another case with secondOrder differencing scheme are the same.

Listing 5.10: p field boundary condition at outlet patch for aorta for WK4Series BC

```
28    outlet
29    {
30        type                windkesselOutletPressure;
31        windkesselModel     WK4Series;
32        diffScheme          firstOrder;
33        Rp                  13997000;   // Proximal resistance in [kgm^-4s^-1]
34        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
35        C                   2.5000e-08; // Compliance in [m^4s^2kg^-1]
36        L                   1.7995e+04; // Blood inertance in [kgm^-4]
37        Pd                  0;          // Distal pressure in [Pa]
38        value               uniform 0;
39    }
```

The values of windkessel parameters used for WK4Parallel boundary condition with firstOrder differencing scheme at the aortic outlet are shown in the Listing 5.11. The values of windkessel parameters for another case with secondOrder differencing scheme are the same.

Listing 5.11: p field boundary condition at outlet patch for aorta for WK4Parallel BC

```
28      outlet
29      {
30          type                windkesselOutletPressure;
31          windkesselModel     WK4Parallel;
32          diffScheme          firstOrder;
33          Rp                  1.3997e+10; // Proximal resistance in [kgm^-4s^-1]
34          Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
35          C                   1.0000e-08; // Compliance in [m^4s^2kg^-1]
36          L                   1.7995e+03; // Blood inertance in [kgm^-4]
37          Pd                  0;          // Distal pressure in [Pa]
38          value               uniform 0;
39      }
```

Similarly, the p boundary condition is defined at coronary outlet using coronaryOutletPressure. The values of LPN model parameters used for coronaryOutletPressure boundary condition with firstOrder differencing scheme at the coronary outlet are shown in the Listing 5.12. The values of LPN model parameters for another case with secondOrder differencing scheme are the same.

It can be seen that apart from the values of resistances and compliance, the path to a file containing time-series data for intramyocardial pressure is specified using PimFile entry. The PimData file, shown in the Listing 5.13, contains the left ventricular pressure data as a time-series i.e., consisting of 999 time-pressure value pairs with time starting from 0.0 s and ending at 0.99910 s, resulting in a time period of around 1.0 s. The pressure values are in Pa. Since the pressure data is for pressure inside left ventricular, the value of a scaling factor of 1.5 is specified using PimScaling entry.

Listing 5.12: p field boundary condition at outlet patch for LCA for coronaryOutletPressure BC

```
28      outlet
29      {
30          type            coronaryOutletPressure;
31          Ra              1.0179e+10;    // Arterial resistance [kg m^-4 s^-1]
32          Ram             1.6541e+10;    // Micro-arterial resistance [kg m^-4 s^-1]
33          Rv              5.0896e+09;    // Veinous resistance [kg m^-4 s^-1]
34          Rvm             0;             // Micro-veinous resistance [kg m^-4 s^-1]
35          Ca              8.6482e-12;    // Arterial compliance [m^4 s^2 kg^-1]
36          Cim             6.9972e-11;    // Intramyocardial compliance [m^4 s^2 kg^-1]
37          PimScaling      1.5;           // Intramyocardial pressure scaling (1.5 for LCA, 0.5 for RCA)
38          Pv              0;             // Distal pressure [Pa]
39          diffScheme      firstOrder;
40          PimFile         "$FOAM_CASE/DataFiles/PimData"; // File containing intramyocardial pressure
     data
41          value           uniform 0; // Initial value for pressure [Pa]
42      }
```

Listing 5.13: Intramyocardial pressure time-series data in PimData file

```
(
( 0.00000 2.85419827E+03 )
( 0.00100 2.88803679E+03 )
( 0.00200 2.92568009E+03 )
( 0.00300 2.96726253E+03 )
.
.
.
( 0.99710 2.78917846E+03 )
( 0.99810 2.81575383E+03 )
( 0.99910 2.81578041E+03 )
);
```

5.4 Physics and properties of the simulation: the constant directory

Apart from `blockMeshDict.m4` script placed inside `constant/polyMesh` directory, the `constant` directory in `solids4foam` environment contains several other dictionaries used to define the physics and other properties of problem.

5.4.1 The physicsProperties dictionary

The `physicsProperties` dictionary is required to define whether the simulation is for fluid, solids or fluid-solid interaction (FSI). In the present case, the simulation is fluid only with the walls consider rigid and not simulated, hence the type of physics is defined as `fluid` for aorta or LCA, as shown in the Listing 5.14.

Listing 5.14: The `constant/physicsProperties` dictionary

```
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       dictionary;
13     location    "constant";
14     object      physicsProperties;
15 }
16 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18 type        fluid;
```

5.4.2 The dynamicMeshDict dictionary

The `dynamicMeshDict` dictionary is required to define whether the mesh have motion or not. In the present case, the simulation is CFD only and the walls are consider rigid, hence no mesh motion is defined for aorta or LCA, as shown in the Listing 5.15.

Listing 5.15: The `constant/dynamicMeshDict` dictionary

```
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       dictionary;
13     object      dynamicMeshDict;
14 }
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17 dynamicFvMesh       staticFvMesh;
18
19 // ************************************************************************* //
```

5.4.3 The transportProperties dictionary

The `transportProperties` dictionary is required to define the properties including kinematic viscosity (ν) and density (ρ) of the fluid. In the present case, blood is considered as `Newtonian` fluid, hence its kinematic viscosity and density are defined for aorta or LCA, as shown in the Listing 5.16.

Listing 5.16: The `constant/transportProperties` dictionary

```
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
```

```
12      class       dictionary;
13      location    "constant";
14      object      transportProperties;
15  }
16  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18  transportModel      Newtonian;
19
20  rho                 rho [1 -3 0 0 0 0 0] 1060;
21  nu                  nu [0 2 -1 0 0 0 0] 3.77e-6;
22
23  // ************************************************************************* //
```

5.4.4 The turbulenceProperties dictionary

The turbulenceProperties dictionary is required to define the type of fluid simulation. In the present case, Reynolds-Averaged Simulation (RAS) model is used for simulating the hemodynamics in the aorta or LCA, as shown in the Listing 5.17.

Listing 5.17: The constant/turbulenceProperties dictionary

```
8   FoamFile
9   {
10      version     2.0;
11      format      ascii;
12      class       dictionary;
13      location    "constant";
14      object      turbulenceProperties;
15  }
16  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18  simulationType      RASModel;
```

5.4.5 The RASProperties dictionary

The RASProperties dictionary is required to define the type of RASModel to be used for fluid simulation. Based on the mean flow rate for aorta and LCA, the flow in both vessels is considered laminar, hence the type of RASModel is defined as laminar and turbulence is turned off, as shown in the Listing 5.18.

Listing 5.18: The constant/RASProperties dictionary

```
8   FoamFile
9   {
10      version     2.0;
11      format      ascii;
12      class       dictionary;
13      location    "constant";
14      object      RASProperties;
15  }
16  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18  RASModel        laminar;
19
20  turbulence      off;
21
22  printCoeffs     on;
23
24  // ************************************************************************* //
```

5.4.6 The fluidProperties dictionary

The fluidProperties dictionary is required to define the type of solver to be used for fluid simulation. A variation of pimpleFoam available in solids4foam for fluid simulations for transient laminar and turbulent simulations, pimpleFluid is used, as shown in the Listing 5.19. Moreover, the ddtCorr is turned off, and adjustPhi is turned on.

Listing 5.19: The constant/fluidProperties dictionary

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fluidProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

fluidModel          pimpleFluid;

pimpleFluidCoeffs
{
    ddtCorr         false;
    adjustPhi       true;
    solveEnergyEq   false;
}

// ************************************************************************* //
```

5.5 Discretization, decomposition and controls: the system directory

The system directory, within the main $FOAM_CASE directory contains the dictionaries for defining the solution schemes, solution methods, domain decomposition settings for parallel run, and simulation control.

5.5.1 The fvSchemes dictionary

The fvSchemes dictionary is used to define the discretization schemes for various mathematical operators used in the governing equations of the simulation. This dictionary ensures that the numerical solution is obtained with appropriate accuracy and stability for the problem under consideration. Each section of the dictionary specifies the method by which a particular operator or quantity is discretized. The settings used in the fvSchemes dictionary for the present simulations are shown in the Listing 5.20.

In the ddtSchemes section, the time derivative terms are discretized using the Euler method. This method is selected for its simplicity and efficiency in handling transient simulations.

The gradSchemes section specifies the treatment of gradient operators. A least-squares method is employed by default, ensuring that gradients are calculated accurately using information from neighboring cells.

In the divSchemes section, the discretization of divergence terms is configured. The default divergence scheme is set to none. For the term div(phi,U), a Gauss linearUpwind scheme with a cell-limited least-squares approach is used, with a limiting factor of 1 to maintain numerical stability. The viscous stress terms, such as div((nuEff*dev(grad(U).T()))) and div((nuEff*dev(T(grad(U)))), are handled using a Gauss linear method.

The laplacianSchemes section describes the handling of Laplacian terms. The default scheme is set to none, while specific Laplacian terms, such as those involving nu and U, are discretized using a Gauss linear corrected scheme. Additionally, Laplacian terms involving pressure correction, like

`laplacian((1|A(U)),p)`, are treated using a `Gauss linear skewCorrected` scheme with a factor of `0.5`.

The `interpolationSchemes` section dictates the interpolation method for field values. The default interpolation method is linear, which is applied to all fields, including velocity `U`.

Finally, the `snGradSchemes` section defines the discretization of surface-normal gradients. A `skewCorrected` method with a correction factor of `0.5` is used by default to account for mesh skewness and improve accuracy in gradient calculations.

Listing 5.20: The `system/fvSchemes` dictionary

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         leastSquares;
}

divSchemes
{
    default                         none;
    div(phi,U)                      Gauss linearUpwind cellLimited leastSquares 1;
    div((nuEff*dev(grad(U).T())))   Gauss linear;
    div((nuEff*dev(T(grad(U)))))    Gauss linear;

    div(phi,epsilon)                Gauss linear;
    div(phi,k)                      Gauss linear;
}

laplacianSchemes
{
    default                         none;
    laplacian(nu,U)                 Gauss linear corrected;
    laplacian(nuEff,U)              Gauss linear corrected;

    laplacian(rAU,p)                Gauss linear corrected;

    laplacian((1|A(U)),p)           Gauss linear skewCorrected 0.5;

    laplacian(DepsilonEff,epsilon)  Gauss linear corrected;
    laplacian(DkEff,k)              Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
    interpolate(U)  linear;
}

snGradSchemes
{
    default         skewCorrected 0.5;
}
```

5.5.2 The fvSolution dictionary

The fvSolution dictionary is utilized to define the solution algorithms and solver settings for the numerical simulation. This dictionary provides detailed configurations for the solvers used for different fields, the iterative procedures, and residual control. The settings used in the fvSolution dictionary for the present simulations are shown in the Listing 5.21.

In the solvers section, the pressure field (p and pFinal) is solved using the Preconditioned Conjugate Gradient (PCG) solver. A Diagonal Incomplete Cholesky (DIC) preconditioner is employed to enhance convergence. A strict absolute tolerance of 1×10^{-7} is specified, while the relative tolerance is set to zero, ensuring high accuracy. A minimum of one iteration is enforced. For velocity (U, UFinal), the Preconditioned Bi-Conjugate Gradient (PBiCG) solver is applied, using a Diagonal Incomplete LU (DILU) preconditioner. The same absolute tolerance, relative tolerance, and minimum iteration criteria are used.

The iterative process for pressure-velocity coupling is governed by the PIMPLE algorithm. For the PIMPLE loop, a single outer corrector is specified, followed by two inner correctors and one non-orthogonal corrector. Residual control is included to monitor and enforce convergence for key variables. For velocity (U) and pressure (p), an absolute tolerance of 1×10^{-7} is set, with the relative tolerance again specified as zero.

Considering the simplicity of the problem, no relaxationFactors are set. However, if required, under-relaxation factors could be applied to improve stability during the solution process. For the velocity fields (U and UFinal), a relaxation factor of 0.7 is proposed, while for pressure fields (p and pFinal), a factor of 0.3 is suggested.

Listing 5.21: The system/fvSolution dictionary

```
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       dictionary;
13     location    "system";
14     object      fvSolution;
15 }
16 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18 solvers
19 {
20     "p|pFinal"
21     {
22         solver          PCG;
23         preconditioner  DIC;
24         tolerance       1e-7;
25         relTol          0;
26         minIter         1;
27     }
28
29     "U|UFinal|k|epsilon"
30     {
31         solver          PBiCG;
32         preconditioner  DILU;
33         tolerance       1e-7;
34         relTol          0;
35         minIter         1;
36     }
37 }
38
39 PISO
40 {
41     nCorrectors                3;
42     nNonOrthogonalCorrectors    1;
43 }
44
45 PIMPLE
46 {
```

```
    nOuterCorrectors            1; // Iterative PISO
    nCorrectors                 2;
    nNonOrthogonalCorrectors    1;

    residualControl
    {
        U
        {
            relTol      0;
            tolerance   1e-7;
        }
        p
        {
            relTol      0;
            tolerance   1e-7;
        }
    }

}
```

5.5.3 The decomposeParDict dictionary

The decomposeParDict dictionary is used to define the settings for domain decomposition, which is necessary for running parallel simulations. This file specifies the number of subdomains, the decomposition method, and additional coefficients required for the chosen method. The settings used in the decomposeParDict dictionary for the present simulations are shown in the Listing 5.22.

The number of subdomains is set to eight to divide the computational domain into eight parts for parallel processing. This allows the simulation to be distributed across multiple processors, enhancing computational efficiency and reducing simulation time.

The decomposition method selected is the simple method. This method is characterized by its straightforward partitioning strategy, making it suitable for domains with relatively simple geometries or structured meshes.

In the simpleCoeffs section, further details about the decomposition are provided. The parameter n specifies the number of divisions along each coordinate direction. Considering the simplicity of the domain, it is divided into one part along the x- and y-axes, and eight parts along the z-axis i.e., length of the straight cylinder. The parameter delta is set to 0.001, which determines the tolerance for load balancing during decomposition, ensuring that computational work is distributed evenly among the processors.

Listing 5.22: The system/decomposeParDict dictionary

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      decomposeParDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

numberOfSubdomains      8;

method                  simple;

simpleCoeffs
{
    n           (1 1 8);
    delta       0.001;
}
```

5.5.4 The controlDict dictionary

The controlDict dictionary serves as the primary configuration file for controlling the simulation's execution and output. It defines key parameters related to time-stepping, output frequency, and function objects for postprocessing. The settings used in the decomposeParDict dictionary for the present simulations are shown in the Listing 5.23.

The library liblumpedParameterNetworkBC.so is loaded to enable the usage of specific LPN model boundary conditions including windkesselOutletPressure and coronaryOutletPressure developed before. The simulation is executed using the solids4Foam application. The simulation is configured to start from the latest available time, with the initial time set to 0 seconds. The end time is specified as 8 seconds to give enough time to LPN models to reach stable periodic values, and a constant time step of 0.001 seconds is used. Time step adjustments are disabled, ensuring a fixed time-step size throughout the simulation. A maximum Courant number of 0.5 is enforced to ensure numerical stability.

Listing 5.23: The system/controlDict dictionary

```
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       dictionary;
13     location    "system";
14     object      controlDict;
15 }
16 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18 libs                ("liblumpedParameterNetworkBC.so");
19
20 application         solids4Foam;
21
22 startFrom           latestTime;
23
24 startTime           0;
25
26 stopAt              endTime;
27
28 endTime             8;
29
30 deltaT              0.001;
31
32 writeControl        timeStep;
33
34 writeInterval       50;
35
36 purgeWrite          0;
37
38 writeFormat         ascii;
39
40 writePrecision      9;
41
42 writeCompression    uncompressed;
43
44 timeFormat          general;
45
46 timePrecision       6;
47
48 runTimeModifiable   yes;
49
50 adjustTimeStep      no;
51
52 maxCo               0.5;
```

The simulation results are written to disk based on the time step control, with an output interval of 50 time steps. Postprocessing functionality is used through the functions section in order to

record flow rates and pressures at the inlet and outlet patches for the comparison with the expected profiles in order to validate the results of LPN boundary conditions being used. Several function objects are defined to calculate and log specific quantities, as shown in the Listing 5.24. The `flowRateInlet` and `flowRateOutlet` objects compute the flow rate at the inlet and outlet patches, respectively, using the `faceSource` type. The operation performed is a summation of the flux field `phi` at the given patch. Similarly, the `pAverageInlet` and `pAverageOutlet` objects compute the area-averaged pressure at the inlet and outlet patches. For all function objects, output is configured to occur at each time step, with results written to the log and stored in the specified output format.

Listing 5.24: The `functions` to compute inlet and outlet flow rates and pressures in the `system/controlDict` dictionary

```
functions
(

    flowRateInlet
    {
        type            faceSource;
        functionObjectLibs ("libfieldFunctionObjects.so");
        enabled         true;
        outputControl   timeStep;
        log             true;
        valueOutput     true;
        source          patch;
        sourceName      inlet;
        operation       sum;
        surfaceFormat   off;

        fields
        (
            phi
        );
    }

    flowRateOutlet
    {
        type            faceSource;
        functionObjectLibs ("libfieldFunctionObjects.so");
        enabled         true;
        outputControl   timeStep;
        log             true;
        valueOutput     true;
        source          patch;
        sourceName      outlet;
        operation       sum;
        surfaceFormat   off;

        fields
        (
            phi
        );
    }

    pAverageInlet
      {
         type            faceSource;
         functionObjectLibs ("libfieldFunctionObjects.so");
         enabled         true;
         outputControl   timeStep;
         log             true;
         valueOutput     true;
         source          patch;
         sourceName      inlet;
         operation      areaAverage;
        surfaceFormat   off;
        fields
        (
```

```
            p
        );
    }

    pAverageOutlet
    {
        type            faceSource;
        functionObjectLibs ("libfieldFunctionObjects.so");
        enabled         true;
        outputControl   timeStep;
        log             true;
        valueOutput     true;
        source          patch;
        sourceName      outlet;
        operation       areaAverage;
        surfaceFormat   off;
        fields
        (
            p
        );
    }

);
```

5.6 Running the simulation

The simulations are run in two steps. First the directory structure required for the `solids4foam` environment is created, and then each case is run one after the other.

5.6.1 Creating the directory structure

A `CreateDirStruc` bash script, shown in the Listing 5.25, is written to create the required directory structure for all 11 cases extracting elements from the master case directory `LPNBCTestCase`.

Listing 5.25: The `CreateDirStruc` bash script

```
#!/bin/bash

echo "Creating directory structure..."

# Base directory name
base_dir="LPNBCTestCase"

# Array of simulation types and schemes
simulations=("Resistive" "WK2" "WK3" "WK4Series" "WK4Parallel" "Coronary")
schemes=("firstOrder" "secondOrder")

# Function to copy and create directory structure
create_simulation_structure() {
    sim=$1
    scheme=$2

    # Define new simulation directory
    sim_dir="${base_dir}_${sim}_${scheme}"

    echo "Creating directory structure for ${sim_dir}..."

    # Create base directories
    mkdir -p "${sim_dir}/0" "${sim_dir}/constant/polyMesh" "${sim_dir}/system" "${sim_dir}/DataFiles
    "

    # Handle U file
    if [[ "${sim}" == "Coronary" ]]; then
        cp "${base_dir}/0.orig/U.Coronary" "${sim_dir}/0/U"
```

```
    else
        cp "${base_dir}/0.orig/U.WK" "${sim_dir}/0/U"
    fi

    # Handle p file
    if [[ "${sim}" == "Resistive" ]]; then
        cp "${base_dir}/0.orig/p.Resistive" "${sim_dir}/0/p"
    else
        cp "${base_dir}/0.orig/p.${sim}.${scheme}" "${sim_dir}/0/p"
    fi

    # Copy constant files
    cp -r "${base_dir}/constant/"* "${sim_dir}/constant/"
    rm -r "${sim_dir}/constant/polyMesh/blockMeshDict.m4."*

    # Handle blockMeshDict
    if [[ "${sim}" == "Coronary" ]]; then
        cp "${base_dir}/constant/polyMesh/blockMeshDict.m4.Coronary" "${sim_dir}/constant/polyMesh/
     blockMeshDict.m4"
    else
        cp "${base_dir}/constant/polyMesh/blockMeshDict.m4.WK" "${sim_dir}/constant/polyMesh/
     blockMeshDict.m4"
    fi

    # Handle DataFiles
    if [[ "${sim}" == "Coronary" ]]; then
        cp "${base_dir}/DataFiles/CoronaryInletFlowRate" "${sim_dir}/DataFiles/"
        cp "${base_dir}/DataFiles/PimData" "${sim_dir}/DataFiles/"
    else
        cp "${base_dir}/DataFiles/AorticInletFlowRate" "${sim_dir}/DataFiles/"
    fi

    # Copy system files
    cp -r "${base_dir}/system/"* "${sim_dir}/system/"

    # Handle Allrun and Allclean scripts
    if [[ "${sim}" == "Coronary" ]]; then
        cp "${base_dir}/Allrun.Coronary" "${sim_dir}/Allrun"
        cp "${base_dir}/Allclean.Coronary" "${sim_dir}/Allclean"
        chmod +x ${sim_dir}/Allrun ${sim_dir}/Allclean
    else
        cp "${base_dir}/Allrun.WK" "${sim_dir}/Allrun"
        cp "${base_dir}/Allclean.WK" "${sim_dir}/Allclean"
        chmod +x ${sim_dir}/Allrun ${sim_dir}/Allclean
    fi
}

# Loop through all simulations and schemes
for sim in "${simulations[@]}"; do
    if [[ "${sim}" == "Resistive" ]]; then
        create_simulation_structure "$sim" ""
    else
        for scheme in "${schemes[@]}"; do
            create_simulation_structure "$sim" "$scheme"
        done
    fi
done

echo "Directory structure creation complete!"
```

The execution of the bash script creates the OpenFOAM case directory structure for each of these 11 cases, very same to the one shown for the `LPNBCTestCase_Coronary_secondOrder` in Figure 5.3. After the directory structure is created, the main `Allrun` bash script is executed which first execute the `Allwmake` script inside the `lumpedParameterNetworkBC` directory to compile the LPN BCs, and then execute the `Allrun` scripts in each of the case directories. A `Allrun` bash script inside the case directory, as shown in the Listing 5.26, is written to execute commands in an ordered

manner for each case. It sources essential functions for tutorial cleaning and utility operations from the `foam-extend RunFunctions` script and the `solids4foam` scripts. These utilities form the basis for running OpenFOAM applications and `solids4foam`-specific tasks. The script then checks the compatibility of the case format by invoking the `solids4Foam::convertCaseFormat` function, ensuring that the case is in the correct format. Additionally, it performs a check to verify that the case can only run with `foam-extend`.

```
LPNBCTestCase_Coronary_secondOrder/
├── 0/ {U, p}
├── {Allclean, Allrun}
├── DataFiles/ {CoronaryInletFlowRate, PimData}
├── constant/
│   ├── {RASProperties, dynamicMeshDict, fluidProperties, physicsProperties,
│   │   transportProperties, turbulenceProperties}
│   └── polyMesh/ {blockMeshDict.m4}
└── system/
    └── {controlDict, decomposeParDict, fvSchemes, fvSolution}
```

Figure 5.3: Directory structure of `LPNBCTestCase_Coronary_secondOrder` case in `solids4foam`

Next, the script generates the block mesh for the simulation by creating the `blockMeshDict` file from its `m4` template using the `m4` macro processor. Once created, the `blockMesh` utility is executed to generate the computational mesh, followed by running `checkMesh` to verify the quality and consistency of the mesh. The domain decomposition for parallel processing is then carried out using the `decomposePar` utility with the `-force` flag, ensuring that the case is correctly divided among the specified number of processors.

To resolve parsing issues in the initial files for velocity (`U`) and pressure (`p`), `sed` commands are employed to correct the paths of the inlet flow rate file and the `PimData` file in all decomposed processor directories. The `solids4Foam` solver is then executed in parallel using `mpirun` with `8` processors, and the output is redirected to a log file, `log.solids4Foam`. After the simulation, the results from all processors are reconstructed into a single dataset using the `reconstructPar` utility.

Residual data from the log file is extracted using the `foamLog` utility for convergence analysis. Finally, a placeholder file (`CoronaryStraight.foam`) is created to facilitate the visualization of the results in ParaView.

Listing 5.26: The `Allrun` bash script

```
#!/bin/bash

# Source tutorial clean functions
. $WM_PROJECT_DIR/bin/tools/RunFunctions

# Source solids4Foam scripts
source solids4FoamScripts.sh

# Check case version is correct
solids4Foam::convertCaseFormat .

# Run only with foam-extend
solids4Foam::caseOnlyRunsWithFoamExtend

# Create blockMeshDict from m4 script
m4 constant/polyMesh/blockMeshDict.m4 > constant/polyMesh/blockMeshDict

# Run blockMesh
solids4Foam::runApplication blockMesh

```

```
# Run checkMesh
solids4Foam::runApplication checkMesh

# Run decomposePar
solids4Foam::runApplication decomposePar -force

# Fix the parcing issues in the 0/U files in all processor directories
sed -i 's/fileName\s\{1,\}\("$FOAM_CASE\/DataFiles\/CoronaryInletFlowRate"\)/"file|fileName"
        \1/' processor*/0/U

# Run solids4Foam simulation in parallel
mpirun -np 8 solids4Foam -parallel > log.solids4Foam

# Run reconstructPar
solids4Foam::runApplication reconstructPar

# Create residual files
foamLog log.solids4Foam

# Create file to for the visualization in ParaView
touch CoronaryStraight.foam
```

5.7 Results and validation

In all the cases, the simulations are fluid-only, where walls are considered rigid i.e., non-compliant or inflexible. The fluid is also incompressible. Moreover, the domain has only one inlet and one outlet. Hence, the law of conservation of mass dictates that the volume flow rate going into the domain at any instant during the simulation equals to the volume flow rate coming out of the domain. Therefore, the volume flow rate specified at the inlet of the domain can be used to as an outlet flow rate to solve the ordinary differential equations (ODEs) for all the LPNs to get the outlet pressure. A MATLAB script is written to solve the ODEs for each LPN using their respective flow rates and parameter values, set same as in OpenFOAM simulations, either employing a standard `ode45` solver or `ode15s` solver. The `ode45` solver is based on the Dormand-Prince method, a type of Runge-Kutta method. It is widely used for solving non-stiff, smooth ODEs and provides a good balance between accuracy and computational efficiency. It adapts its step size during the integration process to achieve the desired accuracy and is typically the default solver for most ODE problems unless the equations exhibit stiffness. Whereas, `ode15s` solver is designed for stiff ODEs, which are characterized by rapidly changing solutions and may require very small time steps for accurate integration. It is based on the numerical method of implicit multi-step methods and is specifically optimized to handle stiff problems more efficiently than `ode45`. It is more suitable for problems where the solution exhibits large differences in timescales, making explicit solvers like `ode45` less effective. In order to validate the results obtained through OpenFOAM simulations, recorded outlet pressure for each simulation case is compared with the numerical solution of the LPN models for the same parameter values and flow rate obtained through MATLAB.

5.7.1 Resistive BC

The OpenFOAM and MATLAB solutions for `Resistive` BC in `windkesselOutletPressure` family are shown in Figure 5.4. It can be seen that the inlet and outlet flow rates are the same ensuring that the mass balance is obtained and continuity holds for the problem. The solution for the pressure obtained through both methods also matches exactly. For the single-element, resistive LPN, the relationship between pressure and volume flow rate is linear with slope equals to the distal resistance and an offset equals to the distal pressure. Henceforth, the the pressure follow exactly the same profile as that of the flow rate.

It should be noted that single-element resistive model is the simplest representation of flow dynamics, where a linear resistance is used to simulate the relationship between flow and pressure drop. This boundary condition ignores the effects of vessel compliance and blood inertia, and is

usually suitable for steady-state simulations where the mean effect of downstream blood vessels is considered. The simplicity of this model makes it computationally inexpensive and straightforward to implement. Given the value of overall impedance, it can provide a quick estimate of the pressure drop across a vascular segment under steady conditions. However, the lack of compliance and inertial effects results in an oversimplification of the pulsatile flow behavior, making this model inadequate for capturing transient flow dynamics and wave reflections. Consequently, the results may diverge significantly from physiological reality, especially in scenarios with time-varying boundary conditions, as in transient conditions the pressure profile produced by this model is just a multiple of flow rate profile with certain offset.

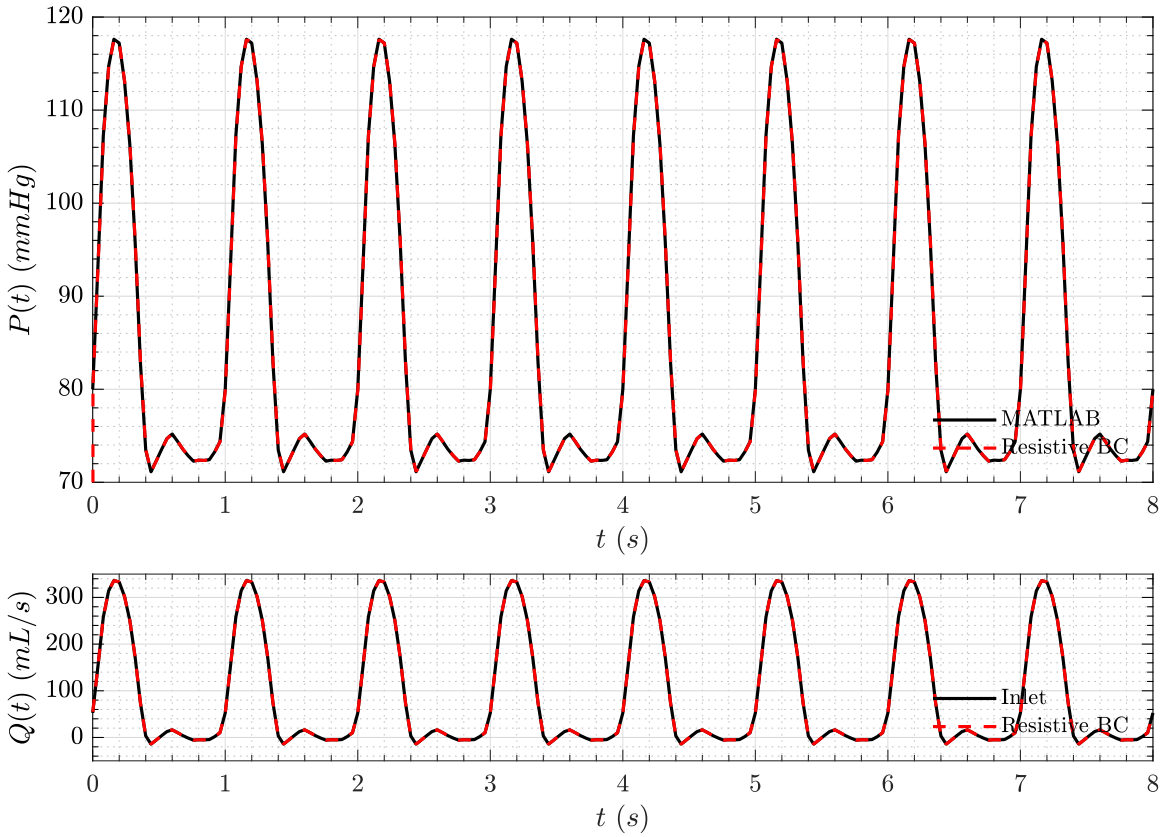


Figure 5.4: The comparison between outlet pressure obtained by OpenFOAM and MATLAB solutions (top), and inlet and outlet flow rates (bottom) for single-element windkessel BC

5.7.2 2-element windkessel BC

The OpenFOAM and MATLAB solutions for `WK2` BC in `windkesselOutletPressure` family are shown in Figure 5.5. It can be seen that the inlet and outlet flow rates are same ensuring that the mass balance is obtained and continuity holds for the problem. The solution for the pressure obtained through both methods also matches exactly. The two-element windkessel model is a RC-circuit which gives pressure out-of-phase from the flow rate. Henceforth, the pressure profile lags the flow rate profile. Given the small time-step size, the solution obtained by both first and second-order differencing matches closely. The initial pressure in both case is set as zero, hence the solution solution also starts from the zero, and no offset due to distal pressure is observed in the solution as the term is set zero.

The RC model introduces compliance (C) in addition to distal resistance (R_d). This improvement allows the model to simulate the storage and release of blood volume in response to pulsatile flow,

approximating the elastic behavior of the vascular wall. By incorporating compliance, the RC model captures basic pulsatile flow dynamics and is better suited for time-dependent simulations than the simplest single-element R model. It accounts for the damping effect of compliance on pressure oscillations, making it suitable for systems with significant capacitive effects. While it adds realism compared to the R model, the RC model does not account for proximal vascular resistance or inertial effects of blood. As a result, it may underestimate the complexity of flow dynamics in larger or branching vascular networks.

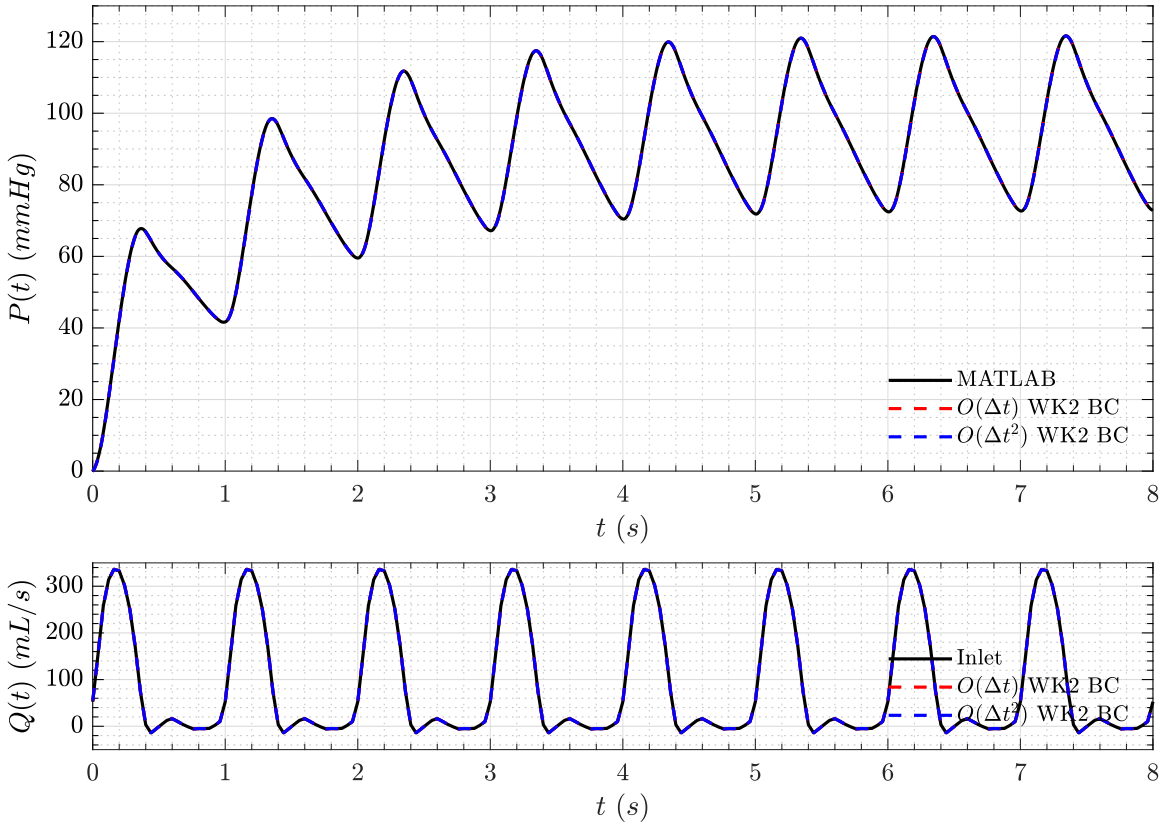


Figure 5.5: The comparison between outlet pressure obtained by OpenFOAM and MATLAB solutions (top), and inlet and outlet flow rates (bottom) for two-element windkessel BC

5.7.3 3-element windkessel BC

The OpenFOAM and MATLAB solutions for `WK3` BC in `windkesselOutletPressure` family are shown in Figure 5.6. It can be seen that the inlet and outlet flow rates are same ensuring that the mass balance is obtained and continuity holds for the problem. After two cardiac cycles, solution for the pressure obtained through both methods also matches exactly. The three-element windkessel model is a RCR-circuit which unlike simple RC-circuit gives pressure slightly less out-of-phase from the flow rate. Henceforth, the pressure profile lags the flow rate profile by a relatively less amount. Given the small time-step size, the solution obtained by both first and second-order differencing matches closely. The initial pressure in both case is set as zero, hence the solution solution also starts from the zero, and no offset due to distal pressure is observed in the solution as the term is set zero. It can be seen that both solutions take around 5 cardiac cycles to settle on a stable pulsatile solution.

The RCR model adds a proximal resistance (R_d) to the RC model, accounting for the pressure drop in immediate downstream larger blood vessels more accurately. This configuration is more

representative of vascular systems where proximal impedance significantly affects the flow and pressure dynamics. The addition of proximal resistance enables the model to proximal pressure drop phenomena, improving the accuracy of overall pressure waveforms and flow distribution. The RCR model is widely used for its balance between computational efficiency and physiological realism. However, it lacks the ability to model inertial effects of blood flow, which can become significant at higher frequencies or in large arteries.

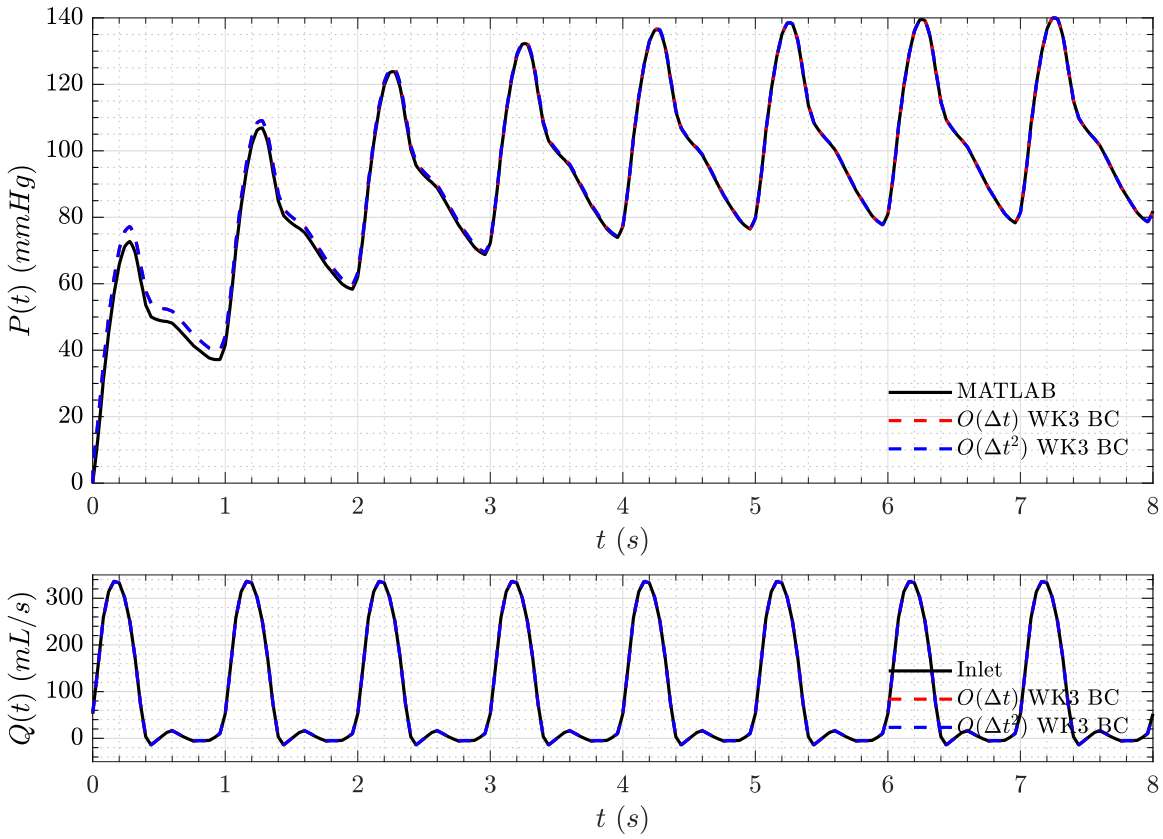


Figure 5.6: The comparison between outlet pressure obtained by OpenFOAM and MATLAB solutions (top), and inlet and outlet flow rates (bottom) for three-element windkessel BC

5.7.4 4-element series windkessel BC

The OpenFOAM and MATLAB solutions for `WK4Series` BC in `windkesselOutletPressure` family are shown in Figure 5.7. It can be seen that the inlet and outlet flow rates are same ensuring that the mass balance is obtained and continuity holds for the problem. After around 5 cardiac cycles, solution for the pressure obtained through both methods also matches exactly. The governing ODE for four-element windkessel mode with inductance in series involve second derivative flow rate, hence the error in the solution is higher for initial cardiac cycles. It should be noted that for the very small value of inductance or blood inertance L which is attached in series to the proximal resistance R_p in this RCRL model, the pressure profile is almost similar to that obtained through three-element windkessel (RCR) model. If the inertia of blood is not considered i.e., the value of L is set equals to zero, RCRL model turns into a simple RCR model, accounting only for the compliance of the vessel as well as the resistanc to the flow in proximal and distal downstream regions. Given the small time-step size, the solution obtained by both first and second-order differencing matches closely. The initial pressure in both case is set as zero, hence the solution solution also starts from the zero, and no offset due to distal pressure is observed in the solution as the term is set zero. It can be seen that both solutions take around 7 to 8 cardiac cycles to settle on a stable pulsatile solution.

This RCRL configuration incorporating an inductance (L) term in series with the proximal resistance (R_p), is able account for the inertial effects of blood as well. The inclusion of inductance allows the model to capture the dynamic interplay between resistance, compliance, and inertia, providing a more comprehensive representation of vascular flow. The addition of inertial effects makes this model suitable for simulations involving rapid changes in flow or pressure, such as during systole. It provides a more accurate depiction of proximal pressure and flow waveforms, especially in larger arteries where inertial effects are significant. The increased complexity of the RCRL model results in higher computational costs. Furthermore, for the problems where the tuning of windkessel parameters is required to match the mean flow in the blood vessel, tuning higher number of parameters can be challenging.

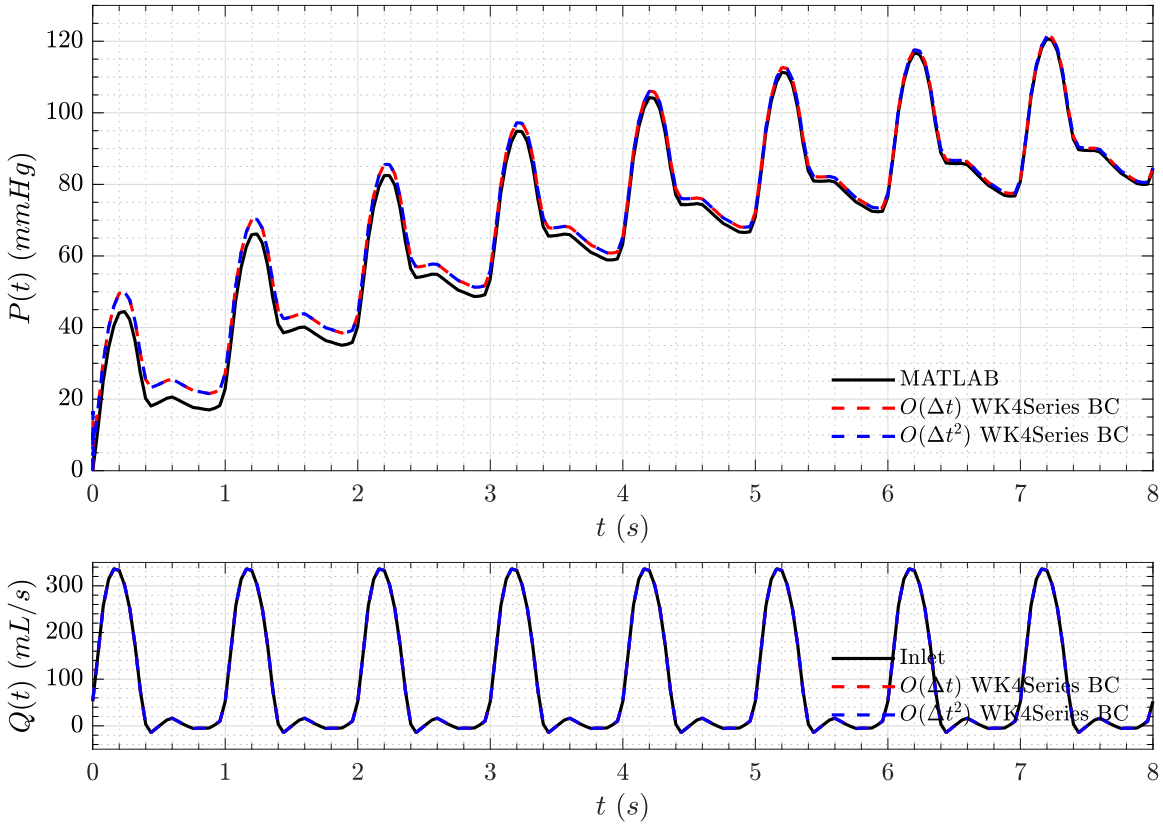


Figure 5.7: The comparison between outlet pressure obtained by OpenFOAM and MATLAB solutions (top), and inlet and outlet flow rates (bottom) for four-element (series) windkessel BC

5.7.5 4-element parallel windkessel BC

The OpenFOAM and MATLAB solutions for `WK4Parallel` BC in `windkesselOutletPressure` family are shown in Figure 5.8. It can be seen that the inlet and outlet flow rates are same ensuring that the mass balance is obtained and continuity holds for the problem. Given the small time-step size, the solution obtained by both first and second-order differencing matches closely. The initial pressure in both case is set as zero, hence the solution solution also starts from the zero, and no offset due to distal pressure is observed in the solution as the term is set zero. It can be seen that both solutions take around 7 to 8 cardiac cycles to settle on a stable pulsatile solution.

It should be noted that in this RCRL model where inductance or blood inertance L which is attached in parallel to the proximal resistance R_p, the effect of proximal resistance R_p and other parameters dominates the effect of blood inertance L until or unless its value is sufficiently large. However, unlike the series RCRL model, the blood inertance L cannot be set equals to zero in this

model, as it will cause the division by zero invoking discontinuity. However, setting the value of proximal resistance R_p equals to zero will not only remove its effect, it will also remove the effect of blood inertance L on the solution, turning this model to a simple RC circuit, accounting only for the compliance of the vessels, and distal elements. Therefore, the pressure profile obtained by this model follows virtually the same shape as that obtained by two-element windkessel model. However, setting the higher value for blood inertance L in this model causes fluctuations which would increase with the increment in inertance value, rendering this model highly prone to noise or inaccuracies if the values of parameters are not fine tuned. All in all, this arrangement modifies the interaction between inertial effects and resistance, influencing the pressure-flow relationship differently compared to the series configuration. Similar to the series configuration, this model is computationally more expensive and requires precise parameter estimation. Additionally, the parallel arrangement may lead to numerical instabilities if the parameters are not carefully chosen.

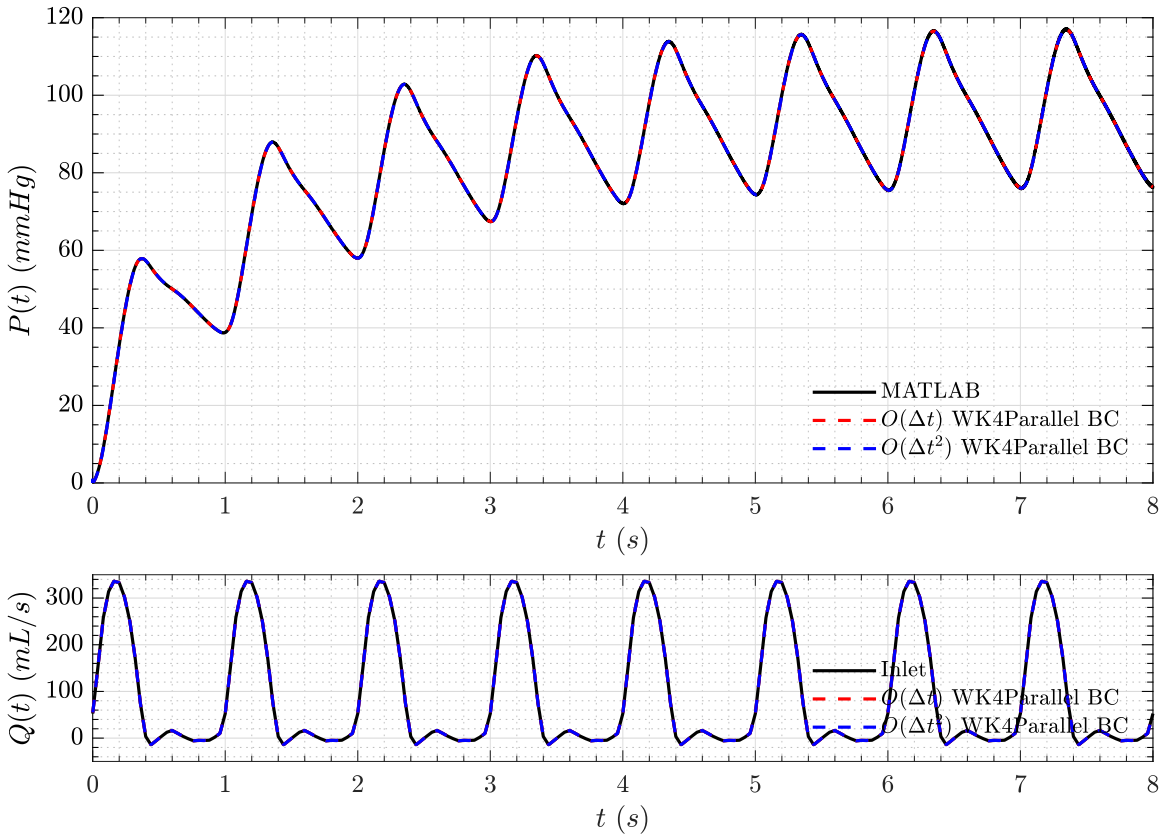


Figure 5.8: The comparison between outlet pressure obtained by OpenFOAM and MATLAB solutions (top), and inlet and outlet flow rates (bottom) for four-element (parallel) windkessel BC

5.7.6 Coronary LPN BC

The OpenFOAM and MATLAB solutions for `coronaryOutletPressure` BC are shown in Figure 5.9. It can be seen that the inlet and outlet flow rates are same ensuring that the mass balance is obtained and continuity holds for the problem. The initial pressure in both case is set as zero, hence the solution solution also starts from the zero, and no offset due to distal pressure is observed in the solution as the term is set zero.

The scaled time-variant intramyocardial pressure $P_{im}(t)$ is also plotted for the sake of comparison. It can be seen that unlike windkessel models, the coronary LPN model, due to prominent contribution of intramyocardial pressure added after intramyocardial compliance between arterial and veinous

sides of coronary bed, gets to a stable pulsatile pressure profile rapidly in only 1 or 2 cardiac cycles. However, this might not be the case in the patient-specific cases where the aortic section is also included in the geometry and the aortic inlet flow is specified. In this case the development of coronary outlet pressure profile depends, apart from the solution of whole domain, upon the windkessel model used at the aortic inlet. As only around 4% of total cardiac output (CO) goes into the coronary arteries, the accuracy of the solution for pressure as well as velocity field in the coronary arteries depends highly on the accurate flow division in the case of both fluid only CFD study and fluid-solid interaction (FSI) study of compliant vessels. And the accuracy of flow division depends highly on the accuracy of boundary modeling. It is interesting to note that error in the solution obtained through OpenFOAM implementation is higher at the upper and lower peaks of the profile. It is due to the fact that the mathematical model of coronary LPN consists of a system of two linear ODEs which should be solved simultaneously to get the solution for the intermediate pressure, and the coronary outlet pressure. However, unlike the ODE solver in MATLAB which does solve the equations simultaneously, the methodology implemented in the OpenFOAM first solve the ODE for intermediate pressure term using the historical values of coronary outlet pressure, and then solve the ODE for coronary outlet pressure using the current value of intermediate pressure. This separate solution of two ODEs whose solution in fact depends on each other introduces a continuous disagreement between these two solution which is relatively more pronounced at the peaks.

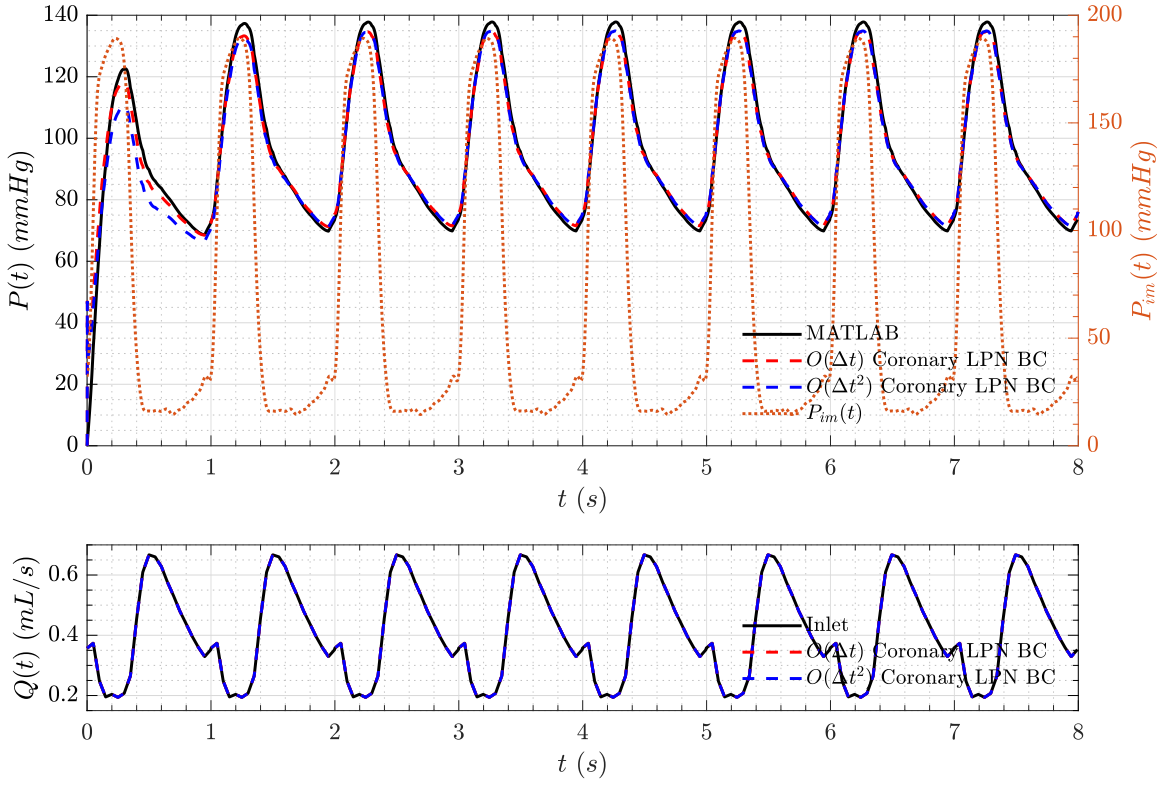


Figure 5.9: The comparison between outlet pressure obtained by OpenFOAM and MATLAB solutions (top), and inlet and outlet flow rates (bottom) for coronary LPN model BC

Chapter 6

Conclusion and Future Work

The implementation of lumped parameter network (LPN) boundary conditions in OpenFOAM for patient-specific coronary artery simulations has been successfully undertaken, with a focus on accurately modeling the hemodynamics of coronary arteries. Through this work, significant advancements have been made in extending the capabilities of OpenFOAM to prescribe realistic and physiologically relevant outlet pressure boundary conditions to simulate blood vessels.

The complexities of coronary hemodynamics, including the distinct systolic and diastolic flow phases and the influence of intramyocardial pressure, have been effectively addressed through the integration of specialized LPN models. These models, encompassing various configurations of resistive, capacitive, and inductive elements, were derived to simulate the unique physiological characteristics of blood flow in larger blood vessels, known as windkessel vessels. The coronary LPN model has been distinguished from traditional windkessel models by incorporating elements that account for intramyocardial compliance and the time-varying intramyocardial pressure experienced by the coronary arteries during the cardiac cycle. This differentiation has enabled the simulation of out-of-phase flow and pressure relationships that are characteristic of coronary circulation.

The numerical implementation has been validated against standard solution of ODEs describing respective LPNs obtained through MATLAB, demonstrating a high degree of consistency and accuracy. The backward finite differencing schemes employed for temporal discretization have shown robustness in solving the ordinary differential equations governing the LPN models. The integration of flow and pressure histories into the numerical framework has ensured stability and reliability in the computational predictions. Furthermore, the implementation has been structured to accommodate modularity, enabling future extensions and modifications to the boundary condition models.

The results have highlighted the suitability of the developed models for capturing the hemodynamic phenomena in larger and smaller blood vessels, as demonstrated by simple tutorial cases. The ability to simulate patient-specific conditions has been enhanced, providing a valuable tool for personalized treatment planning and the evaluation of surgical interventions. The implemented boundary conditions represent a substantial improvement over traditional methods in terms of physiological fidelity and numerical stability. This work contributes to the broader field of cardiovascular simulations by offering a robust methodology for integrating 0D and 3D modeling approaches in OpenFOAM.

This project has laid the foundations for the implementation of 0D LPN model boundary conditions for the patient-specific computational fluid dynamics (CFD) and fluid-structure interaction (FSI) simulations of coronary arteries in OpenFOAM, an open-source CFD software. While the developed boundary conditions have been tested by running laminar CFD simulations on simple cylindrical geometries representing aorta and left coronary artery (LCA). In next steps, a patient-specific simulation of a whole coronary artery tree should will be carried out where the boundary conditions from both the `windkesselOutletPressure` and `coronaryOutletPressure` family will be implemented for aortic and coronary outlets, respectively. Moreover, the suitability of developed boundary conditions in the FSI simulations using `solids4foam` environment will be assessed. The insights gained from this study would be used to facilitate further advancements in patient-specific

CFD and FSI simulations of coronary artery using open-source software. Future work could focus on extending the framework to include more detailed heart models, improving the computational efficiency of the implemented methods, and exploring the application of machine learning based techniques for the estimation and optimization of lumped parameter network (LPN) model parameters. The incorporation of additional physiological factors, such as vessel wall elasticity and microcirculatory dynamics, could further enhance the accuracy and clinical relevance of the simulations.

The latest updates on the ongoing development and implementation of a framework for the patient-specific CFD and FSI simulations of coronary arteries in OpenFOAM and `solids4foam` toolbox will be available on the author's GitHub profile (https://github.com/MA-Raza).

Bibliography

[1] P. Cardiff, Z. Tukovic, and I. Batistic, "solids4foam: An open-source toolbox for solid and fluid-solid interaction simulations," Jul. 2024.

[2] K. Sagawa, R. K. Lie, and J. Schaefer, "Translation of otto frank's paper "die grundform des arteriellen pulses (1899)"," *Journal of Molecular and Cellular Cardiology*, vol. 22, no. 3, pp. 253–254, 1990, doi: 10.1016/0022-2828(90)91459-K. [Online]. Available: https://doi.org/10.1016/0022-2828(90)91459-K

[3] N. Westerhof, J.-W. Lankhaar, and B. E. Westerhof, "The arterial windkessel," *Medical & Biological Engineering & Computing*, vol. 47, no. 2, pp. 131–141, 2009. [Online]. Available: https://doi.org/10.1007/s11517-008-0359-2

[4] S. Mantero, R. Pietrabissa, and R. Fumero, "The coronary bed and its role in the cardiovascular system: a review and an introductory single-branch model," *J. Biomed. Eng.*, vol. 14, 1992. [Online]. Available: https://doi.org/10.1016/0141-5425(92)90015-D

[5] H. J. Kim, I. E. Vignon-Clementel, J. S. Coogan, C. A. Figueroa, K. E. Jansen, and C. A. Taylor, "Patient-specific modeling of blood flow and pressure in human coronary arteries," *Ann Biomed Eng*, vol. 38, 2010. [Online]. Available: https://doi.org/10.1007/s10439-010-0083-6

[6] A. Updegrove, N. M. Wilson, J. Merkow, H. Lan, A. L. Marsden, and S. C. Shadden, "Simvascular: An open source pipeline for cardiovascular simulation," *Annals of Biomedical Engineering*, vol. 45, no. 3, pp. 525–541, 2017. [Online]. Available: https://doi.org/10.1007/s10439-016-1762-8

[7] C. J. Arthurs, R. Khlebnikov, A. Melville, M. Marčan, A. Gomez, D. Dillon-Murphy, F. Cuomo, M. Silva Vieira, J. Schollenberger, S. R. Lynch, C. Tossas-Betancourt, K. Iyer, S. Hopper, E. Livingston, P. Youssefi, A. Noorani, S. Ben Ahmed, F. J. H. Nauta, T. M. J. van Bakel, Y. Ahmed, P. A. J. van Bakel, J. Mynard, P. Di Achille, H. Gharahi, K. D. Lau, V. Filonova, M. Aguirre, N. Nama, N. Xiao, S. Baek, K. Garikipati, O. Sahni, D. Nordsletten, and C. A. Figueroa, "Crimson: An open-source software framework for cardiovascular integrated modelling and simulation," *PLOS Computational Biology*, vol. 17, no. 5, pp. 1–21, 05 2021. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1008881

[8] K. Kirali, *Coronary Artery Disease*. Rijeka: IntechOpen, Nov 2015. [Online]. Available: https://doi.org/10.5772/59455

[9] A. Squeri, *Coronary Artery Disease*. Rijeka: IntechOpen, Mar 2012. [Online]. Available: https://doi.org/10.5772/1168

[10] N. H. Pijls, B. de Bruyne, K. Peels, P. H. van der Voort, H. J. Bonnier, J. Bartunek, and J. J. Koolen, "Measurement of fractional flow reserve to assess the functional severity of coronary-artery stenoses," *New England Journal of Medicine*, vol. 334, no. 26, pp. 1703–1708, 1996. [Online]. Available: https://www.nejm.org/doi/full/10.1056/NEJM199606273342604

[11] D.-Y. Hwang, J.-M. Lee, and B.-K. Koo, "Physiologic assessment of coronary artery disease: Focus on fractional flow reserve," *Korean Journal of Radiology*, vol. 17, no. 3, pp. 307–320, 2016. [Online]. Available: https://doi.org/10.3348/kjr.2016.17.3.307

[12] S. Achenbach, T. Rudolph, J. Rieber, H. Eggebrecht, G. Richardt, T. Schmitz, N. Werner, F. Boenner, and H. Möllmann, "Performing and interpreting fractional flow reserve measurements in clinical practice: An expert consensus document," *Interventional Cardiology (London, England)*, vol. 12, no. 2, pp. 97–109, 2017. [Online]. Available: https://doi.org/10.15420/icr.2017:13:2

[13] M. N. Sheppard, *Practical cardiovascular pathology.* CRC Press, 2022.

[14] D. Tousoulis, *Coronary artery disease: From biology to clinical practice.* Academic Press, 2018.

[15] K. M. Chinnaiyan, T. Akasaka, T. Amano, J. J. Bax, P. Blanke, B. De Bruyne, T. Kawasaki, J. Leipsic, H. Matsuo, Y. Morino, K. Nieman, B. L. Norgaard, M. R. Patel, G. Pontone, M. Rabbat, C. Rogers, N. P. Sand, and G. Raff, "Rationale, design and goals of the heartflow assessing diagnostic value of non-invasive ffrct in coronary care (advance) registry," *Journal of Cardiovascular Computed Tomography*, vol. 11, no. 1, pp. 62–67, 2017. [Online]. Available: https://doi.org/10.1016/j.jcct.2016.12.002

[16] J. Chen, L. H. Wetzel, K. L. Pope, L. J. Meek, T. Rosamond, and C. M. Walker, "Ffrct: Current status," *American Journal of Roentgenology*, vol. 216, no. 3, pp. 640–648, 2021, pMID: 33377794. [Online]. Available: https://doi.org/10.2214/AJR.20.23332

[17] P. Rajiah, K. W. Cummings, E. Williamson, and P. M. Young, "Ct fractional flow reserve: A practical guide to application, interpretation, and problem solving," *RadioGraphics*, vol. 42, no. 2, pp. 340–358, 2022, pMID: 35119968. [Online]. Available: https://doi.org/10.1148/rg.210097

[18] Z. Sun and L. Xu, "Computational fluid dynamics in coronary artery disease," *Computerized Medical Imaging and Graphics*, vol. 38, no. 8, pp. 651–663, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0895611114001463

[19] O. Hajati, K. Zarrabi, R. Karimi, and A. Hajati, "Cfd simulation of hemodynamics in sequential and individual coronary bypass grafts based on multislice ct scan datasets," in *Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, ser. IEEE Engineering in Medicine and Biology Society. Annual International Conference, 2012, pp. 641–644. [Online]. Available: https://doi.org/10.1109/EMBC.2012.6346013

[20] W. Wu, A. N. Panagopoulos, C. H. Vasa, M. Sharzehee, S. Zhao, S. Samant, U. M. Oguz, B. Khan, A. Naser, K. M. Harmouch, G. S. Kassab, A. Siddique, and Y. S. Chatzizisis, "Patient-specific computational simulation of coronary artery bypass grafting," *PLOS ONE*, vol. 18, no. 3, pp. 1–17, 03 2023. [Online]. Available: https://doi.org/10.1371/journal.pone.0281423

[21] A. A. Owida, H. Do, and Y. S. Morsi, "Numerical analysis of coronary artery bypass grafts: An over view," *Computer Methods and Programs in Biomedicine*, vol. 108, no. 2, pp. 689–705, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169260711003221

[22] C. A. Taylor and C. A. Figueroa, "Patient-specific model of cardiovascular mechanics," *Annu. Rev. Biomed. Eng*, vol. 11, 2009. [Online]. Available: https://doi.org/10.1146/annurev.bioeng.10.061807.160521

[23] J. R. Cebral, M. A. Castro, J. E. Burgess, R. S. Pergolizzi, M. J. Sheridan, and C. M. Putman, "Characterization of cerebral aneurysms for assessing risk of rupture by using patient-specific computational hemodynamics models," *Am. J. Neuroradiol.*, vol. 26, 2005.

[24] F. J. H. Gijsen, J. J. Wentzel, A. Thury, F. Mastik, J. A. Schaar, J. C. H. Schuurbiers, C. J. Slager, W. J. Giessen, P. J. Feyter, A. F. W. Steen, and P. W. Serruys, "Strain distribution over plaques in human coronary arteries relates to shear stress," *Am. J. Physiol. Heart Circ. Physiol.*, vol. 295, 2008. [Online]. Available: https://doi.org/10.1152/ajpheart.01081.2007

[25] J. L. Berry, A. Santamarina, J. E. Moore, S. Roychowdhury, and W. D. Routh, "Experimental and computational flow evaluation of coronary stents," *Ann. Biomed. Eng.*, vol. 28, 2000. [Online]. Available: https://doi.org/10.1114/1.276

[26] Z. Li and C. Kleinstreuer, "Blood flow and structure interactions in a stented abdominal aortic aneurysm model," *Med. Eng. Phys.*, vol. 27, 2005. [Online]. Available: https://doi.org/10.1016/j.medengphy.2004.12.003

[27] F. Migliavacca, R. Balossino, G. Pennati, G. Dubini, T. Y. Hsia, M. R. Leval, and E. L. Bove, "Multiscale modelling in biofluidynamics: application to reconstructive paediatric cardiac surgery," *J. Biomech.*, vol. 39, 2006. [Online]. Available: https://doi.org/10.1016/j.jbiomech.2005.02.021

[28] C. A. Taylor, M. T. Draney, J. P. Ku, D. Parker, B. N. Steele, K. Wang, and C. K. Zarins, "Predictive medicine: computational techniques in therapeutic decision-making," *Comput. Aided Surg.*, vol. 4, 1999. [Online]. Available: https://doi.org/10.3109/10929089909148176

[29] L. H. Opie, *Heart Physiology: From Cell to Circulation.* Philadelphia, PA, USA: Lippincott Williams and Wilkins, 2003.

[30] Y. Qiu and J. M. Tarbell, "Numerical simulation of pulsatile flow in a compliant curved tube model of a coronary artery," *J. Biomech. Eng.*, vol. 122, 2000. [Online]. Available: https://doi.org/10.1115/1.429629

[31] D. Zeng, E. Boutsianis, M. Ammann, K. Boomsma, S. Wildermuth, and D. Poulikakos, "A study on the compliance of a right coronary artery and its impact on wall shear stress," *J. Biomech. Eng.*, vol. 130, 2008. [Online]. Available: https://doi.org/10.1115/1.2937744

[32] S. D. Ramaswamy, S. C. Vigmostad, A. Wahle, Y. G. Lai, M. E. Olszewski, K. C. Braddy, T. M. H. Brennan, J. D. Rossen, M. Sonka, and K. B. Chandran, "Fluid dynamic analysis in a human left anterior descending coronary artery with arterial motion," *Ann. Biomed. Eng.*, vol. 32, 2004. [Online]. Available: https://doi.org/10.1007/s10439-004-7816-3

[33] A. Santamarina, E. Weydahl, J. M. Siegel, and J. E. Moore, "Computational analysis of flow in a curved tube model of the coronary arteries: effects of time-varying curvature," *Ann. Biomed. Eng.*, vol. 26, 1998. [Online]. Available: https://doi.org/10.1114/1.113

[34] K. Lagana, R. Balossino, F. Migliavacca, G. Pennati, E. L. Bove, M. R. Leval, and G. Dubini, "Multiscale modeling of the cardiovascular system: application to the study of pulmonary and coronary perfusions in the univentricular circulation," *J. Biomech.*, vol. 38, 2005. [Online]. Available: https://doi.org/10.1016/j.jbiomech.2004.05.027

[35] H. J. Kim, C. A. Figueroa, T. J. R. Hughes, K. E. Jansen, and C. A. Taylor, "Augmented lagrangian method for constraining the shape of velocity profiles at outlet boundaries for three-dimensional finite element simulations of blood flow," *Comput. Methods Appl. Mech. Eng.*, vol. 198, 2009. [Online]. Available: https://doi.org/10.1016/j.cma.2009.02.012

[36] I. E. Vignon-Clementel, C. A. Figueroa, K. E. Jansen, and C. A. Taylor, "Outflow boundary conditions for three-dimensional finite element modeling of blood flow and pressure in arteries," *Comput. Methods Appl. Mech. Eng.*, vol. 195, 2006. [Online]. Available: https://doi.org/10.1016/j.cma.2005.04.014

[37] ——, "Outflow boundary conditions for three-dimensional simulations of non-periodic blood flow and pressure fields in deformable arteries," *Comput. Methods Biomech. Biomed. Eng.*, 2008. [Online]. Available: https://doi.org/10.1080/10255840903413565

[38] H. J. Kim, I. E. Vignon-Clementel, C. A. Figueroa, J. F. LaDisa, K. E. Jansen, J. A. Feinstein, and C. A. Taylor, "On coupling a lumped parameter heart model and a three-dimensional finite element aorta model," *Ann. Biomed. Eng.*, vol. 37, 2009. [Online]. Available: https://doi.org/10.1007/s10439-009-9760-8

[39] A. Piebalgs, "Openfoam-phys-flow," https://github.com/KeepFloyding/OpenFOAM-phys-flow, 2016, accessed: 2025-02-10.

[40] T. Schenkel and I. Halliday, "Continuum scale non-newtonian particle transport model for hæmorheology," *Mathematics*, vol. 9, no. 17, p. 2100, 2021. [Online]. Available: https://doi.org/10.3390/math9172100

[41] T. Schenkel, "haemofoam," https://github.com/TS-CUBED/haemoFoam, 2022, accessed: 2025-02-10.

[42] E. L. Manchester, S. Pirola, M. Y. Salmasi, D. P. O'Regan, T. Athanasiou, and X. Y. Xu, "Analysis of turbulence effects in a patient-specific aorta with aortic valve stenosis," *Cardiovascular Engineering and Technology*, vol. 12, pp. 438–453, 2021. [Online]. Available: https://doi.org/10.1007/s13239-021-00536-9

[43] A. Piebalgs, B. Gu, and E. Manchester, "Openfoam-v4-windkessel-code," https://github.com/EManchester/OpenFOAM-v4-Windkessel-code, 2021, accessed: 2025-02-10.

[44] E. L. Manchester, S. Pirola, M. Y. Salmasi, D. P. O'Regan, T. Athanasiou, and X. Y. Xu, "Openfoam-v8-windkessel-code," https://github.com/EManchester/OpenFOAM-v8-Windkessel-code, 2021, accessed: 2025-02-10.

[45] O. Frank, "The basic shape of the arterial pulse. first treatise: Mathematical analysis," *Journal of Molecular and Cellular Cardiology*, vol. 22, no. 3, pp. 255–277, 1990, doi: 10.1016/0022-2828(90)91460-O. [Online]. Available: https://doi.org/10.1016/0022-2828(90)91460-O

[46] T. Kind, T. J. C. Faes, J.-W. Lankhaar, A. Vonk-Noordegraaf, and M. Verhaegen, "Estimation of three- and four-element windkessel parameters using subspace model identification," *IEEE Transactions on Biomedical Engineering*, vol. 57, no. 7, pp. 1531–1538, 2010.

[47] P. Segers, E. R. Rietzschel, M. L. D. Buyzere, N. Stergiopulos, N. Westerhof, L. M. V. Bortel, T. Gillebert, and P. R. Verdonck, "Three- and four-element windkessel models: Assessment of their fitting performance in a large cohort of healthy middle-aged individuals," *Proceedings of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine*, vol. 222, no. 4, pp. 417–428, 2008, pMID: 18595354. [Online]. Available: https://doi.org/10.1243/09544119JEIM287

Study questions

1. What are lumped parameter models in the fluid dynamics of cardiovascular system?
2. What is windkessel effect in cardiovascular system?
3. What do different elements in a lumped parameter network (LPN) such as a resistor, a capacitor and an inductor represent?
4. What is a windkessel model?
5. What is the difference between a single-element resistive model, 2-element windkessel model?
6. What is the difference between a 2, 3, and 4-element windkessel models?
7. How the lumped parameter network (LPN) model of downstream coronary vascular beds is different from windkessel models?
8. What is the contribution of the constant distal pressure P_d term on the overall pressure obtained by the windkessel models or coronary LPN model?
9. Why does the intramyocardial pressure $P_{im}(t)$ added next to intramyocardial compliance C_{im} have to be time-variant?
10. Which pressure in the cardiovascular system can be used as intramyocardial pressure $P_{im}(t)$ in the coronary LPN model and how?
11. To solve the windkessel model, values of which parameters other than model elements (resistance, capacitor, inductance) are required, and how they can be accessed or calculated in OpenFOAM?
12. How the time-series data such as intramyocardial pressure $P_{im}(t)$ be read from a file in the OpenFOAM boundary condition source file?
13. How can a value of a parameter be estimated at any specific instant in time from its time-series data?
14. Like various other flow rates and pressures in the cardiovascular system, intramyocardial pressure $P_{im}(t)$ too is pulsatile and periodic. If $P_{im}(t)$ time-series data supplied through the file is only for one cycle, how can it be repeated for any number of cycles?
15. Why a lumped parameter network boundary condition should be used at the outlet instead of 0 Pa outlet pressure applied through `fixedValue`?

Appendix A

Lumped Parameter Network BC Codes

A.1 `windkesselOutletPressure` BC

The codes contained in the header and source files for `windkesselOutletPressure` boundary condition are listed in this section.

A.1.1 Header file

The header file `windkesselOutletPressureFvPatchScalarField.H` contains the description and declarations for the implemented windkessel model boundary conditions. The code contained in this source file is given below.

`windkesselOutletPressureFvPatchScalarField.H`

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.1
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

    foam-extend is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation, either version 3 of the License, or (at your
    option) any later version.

    foam-extend is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
    Public License for more details.

    You should have received a copy of the GNU General Public License
    along with foam-extend. If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::windkesselOutletPressureFvPatchScalarField

Group


Description
    A boundary condition that models outlet pressure using the Windkessel
```

```
framework, commonly used in hemodynamics simulations to replicate the
physiological behavior of arterial systems.

This boundary condition can simulate different Windkessel models of
varying complexity, including:

(1) Resistive (R): A simple resistive boundary where pressure drop is
    proportional to flow.

(2) WK2 (2-element Windkessel): A model with a distal resistance (Rd)
    and capacitance (C), commonly used to represent the compliance of
    arteries without considering proximal resistance.

(3) WK3 (3-element Windkessel): The classic 3-element model, which
    includes proximal resistance (Rp), distal resistance (Rd), and
    capacitance (C), and is the most widely used model in hemodynamics.

(4) WK4Series (4-element Windkessel with inductance in series): An
    extension of the 3-element model that adds inductance or inertance (L)
    accounting for the inertial effects of the blood in series with the
    proximal resistance (Rp), making it more closely match patient-specific
    scenarios.

(5) WK4Parallel (4-element Windkessel with inductance in parallel): Another
    extension of the 3-element model, where inductance or inertance (L) is
    placed in parallel to the proximal resistance (Rp), capturing additional
    dynamics related to flow fluctuations and inertial effects.

A distal pressure term (Pd) is added to all the models after the distal
resistance (Rd), making them applicable to cases where distal resistance
(Rd) is not grounded to zero pressure.

The Resistive (R) model is based on the following equation for the
relationship between pressure (P) and flow rate (Q):

    P(t) = Rd * Q(t) + Pd

Where:
    - Rd: Distal resistance
    - Pd: Distal pressure
    - Q(t): Flow rate
    - P(t): Pressure

The 2-element Windkessel (WK2) model is based on the following ordinary
differential equation (ODE) for the relationship between pressure (P) and
flow rate (Q):

    P(t) = Rd * Q(t)
      - Rd * C * dP/dt
      + Pd

Where:
    - Rd: Distal resistance
    - C: Compliance
    - Pd: Distal pressure
    - Q(t): Flow rate
    - P(t): Pressure

The 3-element Windkessel (WK3) model is based on the following ordinary
differential equation (ODE) for the relationship between pressure (P) and
flow rate (Q):

    P(t) = (Rp + Rd) * Q(t)
      + (C * Rd * Rp) * dQ/dt
      + Pd
      - (C * Rd) * dP/dt

Where:
```

```
        - Rd: Distal resistance
        - Rp: Proximal resistance
        - C: Compliance
        - Pd: Distal pressure
        - Q(t): Flow rate
        - P(t): Pressure

    The 4-element Windkessel model with inductance in series (WK4Series) is
    based on the following ordinary differential equation (ODE) for the
    relationship between pressure (P) and flow rate (Q):

        P(t) = (Rp + Rd) * Q(t)
          + (L + Rp * Rd * C) * dQ/dt
          + Rd * C * L * d^2Q/dt^2
          + Pd
          - Rd * C * dP/dt

    Where:
        - Rd: Distal resistance
        - Rp: Proximal resistance
        - C: Compliance
        - L: Inductance or inertance
        - Pd: Distal pressure
        - Q(t): Flow rate
        - P(t): Pressure

    The 4-element Windkessel model with inductance in parallel (WK4Parallel) is
    based on the following ordinary differential equation (ODE) for the
    relationship between pressure (P) and flow rate (Q):

        P(t) = Rd * Q(t)
            + L * (1 + Rd / Rp) * dQ/dt
            + Rd * C * L * d^2Q/dt^2
            + Pd
            - ((L + Rp * Rd * C) / Rp) * dP/dt
            - (Rd * C * L / Rp) * d^2P/dt^2

    Where:
        - Rd: Distal resistance
        - Rp: Proximal resistance
        - C: Compliance
        - L: Inductance or inertance
        - Pd: Distal pressure
        - Q(t): Flow rate
        - P(t): Pressure

    This boundary condition calculates the outlet pressure based on volumetric
    flow rates and the Windkessel parameters using either second-order backward
    differencing scheme or a simpler first-order backward differencing approach.

    This ODE is discretized using either a second-order backward Euler
    scheme or a first-order scheme. The time derivative terms (dP/dt, dQ/dt,
    d^2P/dt^2, and d^2Q/dt^2) are handled differently depending on the selected
    differencing scheme.

    For first-order backward differencing scheme:

        dx/dt = (xn - xo) / dt

    and

        d^2x/dt^2 = (xn - 2 * xo + xoo) / (dt^2)

    Whereas, for second-order backward differencing scheme:

        dx/dt = (3*xn - 4*xo + xoo) / (2 * dt)

    and
```

```
        d^2x/dt^2 = (2 * xn - 5 * xo + 4 * xoo - xooo) / (dt^2)

    The boundary condition computes the pressure (Pn) at each time
    step based on historical values of pressure and flow rates,
    ensuring the boundary accurately models physiological behavior.

Key Features
    - Supports both first-order and second-order backward differencing scheme for
      pressure updates.
    - Updates historical pressure and flow rate values to reflect Windkessel dynamics.
    - Includes parameters such as Rp, Rd, Rc, L, and Pd, which are customizable
      through a dictionary.

Usage
    Depending on the model, specify this boundary condition in the boundary file (0/p) as
    follows:

        <patchName>
        {
        type                windkesselOutletPressure;
        windkesselModel     <Resistive, WK2, WK3, WK4Series or WK4Parallel>; // Type of model, Default
     : WK3
        Rp                  <value>;    // Proximal resistance in [kgm^-4s^-1], Default: 1
        Rd                  <value>;    // Distal resistance in [kgm^-4s^-1], Default: 1
        C                   <value>;    // Capacitance or compliance in [m^4s^2kg^-1], Default: 1
        L                   <value>;    // Inductance or inertance in [kgm^-4], Default: 1
        Pd                  <value>;    // Distal pressure in [Pa], Default: 0
        rho                 <value>;    // Fluid density in [kgm^-3], Default: 1
        diffScheme          <firstOrder or secondOrder>; // Differencing scheme, Default: secondOrder
        value               uniform 0;
        }

Note
    1) For the Resistive (R) boundary condition set windkesselModel = Resistive,
       and give values of Rd and Pd.
    2) For the 2-element Windkessel (WK2) model set windkesselModel = WK2, and give
       values of Rd, C, and Pd. Choose diffScheme.
    3) For the 3-element Windkessel (WK3) model set windkesselModel = WK3, and give
       values of Rp, Rd, C, and Pd. Choose diffScheme.
    4) For the 4-element Windkessel (WK4Series) model with series inductance set
       windkesselModel = WK4Series, and give values of Rp, Rd, C, L, and Pd. Choose
       diffScheme.
    5) For the 4-element Windkessel (WK4Parallel) model with parallel inductance set
       windkesselModel = WK4Parallel, and give values of Rp, Rd, C, L, and Pd. Choose
       diffScheme.

Author
    Muhammad Ahmad Raza, University College Dublin, Ireland.

Cite as
    Raza, M. A.: Implementation of Lumped Parameter Network Boundary Conditions for the
    Patient-Specific CFD Simulations of Coronary Arteries in OpenFOAM. In Proceedings of CFD with
    OpenSource Software, 2024, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2024

See also
    Foam::fixedValueFvPatchField
    Foam::fvPatchField

SourceFiles
    windkesselOutletPressureFvPatchScalarField.C

\*---------------------------------------------------------------------------*/

#ifndef windkesselOutletPressureFvPatchScalarField_H
#define windkesselOutletPressureFvPatchScalarField_H

#include "fixedValueFvPatchFields.H"
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

//- Enumerator for supported windkessel model types
enum WindkesselModel
{
    Resistive = 0,     // Simple resistive model (R)
    WK2 = 1,           // 2-element Windkessel model (RC)
    WK3 = 2,           // 3-element Windkessel model (RCR)
    WK4Series = 3,     // 4-element Windkessel with inductance in series (RCRL)
    WK4Parallel = 4    // 4-element Windkessel with inductance in parallel
};

//- Enumerator for for time differencing schemes
enum DifferencingScheme
{
    firstOrder = 0,    // First-order backward differencing scheme
    secondOrder = 1    // Second-order backward differencing scheme
};

/*---------------------------------------------------------------------------*\
           Class windkesselOutletPressureFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

/**
 * @brief Boundary condition implementing hemodynamic outlet pressure
 *        using windkessel models.
 *
 * This class models outlet pressure as a function of volumetric flow rate
 * using the windkessel framework. The framework includes models of varying
 * complexity:
 *
 * - Resistive (R)
 * - 2-element Windkessel (WK2 or RC)
 * - 3-element Windkessel (WK3 or RCR)
 * - 4-element Windkessel with inductance in series (WK4Series or RCRL-Series)
 * - 4-element Windkessel with inductance in parallel (WK4Parallel or RCRL-Parallel)
 *
 * The models incorporate parameters such as proximal resistance (Rp), distal
 * resistance (Rd), compliance (C), inductance (L), and distal pressure (Pd).
 * The boundary condition supports explicit and implicit formulations for time
 * stepping and allows customization of differencing schemes and parameter settings.
 */

class windkesselOutletPressureFvPatchScalarField
:
    public fixedValueFvPatchScalarField
{

private:

    // Private data

        //- Windkessel parameters
        scalar Rp_;     // Proximal resistance (Rp)
        scalar Rd_;     // Distal resistance (Rd)
        scalar C_;      // Compliance or capacitance (C)
        scalar L_;      // Inductance (L)
        scalar Pd_;     // Distal pressure (Pd)
        scalar rho_;    // Fluid density (rho)

        //- Windkessel state variables
        scalar Pooo_;   // Pressure three time steps ago
        scalar Poo_;    // Pressure two time steps ago
        scalar Po_;     // Previous time-step pressure
```

```
        scalar Pn_;     // Current pressure
        scalar Qooo_;   // Flow rate three time steps ago
        scalar Qoo_;    // Flow rate two time steps ago
        scalar Qo_;     // Previous time-step flow rate
        scalar Qn_;     // Current flow rate

        //- Windkessel LPN configuration
        WindkesselModel windkesselModel_;   // Selected Windkessel model type
        DifferencingScheme diffScheme_;     // Selected time differencing scheme

        //- Time index
        label timeIndex_;   // Tracks the simulation time index

public:

    //- Runtime type information
    TypeName("windkesselOutletPressure");

    // Constructors

        //- Construct from patch and internal field
        windkesselOutletPressureFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct from patch, internal field, and dictionary
        windkesselOutletPressureFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

        //- Map existing object onto a new patch
        windkesselOutletPressureFvPatchScalarField
        (
            const windkesselOutletPressureFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Copy constructor
        windkesselOutletPressureFvPatchScalarField
        (
            const windkesselOutletPressureFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new windkesselOutletPressureFvPatchScalarField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        windkesselOutletPressureFvPatchScalarField
        (
            const windkesselOutletPressureFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
```

```
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new windkesselOutletPressureFvPatchScalarField(*this, iF)
            );
        }

    // Member functions

        //- Evaluation function to update coefficients for the boundary
        //  condition based on flow rate and model parameters
        virtual void updateCoeffs();

        //- Write boundary condition data to output stream
        virtual void write(Ostream&) const;
};

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#endif

// ************************************************************************* //
```

A.1.2 Source file

The source file `windkesselOutletPressureFvPatchScalarField.C` contains the main implementation of the windkessel model boundary conditions. The code contained in this source file is given below.

`windkesselOutletPressureFvPatchScalarField.C`

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.1
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

    foam-extend is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation, either version 3 of the License, or (at your
    option) any later version.

    foam-extend is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with foam-extend.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "windkesselOutletPressureFvPatchScalarField.H"
#include "addToRunTimeSelectionTable.H"
#include "fvPatchFieldMapper.H"
```

```
#include "volFields.H"
#include "surfaceFields.H"
#include "backwardDdtScheme.H"
#include "word.H"

// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * * //

// (No additional details required for static members here)

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * //

//- Constructor: Initialize with default values
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(p, iF), // Call base class constructor
    Rp_(1),     // Proximal resistance
    Rd_(1),     // Distal resistance
    C_(1),      // Compliance
    L_(1),      // Inductance
    Pd_(0),     // Distal pressure
    rho_(1),    // Fluid density
    Pooo_(0), Poo_(0), Po_(0), Pn_(0), // Initialize pressures to zero
    Qooo_(0), Qoo_(0), Qo_(0), Qn_(0), // Initialize flow rates to zero
    windkesselModel_(WK3),     // Default windkessel model
    diffScheme_(secondOrder),  // Default differencing scheme
    timeIndex_(-1)  // Initialize time index to invalid
{}

//- Constructor: Initialize from dictionary (typically used in user input)
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF, dict),      // Call base class constructor
    Rp_(dict.lookupOrDefault<scalar>("Rp", 1)),     // Proximal resistance
    Rd_(dict.lookupOrDefault<scalar>("Rd", 1)),     // Distal resistance
    C_(dict.lookupOrDefault<scalar>("C", 1)),       // Compliance
    L_(dict.lookupOrDefault<scalar>("L", 1)),       // Inductance
    Pd_(dict.lookupOrDefault<scalar>("Pd", 0)),     // Distal pressure
    rho_(dict.lookupOrDefault<scalar>("rho", 1)),   // Fluid density
    Pooo_(dict.lookupOrDefault<scalar>("Pooo", 0)),
    Poo_(dict.lookupOrDefault<scalar>("Poo", 0)),
    Po_(dict.lookupOrDefault<scalar>("Po", 0)),
    Pn_(dict.lookupOrDefault<scalar>("Pn", 0)),
    Qooo_(dict.lookupOrDefault<scalar>("Qooo", 0)),
    Qoo_(dict.lookupOrDefault<scalar>("Qoo", 0)),
    Qo_(dict.lookupOrDefault<scalar>("Qo", 0)),
    Qn_(dict.lookupOrDefault<scalar>("Qn", 0)),
    timeIndex_(-1)  // Time index reset
{

    Info << "\n\nApplying windkesselOutletPressure BC on patch: " << patch().name() << endl;

    //- Retrieve the model as a string and map to the enumerator
    word WKModelStr = dict.lookupOrDefault<word>("windkesselModel", "WK3");

    if (WKModelStr == "Resistive")
    {
        windkesselModel_ = Resistive;
```

```
        //- Print Windkessel model properties for debugging/verification
        Info << "\n\nProperties of Windkessel Model: \n"
             << "Model Type: 1-Element (Resistive) \n"
             << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
             << "Distal Pressure: Pd = " << Pd_ << " Pa \n"
             << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
    }
    else if (WKModelStr == "WK2")
    {
        windkesselModel_ = WK2;

        //- Print Windkessel model properties for debugging/verification
        Info << "\n\nProperties of Windkessel Model: \n"
             << "Model Type: 2-Element (RC) \n"
             << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
             << "Compliance: C = " << Rd_ << " m^4s^2kg^-1 \n"
             << "Distal Pressure: Pd = " << C_ << " Pa \n"
             << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
    }
    else if (WKModelStr == "WK3")
    {
        windkesselModel_ = WK3;

        //- Print Windkessel model properties for debugging/verification
        Info << "\n\nProperties of Windkessel Model: \n"
             << "Model Type: 3-Element (RCR) \n"
             << "Proximal Resistance: Rp = " << Rp_ << " kgm^-4s^-1 \n"
             << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
             << "Compliance: C = " << C_ << " m^4s^2kg^-1 \n"
             << "Distal Pressure: Pd = " << Pd_ << " Pa \n"
             << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
    }
    else if (WKModelStr == "WK4Series")
    {
        windkesselModel_ = WK4Series;

        //- Print Windkessel model properties for debugging/verification
        Info << "\n\nProperties of Windkessel Model: \n"
             << "Model Type: 4-Element (RCRL with L in Series to Rp) \n"
             << "Proximal Resistance: Rp = " << Rp_ << " kgm^-4s^-1 \n"
             << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
             << "Compliance: C = " << C_ << " m^4s^2kg^-1 \n"
             << "Inertance: L = " << L_ << " kgm^-4 \n"
             << "Distal Pressure: Pd = " << Pd_ << " Pa \n"
             << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
    }
    else if (WKModelStr == "WK4Parallel")
    {
        windkesselModel_ = WK4Parallel;

        //- Print Windkessel model properties for debugging/verification
        Info << "\n\nProperties of Windkessel Model: \n"
             << "Model Type: 4-Element (RCRL with L in Parallel to Rp) \n"
             << "Proximal Resistance: Rp = " << Rp_ << " kgm^-4s^-1 \n"
             << "Distal Resistance: Rd = " << Rd_ << " kgm^-4s^-1 \n"
             << "Compliance: C = " << C_ << " m^4s^2kg^-1 \n"
             << "Inertance: L = " << L_ << " kgm^-4 \n"
             << "Distal Pressure: Pd = " << Pd_ << " Pa \n"
             << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
    }
    else
    {
        FatalErrorInFunction << "\n\nUnknown Windkessel Model: " << WKModelStr
                             << "\nValid Windkessel Models (windkesselModel) are : \n\n"
                             << " 5 \n ( \n Resistive \n WK2 \n WK3 \n WK4Series \n WK4Parallel \n ) \
    n"
                             << exit(FatalError);
```

```
    }

    //- Retrieve the scheme as a string and map to the enumerator
    word schemeStr = dict.lookupOrDefault<word>("diffScheme", "secondOrder");

    Info << "Differencing Scheme for Windkessel Model: " << schemeStr << endl;

    if (schemeStr == "firstOrder")
    {
        diffScheme_ = firstOrder;
    }
    else if (schemeStr == "secondOrder")
    {
        diffScheme_ = secondOrder;
    }
    else
    {
        FatalErrorInFunction << "\n\nUnknown Differencing Scheme: " << schemeStr
                             << "\nValid Differencing Schemes (diffScheme) are : \n\n"
                             << " 2 \n ( \n firstOrder \n secondOrder \n ) \n"
                             << exit(FatalError);
    }

}

//- Constructor: Map an existing field onto a new patch
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const windkesselOutletPressureFvPatchScalarField& ptf, // Existing field
    const fvPatch& p, // New patch to map onto
    const DimensionedField<scalar, volMesh>& iF, // Internal field reference
    const fvPatchFieldMapper& mapper // Field mapper for mapping operations
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper), // Map base class properties
    Rp_(ptf.Rp_),     // Copy proximal resistance parameter
    Rd_(ptf.Rd_),     // Copy distal resistance parameter
    C_(ptf.C_),       // Copy compliance parameter
    L_(ptf.L_),       // Copy inertance parameter
    Pd_(ptf.Pd_),     // Copy distal pressure parameter
    rho_(ptf.rho_),   // Copy fluid density
    Pooo_(ptf.Pooo_), // Copy previous state pressure variables
    Poo_(ptf.Poo_),
    Po_(ptf.Po_),
    Pn_(ptf.Pn_),
    Qooo_(ptf.Qooo_), // Copy previous state flow rate variables
    Qoo_(ptf.Qoo_),
    Qo_(ptf.Qo_),
    Qn_(ptf.Qn_),
    windkesselModel_(ptf.windkesselModel_), // Copy Windkessel model type
    diffScheme_(ptf.diffScheme_), // Copy numerical differencing scheme
    timeIndex_(ptf.timeIndex_) // Copy time index
{}

//- Copy constructor: Create a duplicate field with all properties
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const windkesselOutletPressureFvPatchScalarField& wkpsf // Source field
)
:
    fixedValueFvPatchScalarField(wkpsf), // Copy base class properties
    Rp_(wkpsf.Rp_),     // Copy proximal resistance
    Rd_(wkpsf.Rd_),     // Copy distal resistance
    C_(wkpsf.C_),       // Copy compliance
    L_(wkpsf.L_),       // Copy inertance
    Pd_(wkpsf.Pd_),     // Copy distal pressure
```

```
    rho_(wkpsf.rho_),   // Copy fluid density
    Pooo_(wkpsf.Pooo_), // Copy pressure variables
    Poo_(wkpsf.Poo_),
    Po_(wkpsf.Po_),
    Pn_(wkpsf.Pn_),
    Qooo_(wkpsf.Qooo_), // Copy flow rate variables
    Qoo_(wkpsf.Qoo_),
    Qo_(wkpsf.Qo_),
    Qn_(wkpsf.Qn_),
    windkesselModel_(wkpsf.windkesselModel_), // Copy Windkessel model type
    diffScheme_(wkpsf.diffScheme_), // Copy differencing scheme
    timeIndex_(wkpsf.timeIndex_) // Copy time index
{}

//- Copy constructor with new internal field reference
Foam::windkesselOutletPressureFvPatchScalarField::
windkesselOutletPressureFvPatchScalarField
(
    const windkesselOutletPressureFvPatchScalarField& wkpsf, // Source field
    const DimensionedField<scalar, volMesh>& iF // New internal field
)
:
    fixedValueFvPatchScalarField(wkpsf, iF), // Map base class properties
    Rp_(wkpsf.Rp_),     // Copy proximal resistance
    Rd_(wkpsf.Rd_),     // Copy distal resistance
    C_(wkpsf.C_),       // Copy compliance
    L_(wkpsf.L_),       // Copy inertance
    Pd_(wkpsf.Pd_),     // Copy distal pressure
    rho_(wkpsf.rho_),   // Copy fluid density
    Pooo_(wkpsf.Pooo_), // Copy pressure variables
    Poo_(wkpsf.Poo_),
    Po_(wkpsf.Po_),
    Pn_(wkpsf.Pn_),
    Qooo_(wkpsf.Qooo_), // Copy flow rate variables
    Qoo_(wkpsf.Qoo_),
    Qo_(wkpsf.Qo_),
    Qn_(wkpsf.Qn_),
    windkesselModel_(wkpsf.windkesselModel_), // Copy Windkessel model type
    diffScheme_(wkpsf.diffScheme_), // Copy differencing scheme
    timeIndex_(wkpsf.timeIndex_) // Copy time index
{}


// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //

//- Update coefficients for boundary condition
void Foam::windkesselOutletPressureFvPatchScalarField::updateCoeffs()
{
    //- Check if coefficients have already been updated for this time step
    if (updated())
    {
        return;
    }

    //- Retrieve the time step size from the database
    const scalar dt = db().time().deltaTValue();

    //- Retrieve the flux field (phi) from the database
    const surfaceScalarField& phi =
        db().lookupObject<surfaceScalarField>("phi");

    //- Retrieve the boundary pressure field from the database
    const fvPatchField<scalar>& p =
        db().lookupObject<volScalarField>("p").boundaryField()[this->patch().index()];

    //- Calculate the total area of the patch
    scalar area = gSum(patch().magSf());
```

```
    //- Update pressure and flow rate histories if the time index has advanced
    if (db().time().timeIndex() != timeIndex_)
    {
        timeIndex_ = db().time().timeIndex();

        //- Update pressure history variables
        Pooo_ = Poo_;
        Poo_ = Po_;
        Po_ = Pn_;
        Pn_ = gSum(p * patch().magSf()) / area; // Compute mean pressure over the patch

        //- Update flow rate history variables
        Qooo_ = Qoo_;
        Qoo_ = Qo_;
        Qo_ = Qn_;
        Qn_ = gSum(phi.boundaryField()[patch().index()]); // Compute total flux
    }

    //- Calculate new pressure using backward differencing scheme
    if
    (
        db().time().timeIndex()>1 // Perform calculations only if the time index is greater than 1
    )
    {
        //- Switch to the specified windkessel model
        switch (windkesselModel_)
        {
            case Resistive: // Single-element resistive model (R)

                Pn_ = Rd_ * Qn_ + Pd_;

                break;

            case WK2: // 2-element windkessel model (RC)

                //- Switch to the specified differencing scheme
                switch (diffScheme_)
                {
                    case firstOrder:

                        Pn_ = Rd_ * Qn_
                            + Pd_
                            + Rd_ * C_ * Po_ / dt;

                        Pn_ /= (1.0 + Rd_ * C_ / dt) + SMALL;

                        break;

                    case secondOrder:

                        Pn_ = Rd_ * Qn_
                            + Pd_
                            - Rd_ * C_ * ((Poo_ - 4 * Po_) / (2 * dt));

                        Pn_ /= (1.0 + 3 * Rd_ * C_ / (2 * dt)) + SMALL;

                        break;

                    default:

                        FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
                }

                break;

            case WK3: // 3-element windkessel model (RCR)

                //- Switch to the specified differencing scheme
```

```
                switch (diffScheme_)
                {
                    case firstOrder:

                        Pn_ = Rp_ * Rd_ * C_ * ((Qn_ - Qo_) / dt)
                            + (Rp_ + Rd_) * Qn_
                            + Pd_
                            + Rd_ * C_ * (Po_ / dt);

                        Pn_ /= (1.0 + Rd_ * C_ / dt) + SMALL;

                        break;

                    case secondOrder:

                        Pn_ = Rp_ * Rd_ * C_ * ((3 * Qn_ - 4 * Qo_ + Qoo_) / (2 * dt))
                            + (Rp_ + Rd_) * Qn_
                            + Pd_
                            - Rd_ * C_ * ((Poo_ - 4 * Po_) / (2 * dt));

                        Pn_ /= (1.0 + 3 * Rd_ * C_  / (2 * dt)) + SMALL;

                        break;

                    default:

                        FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
                }

                break;

            case WK4Series: // 4-element series windkessel model (RCRL-Series)

                //- Switch to the specified differencing scheme
                switch (diffScheme_)
                {
                    case firstOrder:

                        Pn_ = (Rp_ + Rd_) * Qn_
                            + (L_ + Rp_ * Rd_ * C_) * ((Qn_ - Qo_) / dt)
                            + Rd_ * C_ * L_ * ((Qn_ - 2 * Qo_ + Qoo_) / (pow (dt, 2)))
                            + Pd_
                            + Rd_ * C_ * (Po_ / dt);

                        Pn_ /= (1.0 + Rd_ * C_ / dt) + SMALL;

                        break;

                    case secondOrder:

                        Pn_ = (Rp_ + Rd_) * Qn_
                            + (L_ + Rp_ * Rd_ * C_) * ((3 * Qn_ - 4 * Qo_ + Qoo_) / (2 * dt))
                            + Rd_ * C_ * L_ * ((2 * Qn_ - 5 * Qo_ + 4 * Qoo_ - Qooo_) / (pow (dt, 2)))
                            + Pd_
                            - Rd_ * C_ * ((Poo_ - 4 * Po_) / (2 * dt));

                        Pn_ /= (1.0 + 3 * Rd_ * C_  / (2 * dt)) + SMALL;

                        break;

                    default:

                        FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
                }

                break;

            case WK4Parallel: // 4-element parallel windkessel model (RCRL-Parallel)
```

```
                //- Switch to the specified differencing scheme
                switch (diffScheme_)
                {
                    case firstOrder:

                        Pn_ = Rd_ * Qn_
                            + L_ * (1 + Rd_ / Rp_ ) * ((Qn_ - Qo_) / dt)
                            + Rd_ * C_ * L_ * ((Qn_ - 2 * Qo_ + Qoo_) / (pow (dt, 2)))
                            + Pd_
                            + ((L_ + Rp_ * Rd_ * C_) / Rp_) * (Po_ / dt)
                            + (Rd_ * C_ * L_ / Rp_) * ((2 * Po_ - Poo_) / (pow (dt, 2)));

                        Pn_ /= (1.0 + ((L_ + Rp_ * Rd_ * C_) / (Rp_ * dt)) + (Rd_ * C_ * L_ / (Rp_ *
    pow (dt, 2)))) + SMALL;

                        break;

                    case secondOrder:

                        Pn_ = Rd_ * Qn_
                            + L_ * (1 + Rd_ / Rp_ ) * ((3 * Qn_ - 4 * Qo_ + Qoo_) / (2 * dt))
                            + Rd_ * C_ * L_ * ((2 * Qn_ - 5 * Qo_ + 4 * Qoo_ - Qooo_) / (pow (dt, 2)))
                            + Pd_
                            - ((L_ + Rp_ * Rd_ * C_) / Rp_) * ((Poo_ - 4 * Po_) / (2 * dt))
                            - (Rd_ * C_ * L_ / Rp_) * ((-5 * Po_ + 4 * Poo_ - Pooo_) / (pow (dt, 2)));

                        Pn_ /= (1.0 + (3 * (L_ + Rp_ * Rd_ * C_) / (2 * Rp_ * dt)) + (2 * Rd_ * C_ *
    L_ / (Rp_ * pow (dt, 2)))) + SMALL;

                        break;

                    default:

                        FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
                }

                break;

            default:

                FatalErrorInFunction << "Unknown Windkessel Model!" << exit(FatalError);
        }

    }

    //- Apply implicit pressure update to the boundary field
    operator==(Pn_ / (rho_ + SMALL));

    //- Call base class function to finalize the update
    fixedValueFvPatchScalarField::updateCoeffs();
}

//- Write the boundary condition properties to an output stream
void Foam::windkesselOutletPressureFvPatchScalarField::write(Ostream& os) const
{
    //- Call the base class function to write common boundary field properties
    fvPatchScalarField::write(os);

    //- Write the common windkessel model details to the output stream
    os.writeKeyword("Rd") << Rd_ << token::END_STATEMENT << nl; // Resistance parameter
    os.writeKeyword("Pd") << Pd_ << token::END_STATEMENT << nl; // Prescribed pressure
    os.writeKeyword("rho") << rho_ << token::END_STATEMENT << nl; // Fluid density

    //- Write the specific windkessel model details to the output stream
    switch (windkesselModel_)
    {
        case Resistive:
```

```
            os.writeKeyword("windkesselModel") << "Resistive" << token::END_STATEMENT << nl;
            break;

        case WK2:
            os.writeKeyword("windkesselModel") << "WK2" << token::END_STATEMENT << nl;
            os.writeKeyword("C") << C_ << token::END_STATEMENT << nl; // Compliance
            break;

        case WK3:
            os.writeKeyword("windkesselModel") << "WK3" << token::END_STATEMENT << nl;
            os.writeKeyword("Rp") << Rp_ << token::END_STATEMENT << nl; // Proximal resistance
            os.writeKeyword("C") << C_ << token::END_STATEMENT << nl; // Compliance
            break;

        case WK4Series:
            os.writeKeyword("windkesselModel") << "WK4Series" << token::END_STATEMENT << nl;
            os.writeKeyword("Rp") << Rp_ << token::END_STATEMENT << nl;
            os.writeKeyword("C") << C_ << token::END_STATEMENT << nl;
            os.writeKeyword("L") << L_ << token::END_STATEMENT << nl; // Inductance
            break;

        case WK4Parallel:
            os.writeKeyword("windkesselModel") << "WK4Parallel" << token::END_STATEMENT << nl;
            os.writeKeyword("Rp") << Rp_ << token::END_STATEMENT << nl;
            os.writeKeyword("C") << C_ << token::END_STATEMENT << nl;
            os.writeKeyword("L") << L_ << token::END_STATEMENT << nl;
            break;

        default:
            FatalErrorInFunction << "Unknown Windkessel Model: " << windkesselModel_ << exit(
    FatalError);
    }

    //- Write the differencing scheme used for calculations
    switch (diffScheme_)
    {
        case firstOrder:
            os.writeKeyword("diffScheme") << "firstOrder" << token::END_STATEMENT << nl;
            break;

        case secondOrder:
            os.writeKeyword("diffScheme") << "secondOrder" << token::END_STATEMENT << nl;
            break;

        default:
            FatalErrorInFunction << "Unknown differencing scheme: " << diffScheme_ << exit(FatalError)
    ;
    }

    //- Write flow history parameters
    os.writeKeyword("Qooo") << Qooo_ << token::END_STATEMENT << nl; // Third past flow rate
    os.writeKeyword("Qoo") << Qoo_ << token::END_STATEMENT << nl; // Second past flow rate
    os.writeKeyword("Qo") << Qo_ << token::END_STATEMENT << nl; // Previous flow rate
    os.writeKeyword("Qn") << Qn_ << token::END_STATEMENT << nl; // Current flow rate

    //- Write pressure history parameters
    os.writeKeyword("Pooo") << Pooo_ << token::END_STATEMENT << nl; // Third past pressure
    os.writeKeyword("Poo") << Poo_ << token::END_STATEMENT << nl; // Second past pressure
    os.writeKeyword("Po") << Po_ << token::END_STATEMENT << nl; // Previous pressure
    os.writeKeyword("Pn") << Pn_ << token::END_STATEMENT << nl; // Current pressure

    //- Write the boundary field value entry to the output stream
    writeEntry("value", os);
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
```

```
{
    //- Define the boundary field type for this class
    makePatchTypeField
    (
        fvPatchScalarField,                          // Base class
        windkesselOutletPressureFvPatchScalarField  // Derived class
    );
}

// ************************************************************************* //
```

A.2 coronaryOutletPressure BC

The codes contained in the header and source files for `coronaryOutletPressure` boundary condition are listed in this section.

A.2.1 Header file

The header file `coronaryOutletPressureFvPatchScalarField.H` contains the description and declarations for the coronary lumped parameter network (LPN) model boundary condition. The code contained in this source file is given below.

coronaryOutletPressureFvPatchScalarField.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.1
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

    foam-extend is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation, either version 3 of the License, or (at your
    option) any later version.

    foam-extend is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with foam-extend.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::coronaryOutletPressureFvPatchScalarField

Group


Description
    A boundary condition that models outlet pressure using the lumped parameter
    network (LPN) model of downstream coronary vascular beds, specifically
    designed for simulating hemodynamics in coronary artery systems. The LPN
    model is commonly used in cardiovascular simulations to capture the complex
    interaction between the arterial system and various lumped components representing
    resistance, compliance, and pressure dynamics.

    The model uses a system of ordinary differential equations (ODEs) to relate
    pressure (P) and flow rate (Q) at the boundary, incorporating parameters
    like arterial resistance (Ra), micro-arterial resistance (Ram), arterial compliance
```

```
    (Ca), intramyocardial compliance (Cim), intramyocardial pressure (Pim), veinous
    resistance (Rv) and micro-veinous resistance (Rvm). Moreover, a distal pressure term
    (Pv) is added distal to Rv on veinous side. The veinous compliance (Cv) is not considered
    to simplify the numerics. The LPN model used have been shown in the literature to produce
    the characteristic coronary flow and pressure curve efficiently.

    The system of ordinary differential equations (ODEs) relating pressure (P) and flow rate
    (Q) for the LPN model of downstream coronary vascular beds is given as follows:

        (1)
        Pi(t) = (Rvm + Rv) * Q(t)
            - (Rvm + Rv) * Ca * dP/dt
            + (Rvm + Rv) * Cim * dPim/dt
            + Pv
            - (Rvm + Rv) * Cim * dPi/dt

        (2)
        P(t) = (Ram + Ra) * Q(t)
            + Ram * Ra * Ca * dQ/dt
            + Pi(t)
            - Ram * Ca * dP/dt

    Where:
        - Ra:    Arterial resistance
        - Ram:   Micro-arterial resistance
        - Rv:    Veinous resistance
        - Rvm:   Micro-veinous resistance
        - Ca:    Arterial compliance
        - Cim:   Intramyocardial compliance
        - Pim:   Intramyocardial pressure
        - Pv:    Pressure distal to Rv
        - Q(t):  Flow rate
        - Pi(t): Intermediate pressure
        - P(t):  Coronary outlet pressure

    This boundary condition calculates the coronary outlet pressure based on
    volumetric flow rates and LPN model parameters using either second-order
    backward differencing scheme or a simpler first-order backward differencing
    approach.

    These ODEs are discretized using either a second-order backward Euler
    scheme or a first-order scheme. The time derivative terms (dP/dt, dQ/dt,
    dPi/dt, and dPim/dt) are handled differently depending on the selected
    differencing scheme.

    For first-order backward differencing scheme:

        dx/dt = (xn - xo) / dt

    Whereas, for second-order backward differencing scheme:

        dx/dt = (3*xn - 4*xo + xoo) / (2 * dt)

    The boundary condition computes the pressure (P) at each time
    step based on historical values of pressure and flow rates,
    ensuring the boundary accurately models physiological behavior.

Key Features
    - Supports both first-order and second-order backward differencing scheme for
      pressure updates.
    - Updates historical pressure and flow rate values to reflect LPN dynamics.
    - Includes parameters such as Ra, Ram, Ca, Cim, Rvm, Rv, and Pv which are
      customizable through a dictionary.
    - The intramyocardial pressure time-series data (Pim(t)) can be provided through
      a file.
    - A dictionary entery (PimScaling) can be used to scale the provide Pim(t) data.
      This is particularly useful for the cases where either left or right ventricle
      pressure is used as Pim(t), and the multiplication with different scaling factors
```

```
      is required for left and right coronary arteries (LCA and RCA).

Usage

    Specify this boundary condition in the boundary file (0/p) as follows:

    <patchName>
    {
    type                coronaryOutletPressure;
    Ra                   <value>;   //Arterial resistance in [kgm^-4s^-1], Default: 1
    Ram                  <value>;   //Micro-arterial resistance in [kgm^-4s^-1], Default: 1
    Rv                   <value>;   //Veinous resistance in [kgm^-4s^-1], Default: 1
    Rvm                  <value>;   //Micro-venous resistance in [kgm^-4s^-1], Default: 0
    Ca                   <value>;   //Arterial compliance in [m^4s^2kg^-1], Default: 1
    Cim                  <value>;   //Intramyocardial compliance in in [m^4s^2kg^-1], Default: 1
    PimFile              "<Path to Pim Data File>"; //Path to Pim time-series data file, Default:
     PimData
    PimScaling           <value>;   // Intramyocardial pressure scaling factor, Default: 1
    Pv                   <value>;   //Pressure distal to Rv in [Pa], Default: 0
    rho                  <value>;        // Fluid density in [kgm^-3], Default: 1
    diffScheme           <firstOrder or secondOrder>; // Differencing scheme, Default: secondOrder
    value                uniform 0;
    }

Author
    Muhammad Ahmad Raza, University College Dublin, Ireland.

Cite as
    Raza, M. A.: Implementation of Lumped Parameter Network Boundary Conditions for the
    Patient-Specific CFD Simulations of Coronary Arteries in OpenFOAM. In Proceedings of CFD with
    OpenSource Software, 2024, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2024

See also
    Foam::fixedValueFvPatchField
    Foam::fvPatchField

SourceFiles
    coronaryOutletPressureFvPatchScalarField.C

\*---------------------------------------------------------------------------*/

#ifndef coronaryOutletPressureFvPatchScalarField_H
#define coronaryOutletPressureFvPatchScalarField_H

#include "fixedValueFvPatchFields.H"
#include "scalar.H"
#include "word.H"
#include <vector>  // Required for std::vector
#include <utility> // Required for std::pair

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

// Enumerator for numerical differencing schemes used in the LPN model
enum DifferencingScheme
{
    firstOrder = 0, // First-order backward differencing scheme
    secondOrder = 1 // Second-order backward differencing scheme
};

/*---------------------------------------------------------------------------*\
           Class coronaryOutletPressureFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

/**
 * @brief Custom boundary condition for simulating hemodynamic outlet flow
```

```
 *        using a coronary lumped parameter network (LPN) model.
 *
 * This boundary condition models the outlet pressure based on a system of
 * ordinary differential equations (ODEs) capturing downstream coronary
 * vascular dynamics. It incorporates resistance, compliance, and
 * time-varying intramyocardial pressure to simulate physiological behavior
 * of coronary artery systems.
 *
 * The implementation supports first-order and second-order backward
 * differencing schemes, customizable parameters, and external input for
 * intramyocardial pressure data.
 */

class coronaryOutletPressureFvPatchScalarField
:
    public fixedValueFvPatchScalarField
{

private:

    // Private data

        //- Model parameters
        scalar Ra_;          // Arterial resistance
        scalar Ram_;         // Micro-arterial resistance
        scalar Rv_;          // Veinous resistance
        scalar Rvm_;         // Micro-veinous resistance
        scalar Ca_;          // Arterial compliance
        scalar Cim_;         // Intramyocardial compliance
        scalar PimScaling_;  // Scaling factor for intramyocardial pressure
        scalar Pv_;          // Pressure distal to Rv
        scalar rho_;         // Fluid density

        //- State variables for flow rates and pressures
        scalar Qoo_, Qo_, Qn_;       // Historical flow rate values
        scalar Poo_, Po_, Pn_;       // Historical pressure values
        scalar Pioo_, Pio_, Pin_;    // Historical intermediate pressure values
        scalar Pimoo_, Pimo_, Pimn_; // Historical intramyocardial pressure values

        //- LPN model configuration
        DifferencingScheme diffScheme_; // Numerical differencing scheme
        Foam::fileName PimFile_;      // File containing intramyocardial pressure data

        //- Intramyocardial pressure data
        std::vector<std::pair<scalar, scalar>> PimData_; // Time-series data for (time, Pim)

        //- Time index
        label timeIndex_;             // Current time index for updates

    // Helper methods

        //- Function to interpolate intramyocardial pressure (Pim) data at a given time
        scalar interpolatePim(const scalar& time) const;

        //- Function to load intramyocardial pressure (Pim) data from a specified file
        void loadPimData(const word& fileName);

        //- Function to expands environment variables in a given string
        static std::string expandEnvironmentVariables(const std::string& input);

public:

    //- Runtime type information
    TypeName("coronaryOutletPressure");

    // Constructors

        //- Construct from patch and internal field
```

```
        coronaryOutletPressureFvPatchScalarField
        (
            const fvPatch& patch,
            const DimensionedField<scalar, volMesh>& field
        );

        //- Construct from patch, internal field, and dictionary
        coronaryOutletPressureFvPatchScalarField
        (
            const fvPatch& patch,
            const DimensionedField<scalar, volMesh>& field,
            const dictionary& dict
        );

        //- Map existing object onto a new patch
        coronaryOutletPressureFvPatchScalarField
        (
            const coronaryOutletPressureFvPatchScalarField& other,
            const fvPatch& patch,
            const DimensionedField<scalar, volMesh>& field,
            const fvPatchFieldMapper& mapper
        );

        //- Copy constructor
        coronaryOutletPressureFvPatchScalarField
        (
            const coronaryOutletPressureFvPatchScalarField& other
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
            (
                new coronaryOutletPressureFvPatchScalarField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        coronaryOutletPressureFvPatchScalarField
        (
            const coronaryOutletPressureFvPatchScalarField& other,
            const DimensionedField<scalar, volMesh>& field
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new coronaryOutletPressureFvPatchScalarField(*this, iF)
            );
        }

    // Member functions

        //- Evaluation function to update coefficients for the boundary
        //  condition based on flow rate and model parameters
        virtual void updateCoeffs();

        //- Write boundary condition data to output stream
        virtual void write(Ostream& os) const;
};

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#endif

// ************************************************************************* //
```

A.2.2 Source file

The source file `coronaryOutletPressureFvPatchScalarField.C` contains the main implementation of the coronary lumped parameter network (LPN) model boundary condition. The code contained in this source file is given below.

coronaryOutletPressureFvPatchScalarField.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | foam-extend: Open Source CFD
   \\    /   O peration     | Version:     4.1
    \\  /    A nd           | Web:         http://www.foam-extend.org
     \\/     M anipulation  | For copyright notice see file Copyright
-------------------------------------------------------------------------------
License
    This file is part of foam-extend.

    foam-extend is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation, either version 3 of the License, or (at your
    option) any later version.

    foam-extend is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with foam-extend.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/


// Include the header file defining the class and related dependencies
#include "coronaryOutletPressureFvPatchScalarField.H"
#include "addToRunTimeSelectionTable.H" // Macro for runtime selection table
#include "fvPatchFieldMapper.H"         // Field mapper utility
#include "volFields.H"                  // Volume fields
#include "surfaceFields.H"              // Surface fields
#include "scalar.H"                     // Scalar type definition
#include <fstream>                      // File handling for reading/writing
#include <string.h>                     // String manipulation utilities
#include "IOdictionary.H"               // Dictionary I/O operations
#include "fileName.H"                   // FileName handling
#include "OSspecific.H"                 // OS-specific operations (e.g., paths)
#include <cstdlib>                      // Environment variable handling


// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * * //

// (None in this section)

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * //

//- Default constructor: Initializes the class with default values for all parameters.
Foam::coronaryOutletPressureFvPatchScalarField::
```

```
coronaryOutletPressureFvPatchScalarField
(
    const fvPatch& p,                                   // Patch reference
    const DimensionedField<Foam::scalar, volMesh>& iF // Internal field reference
)
:
    fixedValueFvPatchScalarField(p, iF), // Call the base class constructor
    Ra_(1),                              // Initialize default arterial resistance
    Ram_(1),                             // Initialize micro-arterial resistance
    Rv_(1),                              // Initialize venous resistance
    Rvm_(1),                             // Initialize micro-venous resistance
    Ca_(1),                              // Initialize arterial compliance
    Cim_(1),                             // Initialize intramyocardial compliance
    PimScaling_(1),                      // Initialize scaling factor for intramyocardial pressure
    Pv_(0),                              // Default distal pressure
    rho_(1),                             // Default fluid density
    Qoo_(0), Qo_(0), Qn_(0),             // Initialize flow rate history
    Poo_(0), Po_(0), Pn_(0),             // Initialize pressure history
    Pioo_(0), Pio_(0), Pin_(0),          // Initialize intermediate pressure history
    Pimoo_(0), Pimo_(0), Pimn_(0),       // Initialize intramyocardial pressure history
    diffScheme_(secondOrder),            // Use second-order differencing by default
    PimFile_("PimData"),                   //Default name of the Pim data file
    timeIndex_(-1)                       // Set an invalid time index as the initial state
{}

//- Constructor from a dictionary, typically used for user input configuration.
Foam::coronaryOutletPressureFvPatchScalarField::
coronaryOutletPressureFvPatchScalarField
(
    const fvPatch& p,                                    // Patch reference
    const DimensionedField<Foam::scalar, volMesh>& iF, // Internal field reference
    const dictionary& dict                              // Dictionary containing user-provided values
)
:
    fixedValueFvPatchScalarField(p, iF, dict), // Call the base class constructor
    Ra_(dict.lookupOrDefault<Foam::scalar>("Ra", 1)),       // Arterial resistance
    Ram_(dict.lookupOrDefault<Foam::scalar>("Ram", 1)),     // Micro-arterial resistance
    Rv_(dict.lookupOrDefault<Foam::scalar>("Rv", 1)),       // Venous resistance
    Rvm_(dict.lookupOrDefault<Foam::scalar>("Rvm", 1)),     // Micro-venous resistance
    Ca_(dict.lookupOrDefault<Foam::scalar>("Ca", 1)),       // Arterial compliance
    Cim_(dict.lookupOrDefault<Foam::scalar>("Cim", 1)),     // Intramyocardial compliance
    PimScaling_(dict.lookupOrDefault<Foam::scalar>("PimScaling", 1)), // Pim scaling factor
    Pv_(dict.lookupOrDefault<Foam::scalar>("Pv", 0)),       // Distal pressure
    rho_(dict.lookupOrDefault<Foam::scalar>("rho", 1)),     // Fluid density
    Qoo_(dict.lookupOrDefault<Foam::scalar>("Qoo", 0)),     // Historical flow rate values
    Qo_(dict.lookupOrDefault<Foam::scalar>("Qo", 0)),
    Qn_(dict.lookupOrDefault<Foam::scalar>("Qn", 0)),
    Poo_(dict.lookupOrDefault<Foam::scalar>("Poo", 0)),     // Historical pressure values
    Po_(dict.lookupOrDefault<Foam::scalar>("Po", 0)),
    Pn_(dict.lookupOrDefault<Foam::scalar>("Pn", 0)),
    Pioo_(dict.lookupOrDefault<Foam::scalar>("Pioo", 0)),   // Historical intermediate pressure values
    Pio_(dict.lookupOrDefault<Foam::scalar>("Pio", 0)),
    Pin_(dict.lookupOrDefault<Foam::scalar>("Pin", 0)),
    Pimoo_(dict.lookupOrDefault<Foam::scalar>("Pimoo", 0)), // Historical intramyocardial pressure
    Pimo_(dict.lookupOrDefault<Foam::scalar>("Pimo", 0)),
    Pimn_(dict.lookupOrDefault<Foam::scalar>("Pimn", 0)),
    timeIndex_(-1) // Initialize time index
{
    Info << "\n\nApplying coronaryOutletPressure BC on patch: " << patch().name() << endl;

    //- Extract differencing scheme from dictionary and validate
    word schemeStr = dict.lookupOrDefault<word>("diffScheme", "secondOrder");

    Info << "Differencing Scheme for Coronary LPN Model: " << schemeStr << endl;

    if (schemeStr == "firstOrder")
    {
        diffScheme_ = firstOrder; // First-order scheme
```

```
    }
    else if (schemeStr == "secondOrder")
    {
        diffScheme_ = secondOrder; // Second-order scheme
    }
    else
    {
        FatalErrorInFunction << "\n\nUnknown Differencing Scheme: " << schemeStr
                             << "\nValid Differencing Schemes (diffScheme) are : \n\n"
                             << " 2 \n ( \n firstOrder \n secondOrder \n ) \n"
                             << exit(FatalError);
    }

    //- Extract file name for intramyocardial pressure (Pim) data
    word PimFileStr = dict.lookupOrDefault<fileName>("PimFile", "PimData");

    PimFile_ = PimFileStr;

    //- Load the Pim data from the specified file
    loadPimData(PimFile_);

    //- Print coronary LPN model properties for debugging/verification
    Info << "\n\nProperties of Coronary LPN Model: \n"
            << "Differencig Scheme: " << diffScheme_ << " \n"
            << "Arterial Resistance: Ra = " << Ra_ << " kgm^-4s^-1 \n"
            << "Micro-arterial Resistance: Ram = " << Ram_ << " kgm^-4s^-1 \n"
            << "Veinous Resistance: Rv = " << Rv_ << " kgm^-4s^-1 \n"
            << "Micro-veinous Resistance: Rvm = " << Rvm_ << " kgm^-4s^-1 \n"
            << "Arterial Compliance: Ca = " << Ca_ << " m^4s^2kg^-1 \n"
            << "Intramyocardial Compliance: Cim = " << Cim_ << " m^4s^2kg^-1 \n"
            << "Path to Pim Data File: " << PimFile_ << " \n"
            << "Pim Scaling Factor: PimScaling: " << PimScaling_ << " \n"
            << "Pressure Distal to Rv: Pv = " << Pv_ << " Pa \n"
            << "Density: rho = " << rho_ << " kgm^-3 \n" << endl;
}

//- Constructor: Map an existing field onto a new patch
Foam::coronaryOutletPressureFvPatchScalarField::
coronaryOutletPressureFvPatchScalarField
(
    const coronaryOutletPressureFvPatchScalarField& ptf, // Existing field to map from
    const fvPatch& p,                                    // New patch
    const DimensionedField<Foam::scalar, volMesh>& iF,  // Internal field
    const fvPatchFieldMapper& mapper                     // Mapper for field data
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper),    // Call base class mapping constructor
    Ra_(ptf.Ra_),                                        // Copy resistance parameter Ra
    Ram_(ptf.Ram_),                                      // Copy modified resistance Ram
    Rv_(ptf.Rv_),                                        // Copy venous resistance Rv
    Rvm_(ptf.Rvm_),                                      // Copy modified venous resistance Rvm
    Ca_(ptf.Ca_),                                        // Copy arterial compliance Ca
    Cim_(ptf.Cim_),                                      // Copy impedance compliance Cim
    PimScaling_(ptf.PimScaling_),                        // Copy scaling for Pim values
    Pv_(ptf.Pv_),                                        // Copy distal pressure Pv
    rho_(ptf.rho_),                                      // Copy density rho
    Qoo_(ptf.Qoo_), Qo_(ptf.Qo_), Qn_(ptf.Qn_),          // Copy flow rates Qoo, Qo, Qn
    Poo_(ptf.Poo_), Po_(ptf.Po_), Pn_(ptf.Pn_),          // Copy pressures Poo, Po, Pn
    Pioo_(ptf.Pioo_), Pio_(ptf.Pio_), Pin_(ptf.Pin_),    // Copy intermediate pressures Pioo, Pio, Pin
    Pimoo_(ptf.Pimoo_), Pimo_(ptf.Pimo_), Pimn_(ptf.Pimn_), // Copy Pim intermediate pressures
    diffScheme_(ptf.diffScheme_),                        // Copy differencing scheme
    PimFile_(ptf.PimFile_),                              // Copy Pim data file path
    PimData_(ptf.PimData_),                              // Copy Pim data storage
    timeIndex_(ptf.timeIndex_)                          // Copy current time index
{}

//- Copy constructor: Create a copy of an existing field
Foam::coronaryOutletPressureFvPatchScalarField::
```

```
coronaryOutletPressureFvPatchScalarField
(
    const coronaryOutletPressureFvPatchScalarField& copsf // Field to copy
)
:
    fixedValueFvPatchScalarField(copsf),              // Call base class copy constructor
    Ra_(copsf.Ra_),                                   // Copy resistance parameter Ra
    Ram_(copsf.Ram_),                                 // Copy modified resistance Ram
    Rv_(copsf.Rv_),                                   // Copy venous resistance Rv
    Rvm_(copsf.Rvm_),                                 // Copy modified venous resistance Rvm
    Ca_(copsf.Ca_),                                   // Copy arterial compliance Ca
    Cim_(copsf.Cim_),                                 // Copy impedance compliance Cim
    PimScaling_(copsf.PimScaling_),                   // Copy scaling for Pim values
    Pv_(copsf.Pv_),                                   // Copy distal pressure Pv
    rho_(copsf.rho_),                                 // Copy density rho
    Qoo_(copsf.Qoo_), Qo_(copsf.Qo_), Qn_(copsf.Qn_), // Copy flow rates Qoo, Qo, Qn
    Poo_(copsf.Poo_), Po_(copsf.Po_), Pn_(copsf.Pn_), // Copy pressures Poo, Po, Pn
    Pioo_(copsf.Pioo_), Pio_(copsf.Pio_), Pin_(copsf.Pin_), // Copy intermediate pressures
    Pimoo_(copsf.Pimoo_), Pimo_(copsf.Pimo_), Pimn_(copsf.Pimn_), // Copy Pim intermediate pressures
    diffScheme_(copsf.diffScheme_),                    // Copy differencing scheme
    PimFile_(copsf.PimFile_),                          // Copy Pim data file path
    PimData_(copsf.PimData_),                          // Copy Pim data storage
    timeIndex_(copsf.timeIndex_)                       // Copy current time index
{}

//- Copy constructor with new internal field reference
Foam::coronaryOutletPressureFvPatchScalarField::
coronaryOutletPressureFvPatchScalarField
(
    const coronaryOutletPressureFvPatchScalarField& copsf, // Field to copy
    const DimensionedField<scalar, volMesh>& iF            // New internal field reference
)
:
    fixedValueFvPatchScalarField(copsf, iF),          // Call base class copy with new field
    Ra_(copsf.Ra_),                                   // Copy resistance parameter Ra
    Ram_(copsf.Ram_),                                 // Copy modified resistance Ram
    Rv_(copsf.Rv_),                                   // Copy venous resistance Rv
    Rvm_(copsf.Rvm_),                                 // Copy modified venous resistance Rvm
    Ca_(copsf.Ca_),                                   // Copy arterial compliance Ca
    Cim_(copsf.Cim_),                                 // Copy impedance compliance Cim
    PimScaling_(copsf.PimScaling_),                   // Copy scaling for Pim values
    Pv_(copsf.Pv_),                                   // Copy distal pressure Pv
    rho_(copsf.rho_),                                 // Copy density rho
    Qoo_(copsf.Qoo_), Qo_(copsf.Qo_), Qn_(copsf.Qn_), // Copy flow rates Qoo, Qo, Qn
    Poo_(copsf.Poo_), Po_(copsf.Po_), Pn_(copsf.Pn_), // Copy pressures Poo, Po, Pn
    Pioo_(copsf.Pioo_), Pio_(copsf.Pio_), Pin_(copsf.Pin_), // Copy intermediate pressures
    Pimoo_(copsf.Pimoo_), Pimo_(copsf.Pimo_), Pimn_(copsf.Pimn_), // Copy Pim intermediate pressures
    diffScheme_(copsf.diffScheme_),                   // Copy differencing scheme
    PimFile_(copsf.PimFile_),                         // Copy Pim data file path
    PimData_(copsf.PimData_),                         // Copy Pim data storage
    timeIndex_(copsf.timeIndex_)                     // Copy current time index
{}

// * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //

//- Update coefficients for boundary condition
void Foam::coronaryOutletPressureFvPatchScalarField::updateCoeffs()
{
    //- Check if coefficients are already updated
    if (updated())
    {
        return;
    }

    //- Get time step size and current simulation time
    const scalar dt = db().time().deltaTValue(); // Time step size
    const scalar currentTime = db().time().value(); // Current time
```

```
    //- Retrieve the flux field (phi) from the database
    const surfaceScalarField& phi =
        db().lookupObject<surfaceScalarField>("phi");

    //- Retrieve the boundary pressure field from the database
    const fvPatchField<scalar>& p =
        db().lookupObject<volScalarField>("p").boundaryField()[this->patch().index()];

    //- Calculate total patch area
    scalar area = gSum(patch().magSf());

    //- Update pressure and flow rate histories if time index has advanced
    if (db().time().timeIndex() != timeIndex_)
    {
        timeIndex_ = db().time().timeIndex(); // Update time index

        //- Update flow rate history
        Qoo_ = Qo_;
        Qo_ = Qn_;
        Qn_ = gSum(phi.boundaryField()[patch().index()]); // Compute current flow rate

        //- Update pressure history
        Poo_ = Po_;
        Po_ = Pn_;
        Pn_ = gSum(p * patch().magSf()) / area; // Compute current pressure (area-weighted average)

        //- Update Pim history
        Pimoo_ = Pimo_;
        Pimo_ = Pimn_;
        Pimn_ = interpolatePim(currentTime); // Interpolate to compute current Pim

        //- Update intermediate pressure history
        Pioo_ = Pio_;
        Pio_ = Pin_;
    }

    //- Perform calculations only if time index exceeds 1
    if (db().time().timeIndex() > 1)
    {
        //- Choose differencing scheme
        switch (diffScheme_)
        {
            case firstOrder: // First-order scheme

                //- Compute intermediate pressure at current time
                Pin_ = (Rvm_ + Rv_) * Qn_
                    - (Rvm_ + Rv_) * Ca_ * ((Pn_ - Po_) / dt)
                    + (Rvm_ + Rv_) * Cim_ * ((Pimn_ - Pimo_) / dt)
                    + Pv_
                    + (Rvm_ + Rv_) * Cim_ * Pio_ / dt;

                Pin_ /= (1.0 + (Rvm_ + Rv_) * Cim_ / dt) + SMALL;

                //- Compute boundary pressure at current time
                Pn_ = (Ram_ + Ra_) * Qn_
                    + Ram_ * Ra_ * Ca_ * ((Qn_ - Qo_) / dt)
                    + Pin_
                    + Ram_ * Ca_ * (Po_ / dt);

                Pn_ /= (1.0 + Ram_ * Ca_ / dt) + SMALL;

                break;

            case secondOrder: // Second-order scheme

                //- Compute intermediate pressure at current time
                Pin_ = (Rvm_ + Rv_) * Qn_
                    - (Rvm_ + Rv_) * Ca_ * ((3 * Pn_ - 4 * Po_ + Poo_) / (2 * dt))
```

```
                    + (Rvm_ + Rv_) * Cim_ * ((3 * Pimn_ - 4 * Pimo_ + Pimoo_) / (2 * dt))
                    + Pv_
                    - (Rvm_ + Rv_) * Cim_ * (Pioo_ - 4 * Pio_) / (2 * dt);

                Pin_ /= (1.0 + 3 * (Rvm_ + Rv_) * Cim_ /(2 * dt)) + SMALL;

                //- Compute boundary pressure at current time
                Pn_ = (Ram_ + Ra_) * Qn_
                    + Ram_ * Ra_ * Ca_ * ((3 * Qn_ - 4 * Qo_ + Qoo_) / (2 * dt))
                    + Pin_
                    - Ram_ * Ca_ * (Poo_ - 4 * Po_) / (2 * dt);

                Pn_ /= (1.0 + 3 * Ram_ * Ca_ / (2 * dt)) + SMALL;

                break;

            default:
                //- Handle unknown schemes
                FatalErrorInFunction << "Unknown differencing scheme!" << exit(FatalError);
        }
    }

    //- Apply implicit update to the boundary field
    operator==(Pn_ / (rho_ + SMALL));

    //- Call base class function to finalize the update
    fixedValueFvPatchScalarField::updateCoeffs();
}

// Helper methods

//- Interpolates the intramyocardial pressure (Pim) at a given time
Foam::scalar Foam::coronaryOutletPressureFvPatchScalarField::interpolatePim(const scalar& time) const
{
    //- Ensure PimData_ is populated
    if (PimData_.empty())
    {
        FatalErrorInFunction << "PimData_ is empty. Cannot interpolate!" << exit(FatalError);
    }

    //- Get the start and end times of the Pim data
    const scalar tStart = PimData_.front().first; // Start time of the dataset
    const scalar tEnd = PimData_.back().first;   // End time of the dataset

    //- Calculate the effective time using modulo to handle periodicity
    scalar effectiveTime = tStart + std::fmod(time - tStart, tEnd - tStart);
    if (effectiveTime < tStart)
    {
        effectiveTime += (tEnd - tStart); // Adjust for negative modulo results
    }

    //- Perform linear interpolation to find the corresponding Pim value
    for (size_t i = 1; i < PimData_.size(); ++i)
    {
        if (PimData_[i - 1].first <= effectiveTime && PimData_[i].first >= effectiveTime)
        {
            scalar t1 = PimData_[i - 1].first;      // Previous time point
            scalar t2 = PimData_[i].first;          // Current time point
            scalar Pim1 = PimData_[i - 1].second;   // Pim value at previous time
            scalar Pim2 = PimData_[i].second;       // Pim value at current time

            //- Linear interpolation formula
            return PimScaling_ * (Pim1 + (Pim2 - Pim1) * (effectiveTime - t1) / (t2 - t1));
        }
    }

    //- Fallback case: return the last Pim value (should not typically be reached)
    return PimData_.back().second;
```

```
}

//- Loads intramyocardial pressure data (PimData_) from a file
void Foam::coronaryOutletPressureFvPatchScalarField::loadPimData(const word& fileName)
{
    //- Clear existing data
    PimData_.clear();

    //- Expand environment variables in the file name
    std::string expandedFileName = expandEnvironmentVariables(fileName);

    //- Open the file
    std::ifstream file(expandedFileName.c_str());

    //- Check if the file was successfully opened
    if (!file.is_open())
    {
        FatalErrorInFunction << "Cannot open intramyocardial pressure file: " << fileName << exit(
     FatalError);
    }

    std::string line;
    while (std::getline(file, line)) // Read the file line by line
    {
        //- Remove parentheses from the line
        line.erase(std::remove(line.begin(), line.end(), '('), line.end());
        line.erase(std::remove(line.begin(), line.end(), ')'), line.end());

        //- Skip empty or invalid lines
        if (line.empty()) continue;

        //- Parse the line into time and Pim values
        std::istringstream iss(line);
        scalar time, Pim;
        if (iss >> time >> Pim) // If parsing is successful
        {
            PimData_.emplace_back(time, Pim); // Store the time and Pim values
        }
    }

    file.close(); // Close the file
}

//- Expands environment variables in the given string
std::string Foam::coronaryOutletPressureFvPatchScalarField::expandEnvironmentVariables(const std::
     string& input)
{
    std::string result = input; // Start with the input string
    size_t pos = 0;

    //- Look for occurrences of "$" followed by an alphabetic character (environment variable pattern)
    while ((pos = result.find("$", pos)) != std::string::npos)
    {
        size_t endPos = result.find_first_not_of("ABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789", pos + 1);
        if (endPos == std::string::npos)
        {
            //- No valid environment variable found after "$"
            break;
        }

        //- Extract the environment variable name
        std::string varName = result.substr(pos + 1, endPos - pos - 1);

        //- Get the environment variable value from the system
        const char* envValue = std::getenv(varName.c_str());

        if (envValue) // If the environment variable exists
        {
```

```
            //- Replace the variable with its value
            result.replace(pos, endPos - pos, envValue);
            pos += std::string(envValue).size(); // Move past the replaced value
        }
        else
        {
            //- If the environment variable is not found, skip to the next "$"
            pos = endPos;
        }
    }
    return result; // Return the string with expanded variables
}

//- Write the boundary condition properties to an output stream
void Foam::coronaryOutletPressureFvPatchScalarField::write(Ostream& os) const
{
    //- Call base class method to write common properties
    fvPatchScalarField::write(os);

    //- Write each parameter with its keyword and value
    os.writeKeyword("Ra") << Ra_ << token::END_STATEMENT << nl;
    os.writeKeyword("Ram") << Ram_ << token::END_STATEMENT << nl;
    os.writeKeyword("Rv") << Rv_ << token::END_STATEMENT << nl;
    os.writeKeyword("Rvm") << Rvm_ << token::END_STATEMENT << nl;
    os.writeKeyword("Ca") << Ca_ << token::END_STATEMENT << nl;
    os.writeKeyword("Cim") << Cim_ << token::END_STATEMENT << nl;
    os.writeKeyword("PimFile") << PimFile_ << token::END_STATEMENT << nl;
    os.writeKeyword("PimScaling") << PimScaling_ << token::END_STATEMENT << nl;
    os.writeKeyword("Pv") << Pv_ << token::END_STATEMENT << nl;
    os.writeKeyword("rho") << rho_ << token::END_STATEMENT << nl;

    //- Write the differencing scheme
    switch (diffScheme_)
    {
        case firstOrder:
            os.writeKeyword("diffScheme") << "firstOrder" << token::END_STATEMENT << nl;
            break;
        case secondOrder:
            os.writeKeyword("diffScheme") << "secondOrder" << token::END_STATEMENT << nl;
            break;
        default:
            FatalErrorInFunction << "Unknown differencing scheme: " << diffScheme_ << exit(FatalError)
     ;
    }

    //- Write flow history parameters
    os.writeKeyword("Qoo") << Qoo_ << token::END_STATEMENT << nl;
    os.writeKeyword("Qo") << Qo_ << token::END_STATEMENT << nl;
    os.writeKeyword("Qn") << Qn_ << token::END_STATEMENT << nl;

    //- Write pressure history parameters
    os.writeKeyword("Pimoo") << Pimoo_ << token::END_STATEMENT << nl;
    os.writeKeyword("Pimo") << Pimo_ << token::END_STATEMENT << nl;
    os.writeKeyword("Pimn") << Pimn_ << token::END_STATEMENT << nl;
    os.writeKeyword("Pioo") << Pioo_ << token::END_STATEMENT << nl;
    os.writeKeyword("Pio") << Pio_ << token::END_STATEMENT << nl;
    os.writeKeyword("Pin") << Pin_ << token::END_STATEMENT << nl;
    os.writeKeyword("Poo") << Poo_ << token::END_STATEMENT << nl;
    os.writeKeyword("Po") << Po_ << token::END_STATEMENT << nl;
    os.writeKeyword("Pn") << Pn_ << token::END_STATEMENT << nl;

    //- Write the boundary field value entry to the output stream
    writeEntry("value", os);
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

//- Registration
```

```
namespace Foam
{
    //- Define the boundary field type for this class
    makePatchTypeField
    (
        fvPatchScalarField,                       // Base class
        coronaryOutletPressureFvPatchScalarField  // Derived class
    );
}

// ************************************************************************* //
```

A.3 Make directory

A.3.1 `Make/options` file

Various includes added in `Make/options` file are shown below.

`Make/options`

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools
```

A.3.2 `Make/files` file

The paths to the source files for both boundary conditions are included in the `Make/files` file, and both of them are compiled into a single library `liblumpedParameterNetworkBC` as shown below.

`Make/files`

```
windkesselOutletPressure/windkesselOutletPressureFvPatchScalarField.C
coronaryOutletPressure/coronaryOutletPressureFvPatchScalarField.C
LIB = $(FOAM_USER_LIBBIN)/liblumpedParameterNetworkBC
```

A.4 Directory structure and compilation

The original directory structure of the `lumpedParameterNetworkBC` before compilation is shown below in Figure A.1.

```
lumpedParameterNetworkBC/
├── {Allwclean, Allwmake}
├── Make/
│   └── {files, options}
├── coronaryOutletPressure/
│   └── coronaryOutletPressureFvPatchScalarField.{C, H}
└── windkesselOutletPressure/
    └── windkesselOutletPressureFvPatchScalarField.{C, H}
```

Figure A.1: Original directory structure of the `lumpedParameterNetworkBC`

The library can be compiled by running the `Allwmake` bash script shown below.

`Allwmake`

```
#!/bin/bash

echo "Compiling liblumpedParameterNetworkBC..."

wmake libso

echo "liblumpedParameterNetworkBC is compiled and ready for the use!"
```

The directory structure of `lumpedParameterNetworkBC` after the successful compilation should look something like the tree given in the Figure A.2. Moreover, it will create a library named `liblumpedParameterNetworkBC.so` within the user library directory i.e., `$FOAM_USER_LIBBIN`. Hence, to use the boundary the developed boundary condition, a library `liblumpedParameterNetworkBC.so` should be included in the `controlDict` dictionary of the OpenFOAM case.

The library can be cleaned by running the `Allwclean` bash script shown below.

`Allwclean`

```
#!/bin/bash

echo "Cleaning liblumpedParameterNetworkBC..."

wclean libso

echo "liblumpedParameterNetworkBC is cleaned and ready for Recompilation!"
```

```
lumpedParameterNetworkBC/
├── {Allwclean, Allwmake}
├── Make/
│   ├── {files, options}
│   └── linux64GccDPInt32Opt/
│       └── {coronaryOutletPressureFvPatchScalarField.o, dependencies,
│           dependencyFiles, dontIncludeDeps, files, filesMacros, includeDeps,
│           localObjectFiles, objectFiles, options, sourceFiles,
│           windkesselOutletPressureFvPatchScalarField.o}
├── coronaryOutletPressure/
│   └── coronaryOutletPressureFvPatchScalarField.{C, H, dep}
├── lnInclude/
│   ├── coronaryOutletPressureFvPatchScalarField.C ->
│   │   ../coronaryOutletPressure/coronaryOutletPressureFvPatchScalarField.C
│   ├── coronaryOutletPressureFvPatchScalarField.H ->
│   │   ../coronaryOutletPressure/coronaryOutletPressureFvPatchScalarField.H
│   ├── uptodate
│   ├── windkesselOutletPressureFvPatchScalarField.C ->
│   │   ../windkesselOutletPressure/windkesselOutletPressureFvPatchScalarField.C
│   └── windkesselOutletPressureFvPatchScalarField.H ->
│       ../windkesselOutletPressure/windkesselOutletPressureFvPatchScalarField.H
└── windkesselOutletPressure/
    └── windkesselOutletPressureFvPatchScalarField.{C, H, dep}
```

Figure A.2: Post-compilation directory structure of the `lumpedParameterNetworkBC`

Appendix B

Test Cases Codes

B.1 Original directory structure

The original directory structure of `LumpedParameterNetworkBCTestCase` master folder, containing all the BC files, cases and scripts is shown in the Figure B.1.

```
LumpedParameterNetworkBCTestCase/
├── Allclean, Allrun, CreateDirStruc, DeleteDirStruc
├── LPNBCTestCase/
│   ├── 0.orig/
│   │   ├── U.{Coronary, WK}
│   │   ├── p.{Resistive}
│   │   └── p.{WK2, WK3, WK4Parallel, WK4Series, Coronary}.{firstOrder,
│   │       secondOrder}
│   ├── Allclean.{Coronary, WK}
│   ├── Allrun.{Coronary, WK}
│   ├── DataFiles/
│   │   └── AorticInletFlowRate, CoronaryInletFlowRate, PimData
│   ├── constant/
│   │   ├── RASProperties, dynamicMeshDict, fluidProperties,
│   │   │   physicsProperties, transportProperties, turbulenceProperties
│   │   └── polyMesh/
│   │       └── blockMeshDict.m4.{Coronary, WK}
│   └── system/
│       └── controlDict, decomposeParDict, fvSchemes, fvSolution
├── MATLABPostprocessing/
│   └── LPNBCTestCasePostprocessing.m
└── lumpedParameterNetworkBC/
    ├── Allwclean, Allwmake
    ├── Make/
    │   └── files, options
    ├── coronaryOutletPressure/
    │   └── coronaryOutletPressureFvPatchScalarField.{C, H}
    └── windkesselOutletPressure/
        └── windkesselOutletPressureFvPatchScalarField.{C, H}
```

Figure B.1: Original directory structure of `LumpedParameterNetworkBCTestCase`

B.2 Boundary condition source code

The `lumpedParameterNetworkBC` directory in the master folder contains the files and scripts necessary to compile LPN boundary conditions developed in this project. The code listings of this directory are given in the previous appendix.

B.3 Test cases master directory

The `LPNBCTestCase` directory in the master folder, is a master directory to run all the test cases. It includes the OpenFOAM dictionaries and files necessary to run all the test cases.

B.3.1 `0.orig` directory

The `0.orig` directory inside `LPNBCTestCase` master directory contains the `U` and `p` boundary condition dictionaries for all cases.

B.3.1.1 `U` BC dictionaries

There are two `U` BC dictionaries present in the `0.orig` directory: `U.WK` and `U.Coronary`.

i. `U.WK` BC dictionary: The `U.WK` is for all the simulations involving the boundary condition relating to `windkesselOutletPressure` family i.e., `"Resistive", "WK2", "WK3", "WK4Series", "WK4Parallel"`.

`U.WK` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    inlet
    {
        type                timeVaryingFlowRateInletVelocity;
        flowRate            0;              // Volumetric/mass flow rate [m3/s or kg/s]
        value               uniform (0 0 0); // placeholder
        "file|fileName"     "$FOAM_CASE/DataFiles/AorticInletFlowRate";
        outOfBounds         repeat;         // (error|warn|clamp|repeat)
    }

    outlet
    {
        type            zeroGradient;
    }

    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

}

// ************************************************************************* //
```

ii. `U.Coronary` BC dictionary: The `U.Coronary` is for all the simulations involving coronary LPN boundary condition i.e., `coronaryOutletPressure`.

`U.WK` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    inlet
    {
        type                timeVaryingFlowRateInletVelocity;
        flowRate            0;              // Volumetric/mass flow rate [m3/s or kg/s]
        value               uniform (0 0 0); // placeholder
        "file|fileName"     "$FOAM_CASE/DataFiles/CoronaryInletFlowRate";
        outOfBounds         repeat;         // (error|warn|clamp|repeat)
    }

    outlet
    {
        type            zeroGradient;
    }

    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

}

// ************************************************************************* //
```

B.3.1.2 `p` BC dictionaries

Since the developed boundary conditions are for outlet pressure, hence each simulation case has its own `p` dictionary.

i. `p.Resistive` BC dictionary: It implements the `"Resistive"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` on a straight aorta geometry's outlet.

`p.Resistive` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
```

```
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     Resistive;
        Rd                  17690000;   //Distal resistance in [kgm^-4s^-1]
        Pd                  9.7363e+03; // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

ii. `p.WK2.firstOrder` BC dictionary: It implements the `"WK2"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` with `firstOrder` differencing scheme on a straight aorta geometry's outlet.

`p.WK2.firstOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;
```

```
boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     WK2;
        diffScheme          firstOrder;
        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
        C                   8.3333e-09; // Compliance in [m^4s^2kg^-1]
        Pd                  0;          // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

iii. `p.WK2.secondOrder` BC dictionary: It implements the `"WK2"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` with `secondOrder` differencing scheme on a straight aorta geometry's outlet.

`p.WK2.secondOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     WK2;
        diffScheme          secondOrder;
        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
```

```
        C                   8.3333e-09; // Compliance in [m^4s^2kg^-1]
        Pd                  0;          // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

iv. `p.WK3.firstOrder` BC dictionary: It implements the `"WK3"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` with `firstOrder` differencing scheme on a straight aorta geometry's outlet.

`p.WK3.firstOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     WK3;
        diffScheme          firstOrder;
        Rp                  13997000;   // Proximal resistance in [kgm^-4s^-1]
        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
        C                   1.0000e-08; // Compliance in [m^4s^2kg^-1]
        Pd                  0;          // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}
```

```
// ************************************************************************* //
```

v. `p.WK3.secondOrder` BC dictionary: It implements the `"WK3"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` with `secondOrder` differencing scheme on a straight aorta geometry's outlet.

`p.WK3.secondOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     WK3;
        diffScheme          secondOrder;
        Rp                  13997000;   // Proximal resistance in [kgm^-4s^-1]
        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
        C                   1.0000e-08; // Compliance in [m^4s^2kg^-1]
        Pd                  0;          // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

vi. `p.WK4Series.firstOrder` BC dictionary: It implements the `"WK4Series"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` with `firstOrder` differencing scheme on a straight aorta geometry's outlet.

`p.WK4Series.firstOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
```

```
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     WK4Series;
        diffScheme          firstOrder;
        Rp                  13997000;   // Proximal resistance in [kgm^-4s^-1]
        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
        C                   2.5000e-08; // Compliance in [m^4s^2kg^-1]
        L                   1.7995e+04; // Blood inertance in [kgm^-4]
        Pd                  0;          // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

vii. `p.WK4Series.secondOrder` BC dictionary: It implements the `"WK4Series"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` with `secondOrder` differencing scheme on a straight aorta geometry's outlet.

`p.WK4Series.secondOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     WK4Series;
        diffScheme          secondOrder;
        Rp                  13997000;   // Proximal resistance in [kgm^-4s^-1]
        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
        C                   2.5000e-08; // Compliance in [m^4s^2kg^-1]
        L                   1.7995e+04; // Blood inertance in [kgm^-4]
        Pd                  0;          // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

viii. `p.WK4Parallel.firstOrder` BC dictionary: It implements the `"WK4Parallel"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` with `firstOrder` differencing scheme on a straight aorta geometry's outlet.

`p.WK4Parallel.firstOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
```

```
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     WK4Parallel;
        diffScheme          firstOrder;
        Rp                  1.3997e+10; // Proximal resistance in [kgm^-4s^-1]
        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
        C                   1.0000e-08; // Compliance in [m^4s^2kg^-1]
        L                   1.7995e+03; // Blood inertance in [kgm^-4]
        Pd                  0;          // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

ix. `p.WK4Parallel.secondOrder` BC dictionary: It implements the `"WK4Parallel"` boundary condition from windkessel BC family i.e., `windkesselOutletPressure` with `secondOrder` differencing scheme on a straight aorta geometry's outlet.

`p.WK4Parallel.secondOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type                windkesselOutletPressure;
        windkesselModel     WK4Parallel;
        diffScheme          secondOrder;
        Rp                  1.3997e+10; // Proximal resistance in [kgm^-4s^-1]
        Rd                  141520000;  // Distal resistance in [kgm^-4s^-1]
        C                   1.0000e-08; // Compliance in [m^4s^2kg^-1]
        L                   1.7995e+03; // Blood inertance in [kgm^-4]
```

```
        Pd                  0;          // Distal pressure in [Pa]
        value               uniform 0;
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

x. `p.Coronary.firstOrder` BC dictionary: It implements the `coronaryOutletPressure` boundary condition with `firstOrder` differencing scheme on a straight LCA geometry's outlet.

`p.Coronary.firstOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type            coronaryOutletPressure;
        Ra              1.0179e+10;    // Arterial resistance [kg m^-4 s^-1]
        Ram             1.6541e+10;    // Micro-arterial resistance [kg m^-4 s^-1]
        Rv              5.0896e+09;    // Veinous resistance [kg m^-4 s^-1]
        Rvm             0;             // Micro-veinous resistance [kg m^-4 s^-1]
        Ca              8.6482e-12;    // Arterial compliance [m^4 s^2 kg^-1]
        Cim             6.9972e-11;    // Intramyocardial compliance [m^4 s^2 kg^-1]
        PimScaling      1.5;           // Intramyocardial pressure scaling (1.5 for LCA, 0.5 for RCA)
        Pv              0;             // Distal pressure [Pa]
        diffScheme      firstOrder;
        PimFile         "$FOAM_CASE/DataFiles/PimData"; // File containing intramyocardial pressure
     data
        value           uniform 0; // Initial value for pressure [Pa]
    }

    walls
    {
        type            zeroGradient;
    }
```

```
}

// ************************************************************************* //
```

xi. `p.Coronary.secondOrder` BC dictionary: It implements the `coronaryOutletPressure` boundary condition with `secondOrder` differencing scheme on a straight LCA geometry's outlet.

`p.Coronary.secondOrder` BC dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type            coronaryOutletPressure;
        Ra              1.0179e+10;    // Arterial resistance [kg m^-4 s^-1]
        Ram             1.6541e+10;    // Micro-arterial resistance [kg m^-4 s^-1]
        Rv              5.0896e+09;    // Veinous resistance [kg m^-4 s^-1]
        Rvm             0;             // Micro-veinous resistance [kg m^-4 s^-1]
        Ca              8.6482e-12;    // Arterial compliance [m^4 s^2 kg^-1]
        Cim             6.9972e-11;    // Intramyocardial compliance [m^4 s^2 kg^-1]
        PimScaling      1.5;           // Intramyocardial pressure scaling (1.5 for LCA, 0.5 for RCA)
        Pv              0;             // Distal pressure [Pa]
        diffScheme      secondOrder;
        PimFile         "$FOAM_CASE/DataFiles/PimData"; // File containing intramyocardial pressure
     data
        value           uniform 0; // Initial value for pressure [Pa]
    }

    walls
    {
        type            zeroGradient;
    }

}

// ************************************************************************* //
```

B.3.2 `constant` directory

The `constant` directory inside `LPNBCTestCase` master directory contains the dictionaries describing the physics and properties of the simulation for all the case in the `solids4Foam` environment.

B.3.2.1 `polyMesh` directory

It contains the `.m4` scripts required to create `blockMeshDict` dictionary to generate a specific geometry and mesh of straight aorta or LCA for all the cases.

i. `blockMeshDict.m4.WK` script: It is used to create `blockMeshDict` dictionary to generate geometry and mesh for straight cylindrical aortic section.

`blockMeshDict.m4.WK` script

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    `format'    ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
// General macros to create cylinder mesh

changecom(//)changequote([,])
define(calc, [esyscmd(perl -e 'use Math::Trig; print ($1)')])
define(VCOUNT, 0)
define(vlabel, [[// ]Vertex $1 = VCOUNT define($1, VCOUNT)define([VCOUNT], incr(VCOUNT))])

define(hex2D, hex ($1b $2b $3b $4b $1t $2t $3t $4t))
define(btQuad, ($1b $2b $2t $1t))
define(topQuad, ($1t $4t $3t $2t))
define(bottomQuad, ($1b $2b $3b $4b))

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.001;

// Inner square side half
define(s, 7.50)

// Inner square side curvature
define(sc, 8)

// cylinder radius
define(r, 15)

// Height of cylinder
define(z, 120.0)

// Base z
define(Zb, 0)

// Outlet z
define(Zt, calc(Zb + z))

// Number of cells at inner square
define(Ns, 20)
```

```
// Number of cells between inner square and circle
define(Ni, 10)

// Number of cells in the cylinder height
define(Nz, 60)

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

define(vert, (x$1$2 y$1$2 $3))
define(evert, (ex$1$2 ey$1$2 $3))

// 45 degree points angle
define(a0, -45)
define(a1, -135)
define(a2, 135)
define(a3, 45)

// Half of 45 degree points angle
define(ea0, 0)
define(ea1, -90)
define(ea2, 180)
define(ea3, 90)

define(ca0, calc(cos((pi/180)*a0)))
define(ca1, calc(cos((pi/180)*a1)))
define(ca2, calc(cos((pi/180)*a2)))
define(ca3, calc(cos((pi/180)*a3)))

define(sa0, calc(sin((pi/180)*a0)))
define(sa1, calc(sin((pi/180)*a1)))
define(sa2, calc(sin((pi/180)*a2)))
define(sa3, calc(sin((pi/180)*a3)))

define(cea0, calc(cos((pi/180)*ea0)))
define(cea1, calc(cos((pi/180)*ea1)))
define(cea2, calc(cos((pi/180)*ea2)))
define(cea3, calc(cos((pi/180)*ea3)))

define(sea0, calc(sin((pi/180)*ea0)))
define(sea1, calc(sin((pi/180)*ea1)))
define(sea2, calc(sin((pi/180)*ea2)))
define(sea3, calc(sin((pi/180)*ea3)))

// Inner square x and y position

// x
define(x00, s)
define(x01, calc(-1.0*s))
define(x02, calc(-1.0*s))
define(x03, s)

// y
define(y00, calc(-1.0*s))
define(y01, calc(-1.0*s))
define(y02, s)
define(y03, s)

// Circle x and y positions

// x
define(x10, calc(r*ca0))
define(x11, calc(r*ca1))
define(x12, calc(r*ca2))
define(x13, calc(r*ca3))

// y
define(y10, calc(r*sa0))
```

```
define(y11, calc(r*sa1))
define(y12, calc(r*sa2))
define(y13, calc(r*sa3))

// Inner square x and y position middle curvatures

// x
define(ex00, sc)
define(ex01, 0)
define(ex02, calc(-1.0*sc))
define(ex03, 0)

// y
define(ey00, 0)
define(ey01, calc(-1.0*sc))
define(ey02, 0)
define(ey03, sc)

// Circle x and y positions middle curvatures

// x
define(ex10, calc(r*cea0))
define(ex11, calc(r*cea1))
define(ex12, calc(r*cea2))
define(ex13, calc(r*cea3))

// y
define(ey10, calc(r*sea0))
define(ey11, calc(r*sea1))
define(ey12, calc(r*sea2))
define(ey13, calc(r*sea3))

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

vertices
(
    vert(0, 0, Zb) vlabel(s0b)
    vert(0, 1, Zb) vlabel(s1b)
    vert(0, 2, Zb) vlabel(s2b)
    vert(0, 3, Zb) vlabel(s3b)

    vert(1, 0, Zb) vlabel(r0b)
    vert(1, 1, Zb) vlabel(r1b)
    vert(1, 2, Zb) vlabel(r2b)
    vert(1, 3, Zb) vlabel(r3b)

    vert(0, 0, Zt) vlabel(s0t)
    vert(0, 1, Zt) vlabel(s1t)
    vert(0, 2, Zt) vlabel(s2t)
    vert(0, 3, Zt) vlabel(s3t)

    vert(1, 0, Zt) vlabel(r0t)
    vert(1, 1, Zt) vlabel(r1t)
    vert(1, 2, Zt) vlabel(r2t)
    vert(1, 3, Zt) vlabel(r3t)
);

blocks
(
    //block0
    hex2D(s1, s0, s3, s2)
    square
    (Ns Ns Nz)
    simpleGrading (1 1 1)

    //block1
    hex2D(s0, r0, r3, s3)
    innerCircle
```

```
    (Ni Ns Nz)
    simpleGrading (1 1 1)

    //block2
    hex2D(s3, r3, r2, s2)
    innerCircle
    (Ni Ns Nz)
    simpleGrading (1 1 1)

    //block3
    hex2D(s2, r2, r1, s1)
    innerCircle
    (Ni Ns Nz)
    simpleGrading (1 1 1)

    //block4
    hex2D(s1, r1, r0, s0)
    innerCircle
    (Ni Ns Nz)
    simpleGrading (1 1 1)
);

edges
(
    //Circle edges
    arc r3b r0b evert(1, 0, Zb)
    arc r0b r1b evert(1, 1, Zb)
    arc r1b r2b evert(1, 2, Zb)
    arc r2b r3b evert(1, 3, Zb)

    //Circle edges
    arc r3t r0t evert(1, 0, Zt)
    arc r0t r1t evert(1, 1, Zt)
    arc r1t r2t evert(1, 2, Zt)
    arc r2t r3t evert(1, 3, Zt)

    arc s3b s0b evert(0, 0, Zb)
    arc s0b s1b evert(0, 1, Zb)
    arc s1b s2b evert(0, 2, Zb)
    arc s2b s3b evert(0, 3, Zb)

    arc s3t s0t evert(0, 0, Zt)
    arc s0t s1t evert(0, 1, Zt)
    arc s1t s2t evert(0, 2, Zt)
    arc s2t s3t evert(0, 3, Zt)

);

patches
(
    wall walls
    (
        btQuad(r0, r3)
        btQuad(r1, r0)
        btQuad(r2, r1)
        btQuad(r3, r2)
    )

    patch inlet
    (
        bottomQuad(s3, s0, s1, s2)
        bottomQuad(s3, r3, r0, s0)
        bottomQuad(s2, r2, r3, s3)
        bottomQuad(s1, r1, r2, s2)
        bottomQuad(s0, r0, r1, s1)
    )

    patch outlet
```

```
    (
        topQuad(s3, s0, s1, s2)
        topQuad(s3, r3, r0, s0)
        topQuad(s2, r2, r3, s3)
        topQuad(s1, r1, r2, s2)
        topQuad(s0, r0, r1, s1)
    )
);

mergePatchPairs
(
);
```

ii. `blockMeshDict.m4.Coronary` script: It is used to create `blockMeshDict` dictionary to generate geometry and mesh for straight cylindrical LCA section.

`blockMeshDict.m4.Coronary` script

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    `format'    ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
// General macros to create cylinder mesh

changecom(//)changequote([,])
define(calc, [esyscmd(perl -e 'use Math::Trig; print ($1)')])
define(VCOUNT, 0)
define(vlabel, [[// ]Vertex $1 = VCOUNT define($1, VCOUNT)define([VCOUNT], incr(VCOUNT))])

define(hex2D, hex ($1b $2b $3b $4b $1t $2t $3t $4t))
define(btQuad, ($1b $2b $2t $1t))
define(topQuad, ($1t $4t $3t $2t))
define(bottomQuad, ($1b $2b $3b $4b))

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.001;

// Inner square side half
define(s, 0.750)

// Inner square side curvature
define(sc, 0.8)

// cylinder radius
define(r, 1.50)

// Height of cylinder
define(z, 12.0)

// Base z
define(Zb, 0)

// Outlet z
define(Zt, calc(Zb + z))
```

```
// Number of cells at inner square
define(Ns, 20)

// Number of cells between inner square and circle
define(Ni, 10)

// Number of cells in the cylinder height
define(Nz, 60)

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

define(vert, (x$1$2 y$1$2 $3))
define(evert, (ex$1$2 ey$1$2 $3))

// 45 degree points angle
define(a0, -45)
define(a1, -135)
define(a2, 135)
define(a3, 45)

// Half of 45 degree points angle
define(ea0, 0)
define(ea1, -90)
define(ea2, 180)
define(ea3, 90)

define(ca0, calc(cos((pi/180)*a0)))
define(ca1, calc(cos((pi/180)*a1)))
define(ca2, calc(cos((pi/180)*a2)))
define(ca3, calc(cos((pi/180)*a3)))

define(sa0, calc(sin((pi/180)*a0)))
define(sa1, calc(sin((pi/180)*a1)))
define(sa2, calc(sin((pi/180)*a2)))
define(sa3, calc(sin((pi/180)*a3)))

define(cea0, calc(cos((pi/180)*ea0)))
define(cea1, calc(cos((pi/180)*ea1)))
define(cea2, calc(cos((pi/180)*ea2)))
define(cea3, calc(cos((pi/180)*ea3)))

define(sea0, calc(sin((pi/180)*ea0)))
define(sea1, calc(sin((pi/180)*ea1)))
define(sea2, calc(sin((pi/180)*ea2)))
define(sea3, calc(sin((pi/180)*ea3)))

// Inner square x and y position

// x
define(x00, s)
define(x01, calc(-1.0*s))
define(x02, calc(-1.0*s))
define(x03, s)

// y
define(y00, calc(-1.0*s))
define(y01, calc(-1.0*s))
define(y02, s)
define(y03, s)

// Circle x and y positions

// x
define(x10, calc(r*ca0))
define(x11, calc(r*ca1))
define(x12, calc(r*ca2))
define(x13, calc(r*ca3))
```

```
// y
define(y10, calc(r*sa0))
define(y11, calc(r*sa1))
define(y12, calc(r*sa2))
define(y13, calc(r*sa3))

// Inner square x and y position middle curvatures

// x
define(ex00, sc)
define(ex01, 0)
define(ex02, calc(-1.0*sc))
define(ex03, 0)

// y
define(ey00, 0)
define(ey01, calc(-1.0*sc))
define(ey02, 0)
define(ey03, sc)

// Circle x and y positions middle curvatures

// x
define(ex10, calc(r*cea0))
define(ex11, calc(r*cea1))
define(ex12, calc(r*cea2))
define(ex13, calc(r*cea3))

// y
define(ey10, calc(r*sea0))
define(ey11, calc(r*sea1))
define(ey12, calc(r*sea2))
define(ey13, calc(r*sea3))

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

vertices
(
    vert(0, 0, Zb) vlabel(s0b)
    vert(0, 1, Zb) vlabel(s1b)
    vert(0, 2, Zb) vlabel(s2b)
    vert(0, 3, Zb) vlabel(s3b)

    vert(1, 0, Zb) vlabel(r0b)
    vert(1, 1, Zb) vlabel(r1b)
    vert(1, 2, Zb) vlabel(r2b)
    vert(1, 3, Zb) vlabel(r3b)

    vert(0, 0, Zt) vlabel(s0t)
    vert(0, 1, Zt) vlabel(s1t)
    vert(0, 2, Zt) vlabel(s2t)
    vert(0, 3, Zt) vlabel(s3t)

    vert(1, 0, Zt) vlabel(r0t)
    vert(1, 1, Zt) vlabel(r1t)
    vert(1, 2, Zt) vlabel(r2t)
    vert(1, 3, Zt) vlabel(r3t)
);

blocks
(
    //block0
    hex2D(s1, s0, s3, s2)
    square
    (Ns Ns Nz)
    simpleGrading (1 1 1)

    //block1
```

```
    hex2D(s0, r0, r3, s3)
    innerCircle
    (Ni Ns Nz)
    simpleGrading (1 1 1)

    //block2
    hex2D(s3, r3, r2, s2)
    innerCircle
    (Ni Ns Nz)
    simpleGrading (1 1 1)

    //block3
    hex2D(s2, r2, r1, s1)
    innerCircle
    (Ni Ns Nz)
    simpleGrading (1 1 1)

    //block4
    hex2D(s1, r1, r0, s0)
    innerCircle
    (Ni Ns Nz)
    simpleGrading (1 1 1)
);

edges
(
    //Circle edges
    arc r3b r0b evert(1, 0, Zb)
    arc r0b r1b evert(1, 1, Zb)
    arc r1b r2b evert(1, 2, Zb)
    arc r2b r3b evert(1, 3, Zb)

    //Circle edges
    arc r3t r0t evert(1, 0, Zt)
    arc r0t r1t evert(1, 1, Zt)
    arc r1t r2t evert(1, 2, Zt)
    arc r2t r3t evert(1, 3, Zt)

    arc s3b s0b evert(0, 0, Zb)
    arc s0b s1b evert(0, 1, Zb)
    arc s1b s2b evert(0, 2, Zb)
    arc s2b s3b evert(0, 3, Zb)

    arc s3t s0t evert(0, 0, Zt)
    arc s0t s1t evert(0, 1, Zt)
    arc s1t s2t evert(0, 2, Zt)
    arc s2t s3t evert(0, 3, Zt)

);

patches
(
    wall walls
    (
        btQuad(r0, r3)
        btQuad(r1, r0)
        btQuad(r2, r1)
        btQuad(r3, r2)
    )

    patch inlet
    (
        bottomQuad(s3, s0, s1, s2)
        bottomQuad(s3, r3, r0, s0)
        bottomQuad(s2, r2, r3, s3)
        bottomQuad(s1, r1, r2, s2)
        bottomQuad(s0, r0, r1, s1)
    )
```

```
    patch outlet
    (
        topQuad(s3, s0, s1, s2)
        topQuad(s3, r3, r0, s0)
        topQuad(s2, r2, r3, s3)
        topQuad(s1, r1, r2, s2)
        topQuad(s0, r0, r1, s1)
    )
);

mergePatchPairs
(
);
```

B.3.2.2 `physicsProperties` dictionary

It is required to define whether the simulation is for fluid, solids or fluid-solid interaction (FSI).

`physicsProperties` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      physicsProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

type        fluid;


// ************************************************************************* //
```

B.3.2.3 `dynamicMeshDict` dictionary

It is required to define whether the mesh have motion or not.

`dynamicMeshDict` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      dynamicMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dynamicFvMesh       staticFvMesh;
```

```
// ************************************************************************* //
```

B.3.2.4 transportProperties dictionary

It is required to define the properties of the fluid.

transportProperties dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

transportModel      Newtonian;

rho                 rho [1 -3 0 0 0 0 0] 1060;
nu                  nu [0 2 -1 0 0 0 0] 3.77e-6;

// ************************************************************************* //
```

B.3.2.5 turbulenceProperties dictionary

It is required to define the type of fluid simulation.

turbulenceProperties dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

simulationType      RASModel;

// ************************************************************************* //
```

B.3.2.6 RASProperties dictionary

It is required to define the type of RAS model to be used for fluid simulation.

`RASProperties` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

RASModel        laminar;

turbulence      off;

printCoeffs     on;

// ************************************************************************* //
```

B.3.2.7 `fluidProperties` dictionary

It is required to define the type of solver to be used for fluid simulation.

`fluidProperties` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fluidProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

fluidModel          pimpleFluid;

pimpleFluidCoeffs
{
    ddtCorr         false;
    adjustPhi       true;
    solveEnergyEq   false;
}

// ************************************************************************* //
```

B.3.3 `system` directory

The `system` directory inside `LPNBCTestCase` master directory contains the dictionaries used to set the discretization schemes, solvers, decomposition and controls for all the cases.

B.3.3.1 `fvSchemes` dictionary

It is used to define the discretization schemes for various mathematical operators used in the governing equations of the simulation.

`fvSchemes` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         leastSquares;
}

divSchemes
{
    default                         none;
    div(phi,U)                      Gauss linearUpwind cellLimited leastSquares 1;
    div((nuEff*dev(grad(U).T())))   Gauss linear;
    div((nuEff*dev(T(grad(U)))))    Gauss linear;

    div(phi,epsilon)                Gauss linear;
    div(phi,k)                      Gauss linear;
}

laplacianSchemes
{
    default                         none;
    laplacian(nu,U)                 Gauss linear corrected;
    laplacian(nuEff,U)              Gauss linear corrected;

    laplacian(rAU,p)                Gauss linear corrected;

    laplacian((1|A(U)),p)           Gauss linear skewCorrected 0.5;

    laplacian(DepsilonEff,epsilon)  Gauss linear corrected;
    laplacian(DkEff,k)              Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
    interpolate(U)  linear;
}

snGradSchemes
{
    default         skewCorrected 0.5;
```

```
}

// ************************************************************************* //
```

B.3.3.2 `fvSolution` dictionary

It is used to define the solution algorithms and solver settings for the numerical simulation.

`fvSolution` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    "p|pFinal"
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-7;
        relTol          0;
        minIter         1;
    }

    "U|UFinal|k|epsilon"
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-7;
        relTol          0;
        minIter         1;
    }
}

PISO
{
    nCorrectors                 3;
    nNonOrthogonalCorrectors    1;
}

PIMPLE
{
    nOuterCorrectors            1; // Iterative PISO
    nCorrectors                 2;
    nNonOrthogonalCorrectors    1;

    residualControl
    {
        U
        {
            relTol      0;
            tolerance   1e-7;
        }
```

```
        p
        {
            relTol      0;
            tolerance   1e-7;
        }
    }

}

/*
relaxationFactors
{
    equations
    {
        U    0.7;
        UFinal 0.7;
    }
    fields
    {
        p    0.3;
        pFinal    0.3;
    }
}
*/

// ************************************************************************* //
```

B.3.3.3 `decomposeParDict` dictionary

It is used to define the settings for domain decomposition, which is necessary for running parallel simulations.

`decomposeParDict` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      decomposeParDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

numberOfSubdomains      8;

method                  simple;

simpleCoeffs
{
    n           (1 1 8);
    delta       0.001;
}

// ************************************************************************* //
```

B.3.3.4 `controlDict` dictionary

It serves as the primary configuration file for controlling the simulation's execution and output.

`controlDict` dictionary

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | foam-extend: Open Source CFD                    |
|  \\    /   O peration     | Version:     4.1                                |
|   \\  /    A nd           | Web:         http://www.foam-extend.org         |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

libs                ("liblumpedParameterNetworkBC.so");

application         solids4Foam;

startFrom           latestTime;

startTime           0;

stopAt              endTime;

endTime             8;

deltaT              0.001;

writeControl        timeStep;

writeInterval       50;

purgeWrite          0;

writeFormat         ascii;

writePrecision      9;

writeCompression    uncompressed;

timeFormat          general;

timePrecision       6;

runTimeModifiable   yes;

adjustTimeStep      no;

maxCo               0.5;


functions
(

    flowRateInlet
    {
        type            faceSource;
        functionObjectLibs ("libfieldFunctionObjects.so");
        enabled         true;
        outputControl   timeStep;
        log             true;
        valueOutput     true;
        source          patch;
```

```
        sourceName      inlet;
        operation       sum;
        surfaceFormat   off;

        fields
        (
            phi
        );
    }

    flowRateOutlet
    {
        type            faceSource;
        functionObjectLibs ("libfieldFunctionObjects.so");
        enabled         true;
        outputControl   timeStep;
        log             true;
        valueOutput     true;
        source          patch;
        sourceName      outlet;
        operation       sum;
        surfaceFormat   off;

        fields
        (
            phi
        );
    }

    pAverageInlet
      {
         type            faceSource;
         functionObjectLibs ("libfieldFunctionObjects.so");
         enabled         true;
         outputControl   timeStep;
         log             true;
         valueOutput     true;
         source          patch;
         sourceName      inlet;
         operation      areaAverage;
        surfaceFormat   off;
        fields
        (
            p
        );
      }

    pAverageOutlet
      {
         type            faceSource;
         functionObjectLibs ("libfieldFunctionObjects.so");
         enabled         true;
         outputControl   timeStep;
         log             true;
         valueOutput     true;
         source          patch;
         sourceName      outlet;
         operation       areaAverage;
        surfaceFormat   off;
        fields
        (
            p
        );
      }

);
```

```
// ********************************************************************* //
```

B.3.4 DataFiles directory

The DataFiles directory inside LPNBCTestCase master directory contains the files with inlet flow rate time-series data for the aorta simulation cases using windkessek model BCs, and inlet flow rate as well as intramyocardial pressure time-series data for the LCA simulation cases using coronary LPN model BCs.

B.3.4.1 AorticInletFlowRate file

It contains the volume flow rate data for aorta as a time-series.

Listing B.1: AorticInletFlowRate file

```
(
( 0.00000 5.25792000E-05 )
( 0.00100 5.51130000E-05 )
( 0.00200 5.76468200E-05 )
( 0.00300 6.01806300E-05 )
.
.
.
( 0.99800 5.04552600E-05 )
( 0.99900 5.15172200E-05 )
( 1.00000 5.25792000E-05 )
);
```

B.3.4.2 CoronaryInletFlowRate file

It contains the volume flow rate data for LCA as a time-series.

Listing B.2: CoronaryInletFlowRate file

```
(
( 0.00000 3.56788000E-07 )
( 0.00100 3.57129822E-07 )
( 0.00200 3.57471644E-07 )
( 0.00300 3.57813465E-07 )
.
.
.
( 0.99710 3.51495783E-07 )
( 0.99810 3.51964391E-07 )
( 0.99910 3.52433000E-07 )
);
```

B.3.4.3 PimData file

It contains the intramyocardial pressure data for LCA as a time-series.

Listing B.3: PimData file

```
(
( 0.00000 2.85419827E+03 )
( 0.00100 2.88803679E+03 )
( 0.00200 2.92568009E+03 )
( 0.00300 2.96726253E+03 )
.
.
.
```

```
( 0.99710 2.78917846E+03 )
( 0.99810 2.81575383E+03 )
( 0.99910 2.81578041E+03 )
);
```

B.3.5 `Allrun` scripts

A bash script is used to run the simulation in all the cases. They are written to execute all the commands necessary to set up the case, create mesh, create partitions, resolve any data parsing issues in the dictionaries upon partitioning, run simulation, and merge partitions in `solids4Foam`.

B.3.5.1 `Allrun.WK` script

It is used to run all the aorta simulation cases with windkessel model BCs.

`Allrun.WK` script

```
#!/bin/bash

# Source tutorial clean functions
. $WM_PROJECT_DIR/bin/tools/RunFunctions

# Source solids4Foam scripts
source solids4FoamScripts.sh

# Check case version is correct
solids4Foam::convertCaseFormat .

# Run only with foam-extend
solids4Foam::caseOnlyRunsWithFoamExtend

# Create blockMeshDict from m4 script
m4 constant/polyMesh/blockMeshDict.m4 > constant/polyMesh/blockMeshDict

# Run blockMesh
solids4Foam::runApplication blockMesh

# Run checkMesh
solids4Foam::runApplication checkMesh

# Run decomposePar
solids4Foam::runApplication decomposePar -force

# Fix the parcing issues in the 0/U files in all processor directories
sed -i 's/fileName\s\{1,\}\("$FOAM_CASE\/DataFiles\/AorticInletFlowRate"\)/"file|fileName"
      \1/' processor*/0/U

# Run solids4Foam simulation in parallel
mpirun -np 8 solids4Foam -parallel > log.solids4Foam

# Run reconstructPar
solids4Foam::runApplication reconstructPar

# Create residual files
foamLog log.solids4Foam

# Create file to for the visualization in ParaView
touch AortaStraight.foam
```

B.3.5.2 `Allrun.Coronary` script

It is used to run all the LCA simulation cases with coronary LPN model BCs.

`Allrun.Coronary` script

```
#!/bin/bash

# Source tutorial clean functions
. $WM_PROJECT_DIR/bin/tools/RunFunctions

# Source solids4Foam scripts
source solids4FoamScripts.sh

# Check case version is correct
solids4Foam::convertCaseFormat .

# Run only with foam-extend
solids4Foam::caseOnlyRunsWithFoamExtend

# Create blockMeshDict from m4 script
m4 constant/polyMesh/blockMeshDict.m4 > constant/polyMesh/blockMeshDict

# Run blockMesh
solids4Foam::runApplication blockMesh

# Run checkMesh
solids4Foam::runApplication checkMesh

# Run decomposePar
solids4Foam::runApplication decomposePar -force

# Fix the parcing issues in the 0/U files in all processor directories
sed -i 's/fileName\s\{1,\}\("$FOAM_CASE\/DataFiles\/CoronaryInletFlowRate"\)/"file|fileName"
        \1/' processor*/0/U

# Run solids4Foam simulation in parallel
mpirun -np 8 solids4Foam -parallel > log.solids4Foam

# Run reconstructPar
solids4Foam::runApplication reconstructPar

# Create residual files
foamLog log.solids4Foam

# Create file to for the visualization in ParaView
touch CoronaryStraight.foam
```

B.3.6 `Allclean` scripts

`Allclean` bash scripts can be executed to quickly clean the solution and log files, and restore the cases in their original form to rerun the simulations.

B.3.6.1 `Allclean.WK` script

It is used to clean all the aorta simulation cases with windkessel model BCs.

`Allclean.WK` script

```
#!/bin/bash

# Source tutorial clean functions
. $WM_PROJECT_DIR/bin/tools/CleanFunctions

# Source solids4Foam scripts
source solids4FoamScripts.sh

cleanCase
cleanTimeDirectories
\rm -f constant/polyMesh/boundary
```

```
\rm -f constant/polyMesh/blockMeshDict
\rm -rf processor*
\rm -f *.eps
\rm -f *.ps
\rm -rf history
\rm -rf VTK
\rm -rf logs
\rm -rf postProcessing
\rm -f AortaStraight.foam
```

B.3.6.2 Allrun.Coronary script

It is used to clean all the LCA simulation cases with coronary LPN model BCs.

Allclean.Coronary script

```
#!/bin/bash

# Source tutorial clean functions
. $WM_PROJECT_DIR/bin/tools/CleanFunctions

# Source solids4Foam scripts
source solids4FoamScripts.sh

cleanCase
cleanTimeDirectories
\rm -f constant/polyMesh/boundary
\rm -f constant/polyMesh/blockMeshDict
\rm -rf processor*
\rm -f *.eps
\rm -f *.ps
\rm -rf history
\rm -rf VTK
\rm -rf logs
\rm -rf postProcessing
\rm -f CoronaryStraight.foam
```

B.4 CreateDirStruc and DeleteDirStruc scripts

CreateDirStruc bash script is written to set up the basic OpenFOAM/solids4Foam directory structure for all the simulation cases by copying and renaming the files required for each case into a new directory for that case so that simulation can be run.

CreateDirStruc script

```
#!/bin/bash

echo "Creating directory structure..."

# Base directory name
base_dir="LPNBCTestCase"

# Array of simulation types and schemes
simulations=("Resistive" "WK2" "WK3" "WK4Series" "WK4Parallel" "Coronary")
schemes=("firstOrder" "secondOrder")

# Function to copy and create directory structure
create_simulation_structure() {
    sim=$1
    scheme=$2

    # Define new simulation directory
    sim_dir="${base_dir}_${sim}_${scheme}"
```

```
    echo "Creating directory structure for ${sim_dir}..."

    # Create base directories
    mkdir -p "${sim_dir}/0" "${sim_dir}/constant/polyMesh" "${sim_dir}/system" "${sim_dir}/DataFiles
    "

    # Handle U file
    if [[ "${sim}" == "Coronary" ]]; then
        cp "${base_dir}/0.orig/U.Coronary" "${sim_dir}/0/U"
    else
        cp "${base_dir}/0.orig/U.WK" "${sim_dir}/0/U"
    fi

    # Handle p file
    if [[ "${sim}" == "Resistive" ]]; then
        cp "${base_dir}/0.orig/p.Resistive" "${sim_dir}/0/p"
    else
        cp "${base_dir}/0.orig/p.${sim}.${scheme}" "${sim_dir}/0/p"
    fi

    # Copy constant files
    cp -r "${base_dir}/constant/"* "${sim_dir}/constant/"
    rm -r "${sim_dir}/constant/polyMesh/blockMeshDict.m4."*

    # Handle blockMeshDict
    if [[ "${sim}" == "Coronary" ]]; then
        cp "${base_dir}/constant/polyMesh/blockMeshDict.m4.Coronary" "${sim_dir}/constant/polyMesh/
    blockMeshDict.m4"
    else
        cp "${base_dir}/constant/polyMesh/blockMeshDict.m4.WK" "${sim_dir}/constant/polyMesh/
    blockMeshDict.m4"
    fi

    # Handle DataFiles
    if [[ "${sim}" == "Coronary" ]]; then
        cp "${base_dir}/DataFiles/CoronaryInletFlowRate" "${sim_dir}/DataFiles/"
        cp "${base_dir}/DataFiles/PimData" "${sim_dir}/DataFiles/"
    else
        cp "${base_dir}/DataFiles/AorticInletFlowRate" "${sim_dir}/DataFiles/"
    fi

    # Copy system files
    cp -r "${base_dir}/system/"* "${sim_dir}/system/"

    # Handle Allrun and Allclean scripts
    if [[ "${sim}" == "Coronary" ]]; then
        cp "${base_dir}/Allrun.Coronary" "${sim_dir}/Allrun"
        cp "${base_dir}/Allclean.Coronary" "${sim_dir}/Allclean"
        chmod +x ${sim_dir}/Allrun ${sim_dir}/Allclean
    else
        cp "${base_dir}/Allrun.WK" "${sim_dir}/Allrun"
        cp "${base_dir}/Allclean.WK" "${sim_dir}/Allclean"
        chmod +x ${sim_dir}/Allrun ${sim_dir}/Allclean
    fi
}

# Loop through all simulations and schemes
for sim in "${simulations[@]}"; do
    if [[ "${sim}" == "Resistive" ]]; then
        create_simulation_structure "$sim" ""
    else
        for scheme in "${schemes[@]}"; do
            create_simulation_structure "$sim" "$scheme"
        done
    fi
done

echo "Directory structure creation complete!"
```

DeleteDirStruc script can be used to delete all the newly created case directories and leave only the master case folder.

DeleteDirStruc script

```
#!/bin/bash

echo "Deleting directory structure..."

rm -r LPNBCTestCase_*

echo "Directory structure deletion completed!"
```

B.5 Main Allrun and Allclean scripts

The main Allrun script can be executed to run all the cases one after another.

Main Allrun script

```
#!/bin/bash

echo "Compiling lumpedParameterNetworkBC..."

cd lumpedParameterNetworkBC/

./Allwmake

cd ../

echo "Running all cases, one by one..."

cd LPNBCTestCase_Resistive_/

echo "Running LPNBCTestCase_Resistive_ case..."

./Allrun

cd ../LPNBCTestCase_WK2_firstOrder

echo "Running LPNBCTestCase_WK2_firstOrder case..."

./Allrun

cd ../LPNBCTestCase_WK2_secondOrder

echo "Running LPNBCTestCase_WK2_secondOrder case..."

./Allrun

cd ../LPNBCTestCase_WK3_firstOrder

echo "Running LPNBCTestCase_WK3_firstOrder case..."

./Allrun

cd ../LPNBCTestCase_WK3_secondOrder

echo "Running LPNBCTestCase_WK3_secondOrder case..."

./Allrun

cd ../LPNBCTestCase_WK4Series_firstOrder

echo "Running LPNBCTestCase_WK4Series_firstOrder case..."

./Allrun
```

```
cd ../LPNBCTestCase_WK4Series_secondOrder

echo "Running LPNBCTestCase_WK4Series_secondOrder case..."

./Allrun

cd ../LPNBCTestCase_WK4Parallel_firstOrder

echo "Running LPNBCTestCase_WK4Parallel_firstOrder case..."

./Allrun

cd ../LPNBCTestCase_WK4Parallel_secondOrder

echo "Running LPNBCTestCase_WK4Parallel_secondOrder case..."

./Allrun

cd ../LPNBCTestCase_Coronary_firstOrder

echo "Running LPNBCTestCase_Coronary_firstOrder case..."

./Allrun

cd ../LPNBCTestCase_Coronary_secondOrder

echo "Running LPNBCTestCase_Coronary_secondOrder case..."

./Allrun

cd ../

echo "All simulations are completed!"

echo "Current directory:"

pwd

echo "Run the MATLAB scripts in MATLABPostprocessing folder for the postprocessing!"
```

The main `Allclean` script can be used to clean all the cases one after another.

Main `Allclean` script

```
#!/bin/bash

echo "Cleaning lumpedParameterNetworkBC..."

cd lumpedParameterNetworkBC/

./Allwclean

cd ../

echo "Cleaning all cases, one by one..."

cd LPNBCTestCase_Resistive_/

echo "Cleaning LPNBCTestCase_Resistive_ case..."

./Allclean

cd ../LPNBCTestCase_WK2_firstOrder

echo "Cleaning LPNBCTestCase_WK2_firstOrder case..."

./Allclean
```

```
cd ../LPNBCTestCase_WK2_secondOrder
echo "Cleaning LPNBCTestCase_WK2_secondOrder case..."
./Allclean
cd ../LPNBCTestCase_WK3_firstOrder
echo "Cleaning LPNBCTestCase_WK3_firstOrder case..."
./Allclean
cd ../LPNBCTestCase_WK3_secondOrder
echo "Cleaning LPNBCTestCase_WK3_secondOrder case..."
./Allclean
cd ../LPNBCTestCase_WK4Series_firstOrder
echo "Cleaning LPNBCTestCase_WK4Series_firstOrder case..."
./Allclean
cd ../LPNBCTestCase_WK4Series_secondOrder
echo "Cleaning LPNBCTestCase_WK4Series_secondOrder case..."
./Allclean
cd ../LPNBCTestCase_WK4Parallel_firstOrder
echo "Cleaning LPNBCTestCase_WK4Parallel_firstOrder case..."
./Allclean
cd ../LPNBCTestCase_WK4Parallel_secondOrder
echo "Cleaning LPNBCTestCase_WK4Parallel_secondOrder case..."
./Allclean
cd ../LPNBCTestCase_Coronary_firstOrder
echo "Cleaning LPNBCTestCase_Coronary_firstOrder case..."
./Allclean
cd ../LPNBCTestCase_Coronary_secondOrder
echo "Cleaning LPNBCTestCase_Coronary_secondOrder case..."
./Allclean
cd ../
echo "All cases are cleaned!"
echo "Cleaning Postprocessing Results..."
cd MATLABPostprocessing/
\rm -f *.eps *.png *.jpg
echo "Postprocessing Results are cleaned!"
cd ../
```

```
echo "Current directory:"

pwd
```

B.6 MATLAB postprocessing script

After all the simulations are completed, a MATLAB script `LPNBCTestCasePostprocessing.m` present in the `MATLABPostprocessing` directory within the master directory can be run to generate the plots comparing the results of OpenFOAM simulations with those obtained through numerical solution of LPN model ODEs in MATLAB using the same parameters and conditions as in simulations.

A MATLAB postprocessing script

```
%% LPN Boundary Condition Test Case Postprocessing Script

%{
Author:
    Muhammad Ahmad Raza, University College Dublin, Ireland.

Cite as:
    Raza, M. A.: Implementation of Lumped Parameter Network Boundary Conditions for the
    Patient-Specific CFD Simulations of Coronary Arteries in OpenFOAM. In Proceedings of CFD with
    OpenSource Software, 2024, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2024
%}

%% Clear All

clc;
clear all;
close all;
%% Parameters Values

N = 8; %Number of cycles

%Resistive model parameters
R_Rd = 17690000;
R_Pd = 9.7363e+03;

%2-Element Windkessel model parameters
WK2_Rd = 141520000;
WK2_C = 8.3333e-09;
WK2_Pd = 0;

%3-Element Windkessel model parameters
WK3_Rp = 13997000;
WK3_Rd = 141520000;
WK3_C = 1.0000e-08;
WK3_Pd = 0;

%4-Element (Series) Windkessel model parameters
WK4s_Rp = 13997000;
WK4s_Rd = 141520000;
WK4s_C = 2.5000e-08;
WK4s_L = 1.7995e+04;
WK4s_Pd = 0;

%4-Element (Parallel) Windkessel model parameters
WK4p_Rp = 1.3997e+10;
WK4p_Rd = 141520000;
WK4p_C = 1.0000e-08;
WK4p_L = 1.7995e+03;
WK4p_Pd = 0;

%Coronary LPN Model parameters
```

```
Ra = 1.0179e+10; %Arterial resistance [kg m^-4 s^-1]
Ram = 1.6541e+10; %Micro-arterial resistance [kg m^-4 s^-1]
Rv = 5.0896e+09; %Veinous resistance [kg m^-4 s^-1]
Rvm = 0; %Micro-veinous resistance [kg m^-4 s^-1]
Ca = 8.6482e-12; %Arterial compliance [m^4 s^2 kg^-1]
Cim = 6.9972e-11; %Intramyocardial compliance [m^4 s^2 kg^-1]
PimScaling = 1.5; %Intramyocardial pressure scaling (1.5 for LCA, 0.5 for RCA)
Pv = 0; %Distal pressure [Pa]

%% Read and prepare Aortic Flow Rate Data

% Open the flow rate data file
AorticFlowfilename = '../LPNBCTestCase/DataFiles/AorticInletFlowRate'; % Replace with your actual file
      name
AorticFlowfileID = fopen(AorticFlowfilename, 'r');

% Read the entire content as a string
AorticFlowrawData = fscanf(AorticFlowfileID, '%c');

% Close the file
fclose(AorticFlowfileID);

% Remove outer parentheses
AorticFlowcleanedData = strrep(AorticFlowrawData, '(', '');
AorticFlowcleanedData = strrep(AorticFlowcleanedData, ')', '');

% Convert to numeric array
AorticFlowData = sscanf(AorticFlowcleanedData, '%f %f', [2, Inf])';

%%
% Repeat the Aortic Flow Rate data for multiple cycles
cycleTime = max(AorticFlowData(:,1)); % Duration of one cycle
AorticFlowTime = []; % Initialize extended time array
AorticFlowRate = []; % Initialize extended flow rate array

for i = 0:(N-1)
    % Offset time for the i-th cycle and avoid duplication at overlap
    newAorticFlowTime = AorticFlowData(:,1) + i * cycleTime;
    newAorticFlowRate = AorticFlowData(:,2);
    if i > 0
        newAorticFlowTime(1) = []; % Remove the first point to avoid duplicate
        newAorticFlowRate(1) = []; % Remove the corresponding flow rate value
    end
    AorticFlowTime = [AorticFlowTime;  newAorticFlowTime];
    AorticFlowRate = [AorticFlowRate;  newAorticFlowRate];
end

%% Read and prepare Coronary Flow Rate Data

% Open the flow rate data file
CoronaryFlowfilename = '../LPNBCTestCase/DataFiles/CoronaryInletFlowRate'; % Replace with your actual
     file name
CoronaryFlowfileID = fopen(CoronaryFlowfilename, 'r');

% Read the entire content as a string
CoronaryFlowrawData = fscanf(CoronaryFlowfileID, '%c');

% Close the file
fclose(CoronaryFlowfileID);

% Remove outer parentheses
CoronaryFlowcleanedData = strrep(CoronaryFlowrawData, '(', '');
CoronaryFlowcleanedData = strrep(CoronaryFlowcleanedData, ')', '');

% Convert to numeric array
CoronaryFlowData = sscanf(CoronaryFlowcleanedData, '%f %f', [2, Inf])';

%%
```

```
% Repeat the Coronary Flow Rate data for multiple cycles
cycleTime = max(CoronaryFlowData(:,1)); % Duration of one cycle
CoronaryFlowTime = []; % Initialize extended time array
CoronaryFlowRate = []; % Initialize extended flow rate array

for i = 0:(N-1)
    % Offset time for the i-th cycle and avoid duplication at overlap
    newCoronaryFlowTime = CoronaryFlowData(:,1) + i * cycleTime;
    newCoronaryFlowRate = CoronaryFlowData(:,2);
    if i > 0
        newCoronaryFlowTime(1) = []; % Remove the first point to avoid duplicate
        newCoronaryFlowRate(1) = []; % Remove the corresponding flow rate value
    end
    CoronaryFlowTime = [CoronaryFlowTime;  newCoronaryFlowTime];
    CoronaryFlowRate = [CoronaryFlowRate;  newCoronaryFlowRate];
end

%% Read and prepare Pim Data File

% Open the Pim data file
Pimfilename = '../LPNBCTestCase/DataFiles/PimData'; % Replace with your actual file name
PimfileID = fopen(Pimfilename, 'r');

% Read the entire content as a string
PimrawData = fscanf(PimfileID, '%c');

% Close the file
fclose(PimfileID);

% Remove outer parentheses
PimcleanedData = strrep(PimrawData, '(', '');
PimcleanedData = strrep(PimcleanedData, ')', '');

% Convert to numeric array
PimData = sscanf(PimcleanedData, '%f %f', [2, Inf])';

%%
% Repeat the Coronary Flow Rate data for multiple cycles
PimcycleTime = max(PimData(:,1)); % Duration of one cycle
PimTime = []; % Initialize extended time array
Pim = []; % Initialize extended flow rate array

for i = 0:(N-1)
    % Offset time for the i-th cycle and avoid duplication at overlap
    newPimTime = PimData(:,1) + i * PimcycleTime;
    newPim = PimData(:,2);
    if i > 0
        newPimTime(1) = []; % Remove the first point to avoid duplicate
        newPim(1) = []; % Remove the corresponding flow rate value
    end
    PimTime = [PimTime;  newPimTime];
    Pim = [Pim;  newPim];
end

%% Solve Windkessel Models

[R_tSol, R_PSol] = Resistive(R_Rd, R_Pd, AorticFlowRate, AorticFlowTime);

[WK2_tSol, WK2_PSol]= WK2(WK2_Rd, WK2_C, WK2_Pd, AorticFlowRate, AorticFlowTime);

[WK3_tSol, WK3_PSol] = WK3(WK3_Rp, WK3_Rd, WK3_C, WK3_Pd, AorticFlowRate, AorticFlowTime);

[WK4s_tSol, WK4s_PSol] = WK4Series(WK4s_Rp, WK4s_Rd, WK4s_C, WK4s_L, WK4s_Pd, AorticFlowRate,
    AorticFlowTime);

[WK4p_tSol, WK4p_PSol] = WK4Parallel(WK4p_Rp, WK4p_Rd, WK4p_C, WK4p_L, WK4p_Pd, AorticFlowRate,
    AorticFlowTime);
```

```
%% Solve Coronary LPN Model

[Coronary_tSol, Coronary_PSol] = CoronaryLPN(Ra, Ram, Rv, Rvm, Ca, Cim, Pv, PimScaling, Pim, PimTime,
    CoronaryFlowRate, CoronaryFlowTime);

%% Read simulation data

R_Qout = table2array(readtable('../LPNBCTestCase_Resistive_/postProcessing/flowRateOutlet/0/faceSource
    .dat'));
R_Pout = table2array(readtable('../LPNBCTestCase_Resistive_/postProcessing/pAverageOutlet/0/faceSource
    .dat'));

WK2_Qout1 = table2array(readtable('../LPNBCTestCase_WK2_firstOrder/postProcessing/flowRateOutlet/0/
    faceSource.dat'));
WK2_Qout2 = table2array(readtable('../LPNBCTestCase_WK2_secondOrder/postProcessing/flowRateOutlet/0/
    faceSource.dat'));
WK2_Pout1 = table2array(readtable('../LPNBCTestCase_WK2_firstOrder/postProcessing/pAverageOutlet/0/
    faceSource.dat'));
WK2_Pout2 = table2array(readtable('../LPNBCTestCase_WK2_secondOrder/postProcessing/pAverageOutlet/0/
    faceSource.dat'));

WK3_Qout1 = table2array(readtable('../LPNBCTestCase_WK3_firstOrder/postProcessing/flowRateOutlet/0/
    faceSource.dat'));
WK3_Qout2 = table2array(readtable('../LPNBCTestCase_WK3_secondOrder/postProcessing/flowRateOutlet/0/
    faceSource.dat'));
WK3_Pout1 = table2array(readtable('../LPNBCTestCase_WK3_firstOrder/postProcessing/pAverageOutlet/0/
    faceSource.dat'));
WK3_Pout2 = table2array(readtable('../LPNBCTestCase_WK3_secondOrder/postProcessing/pAverageOutlet/0/
    faceSource.dat'));

WK4s_Qout1 = table2array(readtable('../LPNBCTestCase_WK4Series_firstOrder/postProcessing/
    flowRateOutlet/0/faceSource.dat'));
WK4s_Qout2 = table2array(readtable('../LPNBCTestCase_WK4Series_secondOrder/postProcessing/
    flowRateOutlet/0/faceSource.dat'));
WK4s_Pout1 = table2array(readtable('../LPNBCTestCase_WK4Series_firstOrder/postProcessing/
    pAverageOutlet/0/faceSource.dat'));
WK4s_Pout2 = table2array(readtable('../LPNBCTestCase_WK4Series_secondOrder/postProcessing/
    pAverageOutlet/0/faceSource.dat'));

WK4p_Qout1 = table2array(readtable('../LPNBCTestCase_WK4Parallel_firstOrder/postProcessing/
    flowRateOutlet/0/faceSource.dat'));
WK4p_Qout2 = table2array(readtable('../LPNBCTestCase_WK4Parallel_secondOrder/postProcessing/
    flowRateOutlet/0/faceSource.dat'));
WK4p_Pout1 = table2array(readtable('../LPNBCTestCase_WK4Parallel_firstOrder/postProcessing/
    pAverageOutlet/0/faceSource.dat'));
WK4p_Pout2 = table2array(readtable('../LPNBCTestCase_WK4Parallel_secondOrder/postProcessing/
    pAverageOutlet/0/faceSource.dat'));

Coronary_Qout1 = table2array(readtable('../LPNBCTestCase_Coronary_firstOrder/postProcessing/
    flowRateOutlet/0/faceSource.dat'));
Coronary_Qout2 = table2array(readtable('../LPNBCTestCase_Coronary_secondOrder/postProcessing/
    flowRateOutlet/0/faceSource.dat'));
Coronary_Pout1 = table2array(readtable('../LPNBCTestCase_Coronary_firstOrder/postProcessing/
    pAverageOutlet/0/faceSource.dat'));
Coronary_Pout2 = table2array(readtable('../LPNBCTestCase_Coronary_secondOrder/postProcessing/
    pAverageOutlet/0/faceSource.dat'));

%% Plot results

% Plot Resistive Model results
figure('defaultAxesFontSize', 12, 'defaultLineLineWidth', 1, ...
       'Units', 'inches', 'Position', [1, 1, 9, 6]);

% Define layout for the subplots
tiledlayout(3, 1); % 3 rows, 1 column (upper subplot takes 2 rows, bottom takes 1 row)

% Upper subplot for Pressure
nexttile([2 1]); % Occupy 2 rows
```

```
hold on;

plot(R_tSol, R_PSol / 133.33, '-k', 'LineWidth', 1.5); % Pressure plot
plot(R_Pout(:,1), R_Pout(:,2) /133.33, '--r', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$P(t) \ (mmHg)$', 'Interpreter', 'latex');
%title('Pressure Solution', 'Interpreter', 'latex');
legend('MATLAB', 'Resistive BC', 'Interpreter', 'latex', 'location', 'southeast', 'Box', 'off');
ax = gca;
ax.YColor = 'k'; % Set left axis color to blue
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([70 120]);
hold off;

% Lower subplot for Flow Rate
nexttile; % Occupy 1 row
hold on;

plot(AorticFlowTime, AorticFlowRate * 1e6, '-k', 'LineWidth', 1.5); % Flow rate plot
plot(R_Qout(:,1), R_Qout(:,2) * 1e6, '--r', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$Q(t) \ (mL/s)$', 'Interpreter', 'latex');
%title('Flow Rate Solution', 'Interpreter', 'latex');
legend('Inlet','Resistive BC', 'Interpreter','latex', 'location', 'southeast', 'Box', 'off')
ax = gca;
ax.YColor = 'k'; % Set left axis color to red
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([-50 350]);
% Save the figure as an EPS file
print(gcf, 'R_Pressure', '-depsc', '-r300'); % '-depsc' for color EPS, '-r300' for resolution
hold off;


% Plot 2-Element Windkessel Model results
figure('defaultAxesFontSize', 12, 'defaultLineLineWidth', 1, ...
       'Units', 'inches', 'Position', [1, 1, 9, 6]);

% Define layout for the subplots
tiledlayout(3, 1); % 3 rows, 1 column (upper subplot takes 2 rows, bottom takes 1 row)

% Upper subplot for Pressure
nexttile([2 1]); % Occupy 2 rows
hold on;

plot(WK2_tSol, WK2_PSol / 133.33, '-k', 'LineWidth', 1.5); % Pressure plot
plot(WK2_Pout1(:,1), WK2_Pout1(:,2) /133.33, '--r', 'LineWidth', 1.5);
plot(WK2_Pout2(:,1), WK2_Pout2(:,2) /133.33, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$P(t) \ (mmHg)$', 'Interpreter', 'latex');
%title('Pressure Solution', 'Interpreter', 'latex');
legend('MATLAB', '$O(\Delta t)$ WK2 BC', '$O(\Delta t^2)$ WK2 BC', 'Interpreter', 'latex', 'location',
      'southeast', 'Box', 'off');
ax = gca;
ax.YColor = 'k'; % Set left axis color to blue
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([0 130]);
```

```
hold off;

% Lower subplot for Flow Rate
nexttile; % Occupy 1 row
hold on;

plot(AorticFlowTime, AorticFlowRate * 1e6, '-k', 'LineWidth', 1.5); % Flow rate plot
plot(WK2_Qout1(:,1), WK2_Qout1(:,2) * 1e6, '--r', 'LineWidth', 1.5);
plot(WK2_Qout2(:,1), WK2_Qout2(:,2) * 1e6, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$Q(t) \ (mL/s)$', 'Interpreter', 'latex');
%title('Flow Rate Solution', 'Interpreter', 'latex');
legend('Inlet','$O(\Delta t)$ WK2 BC', '$O(\Delta t^2)$ WK2 BC', 'Interpreter','latex', 'location', '
    southeast', 'Box', 'off')
ax = gca;
ax.YColor = 'k'; % Set left axis color to red
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([-50 350]);
% Save the figure as an EPS file
print(gcf, 'WK2_Pressure', '-depsc', '-r300'); % '-depsc' for color EPS, '-r300' for resolution
hold off;

% Plot 3-Element Windkessel Model results
figure('defaultAxesFontSize', 12, 'defaultLineLineWidth', 1, ...
       'Units', 'inches', 'Position', [1, 1, 9, 6]);

% Define layout for the subplots
tiledlayout(3, 1); % 3 rows, 1 column (upper subplot takes 2 rows, bottom takes 1 row)

% Upper subplot for Pressure
nexttile([2 1]); % Occupy 2 rows
hold on;

plot(WK3_tSol, WK3_PSol / 133.33, '-k', 'LineWidth', 1.5); % Pressure plot
plot(WK3_Pout1(:,1), WK3_Pout1(:,2) /133.33, '--r', 'LineWidth', 1.5);
plot(WK3_Pout2(:,1), WK3_Pout2(:,2) /133.33, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$P(t) \ (mmHg)$', 'Interpreter', 'latex');
%title('Pressure Solution', 'Interpreter', 'latex');
legend('MATLAB', '$O(\Delta t)$ WK3 BC', '$O(\Delta t^2)$ WK3 BC', 'Interpreter', 'latex', 'location',
      'southeast', 'Box', 'off');
ax = gca;
ax.YColor = 'k'; % Set left axis color to blue
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([0 140]);
hold off;

% Lower subplot for Flow Rate
nexttile; % Occupy 1 row
hold on;

plot(AorticFlowTime, AorticFlowRate * 1e6, '-k', 'LineWidth', 1.5); % Flow rate plot
plot(WK3_Qout1(:,1), WK3_Qout1(:,2) * 1e6, '--r', 'LineWidth', 1.5);
plot(WK3_Qout2(:,1), WK3_Qout2(:,2) * 1e6, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$Q(t) \ (mL/s)$', 'Interpreter', 'latex');
%title('Flow Rate Solution', 'Interpreter', 'latex');
legend('Inlet','$O(\Delta t)$ WK3 BC', '$O(\Delta t^2)$ WK3 BC', 'Interpreter','latex', 'location', '
    southeast', 'Box', 'off')
ax = gca;
```

```
ax.YColor = 'k'; % Set left axis color to red
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([-50 350]);
% Save the figure as an EPS file
print(gcf, 'WK3_Pressure', '-depsc', '-r300'); % '-depsc' for color EPS, '-r300' for resolution
hold off;


% Plot 4-Element (Series) Windkessel Model results
figure('defaultAxesFontSize', 12, 'defaultLineLineWidth', 1, ...
       'Units', 'inches', 'Position', [1, 1, 9, 6]);

% Define layout for the subplots
tiledlayout(3, 1); % 3 rows, 1 column (upper subplot takes 2 rows, bottom takes 1 row)

% Upper subplot for Pressure
nexttile([2 1]); % Occupy 2 rows
hold on;

plot(WK4s_tSol, WK4s_PSol / 133.33, '-k', 'LineWidth', 1.5); % Pressure plot
plot(WK4s_Pout1(:,1), WK4s_Pout1(:,2) /133.33, '--r', 'LineWidth', 1.5);
plot(WK4s_Pout2(:,1), WK4s_Pout2(:,2) /133.33, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$P(t) \ (mmHg)$', 'Interpreter', 'latex');
%title('Pressure Solution', 'Interpreter', 'latex');
legend('MATLAB', '$O(\Delta t)$ WK4Series BC', '$O(\Delta t^2)$ WK4Series BC', 'Interpreter', 'latex',
      'location', 'southeast', 'Box', 'off');
ax = gca;
ax.YColor = 'k'; % Set left axis color to blue
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([0 130]);
hold off;

% Lower subplot for Flow Rate
nexttile; % Occupy 1 row
hold on;

plot(AorticFlowTime, AorticFlowRate * 1e6, '-k', 'LineWidth', 1.5); % Flow rate plot
plot(WK4s_Qout1(:,1), WK4s_Qout1(:,2) * 1e6, '--r', 'LineWidth', 1.5);
plot(WK4s_Qout2(:,1), WK4s_Qout2(:,2) * 1e6, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$Q(t) \ (mL/s)$', 'Interpreter', 'latex');
%title('Flow Rate Solution', 'Interpreter', 'latex');
legend('Inlet','$O(\Delta t)$ WK4Series BC', '$O(\Delta t^2)$ WK4Series BC', 'Interpreter','latex', '
    location', 'southeast', 'Box', 'off')
ax = gca;
ax.YColor = 'k'; % Set left axis color to red
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([-50 350]);
% Save the figure as an EPS file
print(gcf, 'WK4s_Pressure', '-depsc', '-r300'); % '-depsc' for color EPS, '-r300' for resolution
hold off;


% Plot 4-Element (Parallel) Windkessel Model results
figure('defaultAxesFontSize', 12, 'defaultLineLineWidth', 1, ...
```

```
       'Units', 'inches', 'Position', [1, 1, 9, 6]);

% Define layout for the subplots
tiledlayout(3, 1); % 3 rows, 1 column (upper subplot takes 2 rows, bottom takes 1 row)

% Upper subplot for Pressure
nexttile([2 1]); % Occupy 2 rows
hold on;

plot(WK4p_tSol, WK4p_PSol(:,1) / 133.33, '-k', 'LineWidth', 1.5); % Pressure plot
plot(WK4p_Pout1(:,1), WK4p_Pout1(:,2) /133.33, '--r', 'LineWidth', 1.5);
plot(WK4p_Pout2(:,1), WK4p_Pout2(:,2) /133.33, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$P(t) \ (mmHg)$', 'Interpreter', 'latex');
%title('Pressure Solution', 'Interpreter', 'latex');
legend('MATLAB', '$O(\Delta t)$ WK4Parallel BC', '$O(\Delta t^2)$ WK4Parallel BC', 'Interpreter', '
    latex', 'location', 'southeast', 'Box', 'off');
ax = gca;
ax.YColor = 'k'; % Set left axis color to blue
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([0 120]);
hold off;

% Lower subplot for Flow Rate
nexttile; % Occupy 1 row
hold on;

plot(AorticFlowTime, AorticFlowRate * 1e6, '-k', 'LineWidth', 1.5); % Flow rate plot
plot(WK4p_Qout1(:,1), WK4p_Qout1(:,2) * 1e6, '--r', 'LineWidth', 1.5);
plot(WK4p_Qout2(:,1), WK4p_Qout2(:,2) * 1e6, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$Q(t) \ (mL/s)$', 'Interpreter', 'latex');
%title('Flow Rate Solution', 'Interpreter', 'latex');
legend('Inlet','$O(\Delta t)$ WK4Parallel BC', '$O(\Delta t^2)$ WK4Parallel BC', 'Interpreter','latex'
    , 'location', 'southeast', 'Box', 'off')
ax = gca;
ax.YColor = 'k'; % Set left axis color to red
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([-50 350]);
% Save the figure as an EPS file
print(gcf, 'WK4p_Pressure', '-depsc', '-r300'); % '-depsc' for color EPS, '-r300' for resolution
hold off;

% Plot Coronary LPN Model results
figure('defaultAxesFontSize', 12, 'defaultLineLineWidth', 1, ...
       'Units', 'inches', 'Position', [1, 1, 9, 6]);

% Define layout for the subplots
tiledlayout(3, 1); % 3 rows, 1 column (upper subplot takes 2 rows, bottom takes 1 row)

% Upper subplot for Pressure
nexttile([2 1]); % Occupy 2 rows
hold on;

% Plotting on the left y-axis
yyaxis left;
plot(Coronary_tSol, Coronary_PSol(:,1) / 133.33, '-k', 'LineWidth', 1.5); % Pressure plot
plot(Coronary_Pout1(:,1), Coronary_Pout1(:,2) / 133.33, '--r', 'LineWidth', 1.5);
plot(Coronary_Pout2(:,1), Coronary_Pout2(:,2) / 133.33, '--b', 'LineWidth', 1.5);
```

```
ylabel('$P(t) \ (mmHg)$', 'Interpreter', 'latex'); % Left y-axis label
ax = gca;
ax.YColor = 'k'; % Set left axis color
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels

% Plotting on the right y-axis
yyaxis right;
plot(PimTime, PimScaling * Pim / 133.33, ':', 'LineWidth', 1.5);
ylabel('$P_{im}(t) \ (mmHg)$', 'Interpreter', 'latex'); % Right y-axis label
ax = gca;
ax.YColor = [0.8500 0.3250 0.0980]; %'m'; % Set right axis color

xlabel('$t \ (s)$', 'Interpreter', 'latex'); % X-axis label
%title('Pressure Solution', 'Interpreter', 'latex');

legend('MATLAB', '$O(\Delta t)$ Coronary LPN BC', '$O(\Delta t^2)$ Coronary LPN BC', '$P_{im}(t)$',
    ...
      'Interpreter', 'latex', 'location', 'southeast', 'Box', 'off');

% Use LaTeX for tick labels and customize ticks
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;

% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]);
yyaxis left; ylim([0 140]); % Adjust left y-axis limits
yyaxis right; ylim([0 200]); %ylim([min(PimScaling * Pim / 133.33), max(PimScaling * Pim / 133.33)]);
    % Adjust right y-axis limits

hold off;

% Lower subplot for Flow Rate
nexttile; % Occupy 1 row
hold on;

plot(CoronaryFlowTime, CoronaryFlowRate * 1e6, '-k', 'LineWidth', 1.5); % Flow rate plot
plot(Coronary_Qout1(:,1), Coronary_Qout1(:,2) * 1e6, '--r', 'LineWidth', 1.5);
plot(Coronary_Qout2(:,1), Coronary_Qout2(:,2) * 1e6, '--b', 'LineWidth', 1.5);
xlabel('$t \ (s)$', 'Interpreter', 'latex');
ylabel('$Q(t) \ (mL/s)$', 'Interpreter', 'latex');
%title('Flow Rate Solution', 'Interpreter', 'latex');
legend('Inlet','$O(\Delta t)$ Coronary LPN BC', '$O(\Delta t^2)$ Coronary LPN BC', 'Interpreter','
    latex', 'location', 'southeast', 'Box', 'off')
ax = gca;
ax.YColor = 'k'; % Set left axis color to red
ax.TickLabelInterpreter = 'latex'; % Use LaTeX for tick labels
set(gca, 'XMinorTick', 'on', 'YMinorTick', 'on');
grid on; grid minor; box on;
% Adjust figure for better visibility
set(gca, 'FontSize', 12);
xlim([0 8]); ylim([0.15 0.7]);
% Save the figure as an EPS file
print(gcf, 'Coronary_Pressure', '-depsc', '-r300'); % '-depsc' for color EPS, '-r300' for resolution
hold off;

%% WK1 Function
function [tSol, PSol] = Resistive(Rd, Pd, FlowRate, FlowTime)

    % Q_fun represents the flow rate as a function of time
    Q_fun =@(t) interp1(FlowTime, FlowRate, t, 'linear', 'extrap');

    % Time span for the solution
    tSol = FlowTime;

    % Calculate P(t) using the WK1 equation
    PSol = Rd .* arrayfun(Q_fun, tSol) + Pd;
```

```
end


%% WK2 Function
function [tSol, PSol]= WK2(Rd, C, Pd, FlowRate, FlowTime)

    Q_fun =@(t) interp1(FlowTime, FlowRate, t, 'linear', 'extrap');

    % Define the differential equation
    % P'(t) + (1/Rd*C) * (P(t)-Pd) = Q(t)/C
    % Rearrange as: P'(t) = Q(t)/C - (1/Rd*C) * (P(t)-Pd)

    % Solve using MATLAB's ode45
    odefun = @(t, P) Q_fun(t) / C - (1 / (Rd * C)) * (P-Pd);

    % Initial condition
    P0 = 0; % Initial pressure, e.g., in Pa

    opts = odeset('RelTol',1e-6,'AbsTol',1e-8);

    tspan = [min(FlowTime) max(FlowTime)];

    % Solve ODE
    [tSol, PSol] = ode45(odefun, tspan, P0, opts);

end

%% WK3 Function
function [tSol, PSol] = WK3(Rp, Rd, C, Pd, FlowRate, FlowTime)

    % Interpolation of Q(t)
    Q_fun = @(t) interp1(FlowTime, FlowRate, t, 'linear', 'extrap');

    % Numerical differentiation of Q(t)
    dQdt = gradient(FlowRate, FlowTime);

    dQdt_fun = @(t) interp1(FlowTime, dQdt, t);

    % Define the differential equation
    % C * dP/dt = (1 + Rp/Rd) * Q(t) + Rp * C * dQ(t)/dt - (P - Pd) / Rd
    odefun = @(t, P) ((1 + Rp / Rd) * Q_fun(t) + Rp * C * dQdt_fun(t) - (P - Pd) / Rd) / C;

    % Initial condition
    P0 = 0; % Initial pressure

    opts = odeset('RelTol', 1e-6, 'AbsTol', 1e-8);

    tspan = [min(FlowTime) max(FlowTime)];

    % Solve ODE
    [tSol, PSol] = ode45(odefun, tspan, P0, opts);

end

%% WK4Series Function
function [tSol, PSol] = WK4Series(Rp, Rd, C, L, Pd, FlowRate, FlowTime)

    % Interpolation of Q(t)
    Q_fun = @(t) interp1(FlowTime, FlowRate, t, 'linear', 'extrap');

    % First derivative of Q(t)
    dQdt = gradient(FlowRate, FlowTime);
    dQdt_fun = @(t) interp1(FlowTime, dQdt, t, 'linear', 'extrap');

    % Second derivative of Q(t)
    d2Qdt2 = gradient(dQdt, FlowTime);
    d2Qdt2_fun = @(t) interp1(FlowTime, d2Qdt2, t, 'linear', 'extrap');
```

```
    % Define the differential equation
    % C * dP/dt = (1 + Rp/Rd) * Q(t) + (L/Rd + Rp * C) * dQ(t)/dt + C * L * d2Q(t)/dt^2 - (P - Pd) /
     Rd
    odefun = @(t, P) ...
        ((1 + Rp / Rd) * Q_fun(t) + ...
         (L / Rd + Rp * C) * dQdt_fun(t) + ...
         C * L * d2Qdt2_fun(t) - ...
         (P - Pd) / Rd) / C;

    % Initial condition
    P0 = 0; % Initial pressure

    opts = odeset('RelTol', 1e-6, 'AbsTol', 1e-8);

    tspan = [min(FlowTime) max(FlowTime)];

    % Solve ODE
    [tSol, PSol] = ode45(odefun, tspan, P0, opts);

end

%% WK4Parallel Function
function [tSol, PSol] = WK4Parallel(Rp, Rd, C, L, Pd, FlowRate, FlowTime)

    % Interpolation of Q(t)
    Q_fun = @(t) interp1(FlowTime, FlowRate, t, 'linear', 'extrap');

    % First derivative of Q(t)
    dQdt = gradient(FlowRate, FlowTime);
    dQdt_fun = @(t) interp1(FlowTime, dQdt, t, 'linear', 'extrap');

    % Second derivative of Q(t)
    d2Qdt2 = gradient(dQdt, FlowTime);
    d2Qdt2_fun = @(t) interp1(FlowTime, d2Qdt2, t, 'linear', 'extrap');

    % Define the differential equation (converted to a system of first-order ODEs)
    odefun = @(t, P) [
        P(2); % dP1/dt = P2 (pressure rate of change)
        P(3); % dP2/dt = P3 (second derivative of pressure)
        % Use the equation: (C*L/Rp)*d2P/dt^2 = Q(t) + L*((Rp + Rd)/(Rp*Rd))*dQ/dt + C*L*d2Q/dt^2 - (C
      + L/(Rp*Rd))*dP/dt - (P - Pd)/Rd
        (Q_fun(t) + L * ((Rp + Rd) / (Rp * Rd)) * dQdt_fun(t) + C * L * d2Qdt2_fun(t) - ...
         (C + L / (Rp * Rd)) * P(2) - (P(1) - Pd) / Rd) * (Rp / (C * L));
    ];

    % Initial conditions for the system
    P0 = 0;       % Initial pressure P(t)
    dPdt0 = 0;    % Initial first derivative of pressure
    d2Pdt20 = 0;  % Initial second derivative of pressure

    % Set options for ODE solver
    opts = odeset('RelTol', 1e-6, 'AbsTol', 1e-8);

    tspan = [min(FlowTime) max(FlowTime)];

    % Solve the system of ODEs using ode45
    [tSol, PSol] = ode15s(odefun, tspan, [P0, dPdt0, d2Pdt20], opts);

end

%% Coronary LPN Function
function [tSol, PSol] = CoronaryLPN(Ra, Ram, Rv, Rvm, Ca, Cim, Pv, PimScaling, Pim, PimTime, FlowRate,
      FlowTime)

    % Interpolation of Q(t)
    Q_fun = @(t) interp1(FlowTime, FlowRate, t, 'linear', 'extrap');

    % First derivative of Q(t)
```

```
    dQdt = gradient(FlowRate, FlowTime);
    dQdt_fun = @(t) interp1(FlowTime, dQdt, t, 'linear', 'extrap');

    % Interpolation of Pim(t)
    ScaledPim = PimScaling * Pim;

    Pim_fun = @(t) interp1(PimTime, ScaledPim, t, 'linear', 'extrap');

    % First derivative of Pim(t)
    dPimdt = gradient(ScaledPim, PimTime);
    dPimdt_fun = @(t) interp1(PimTime, dPimdt, t, 'linear', 'extrap');

    % Define the system of 2 ODEs
    odefun = @(t, P) [

        (1 / Ca) * (Q_fun(t) * (1 + Ra / Ram) + Ca * Ra * dQdt_fun(t) - (P(1) - P(2)) / Ram);

        (1 / Cim) * (Q_fun(t) - Ca * gradient(P(1),t) - (P(2) - Pv) / (Rv + Rvm) + Cim * dPimdt_fun(t)
     )
    ];

    % Initial conditions
    P0 = [0, 0];

    % Set options for ODE solver
    opts = odeset('RelTol', 1e-6, 'AbsTol', 1e-8);

    tspan = [min(FlowTime) max(FlowTime)];

    % Solve the system of ODEs using ode15s
    [tSol, PSol] = ode15s(odefun, tspan, P0, opts);

end
```

Index