Distribution Agreement

In presenting this thesis as a partial fulfillment of the requirements for a degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis in whole or in part in all forms of media, now or hereafter now, including display on the World Wide Web. I understand that I may select some access restrictions as part of the online submission of this thesis. I retain all ownership rights to the copyright of the thesis. I also retain the right to use in future works (such as articles or books) all or part of this thesis.

Wenda Zheng April 6, 2018

Domain Decomposition in Computational Fluid Dynamics for Pipe-like Domains

By

Wenda Zheng

Alessandro Veneziani

Advisor

Department of Mathematics and Computer Science

Alessandro Veneziani

Advisor

Hao Huang

Committee Member

Marina Piccinelli

Committee Member

2018

Domain Decomposition in Computational Fluid Dynamics for Pipe-like Domains

By

Wenda Zheng

Alessandro Veneziani

Advisor

An Abstract of

a thesis submitted to the Faculty of Emory College of Arts and Sciences

of Emory University in partial fulfillment

of the requirements of the degree of

Bachelors of Science with Honors

Department of Mathematics and Computer Science

2018

Abstract

Domain Decomposition in Computational Fluid Dynamics for Pipe-like Domains

By Wenda Zheng

Nowadays, as the result of progress of numerical methods for solving complex problems, simulations of patient-specific cardiovascular districts are possible and used in medicine. Nevertheless, when covering large portion of the circulation, simulations on regular computers can be not affordable. In this thesis, starting from the evidence that the circulatory network is made of pipes and junctions, we test the efficacy of "domain decomposition" techniques on our constructed 2D models. Using FreeFem++ as the simulation tool, we evaluates the Domain Decomposition techniques on 2D problems and try to find the possibilities to extend our work into 3D realistic cases.

Domain Decomposition in Computational Fluid Dynamics for Pipe-like Domains

By

Wenda Zheng

Alessandro Veneziani

Advisor

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
the degree of Bachelors of Science with Honors

Department of Mathematics and Computer Science

2018

Contents

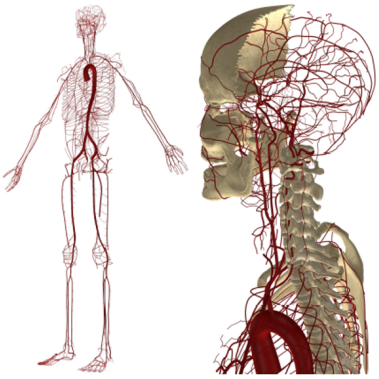
Chapter 1

Introduction

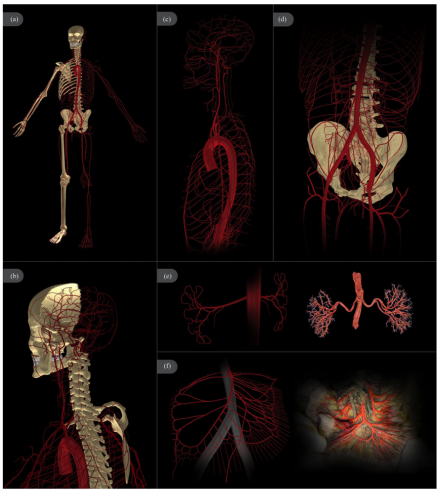
In the last 20 years, the contemporary progress of numerical methods for solving complex problems, medical imaging devices and computational parallel architectures made it possible the accurate simulation of difficult problems relevant for cardiovascular sciences. Nowadays, simulations of patient-specific cardiovascular districts are possible and used in medicine.

Nevertheless, some challenges need to be addressed for a penetration of mathematical models in the practice of cardiovascular clinics. In particular, when covering large portion of the circulation, simulations on regular computers can be not affordable. As in Figure 1.1, the circulatory systems are very complex. When viewing them as a single whole domain, it requires a lot for a regular computer to run simulations on it.

In this work, starting from the evidence that the circulatory network is made of pipes and junctions, we test the efficacy of "domain decomposition" techniques, where nontrivial networks are regarded as the result of elementary components, cylindrical pipes



(a) Example of Cardiovascular System [1]

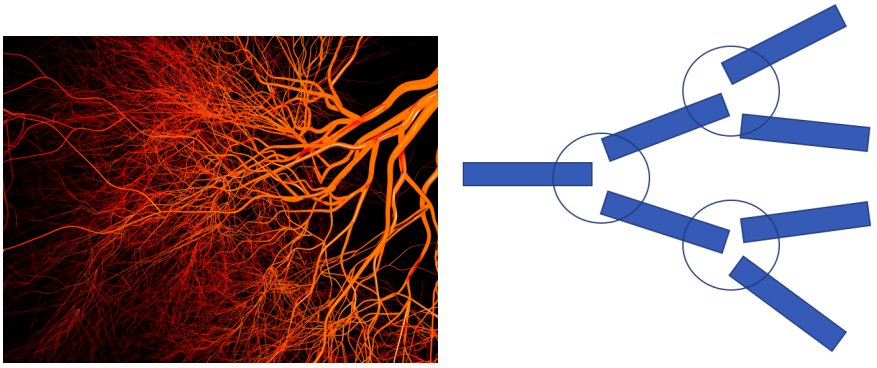


(b) Unique vascular anatomy of the ADAN model comprising: 1598 named arteries, 544 perforator vessels, blood supply to 28 specific organs, and to 116 vascular territories [2].

Figure 1.1: Large Circulation Examples

and the corresponding bifurcations. With domain decomposition techniques (see e.g. [QuarteroniValli]) it is possible to split a complex domain into a sequence of simple elementary domains, the solution being coupled by appropriate iterative procedures. As the numerical solution of 3D problems is beyond the breath of a dissertation like the present one, we will limit to 2D problems as a simplification of real problems, yet realistic enough to identify possible numerical troubles.

Figure 1.2 refers to how a circulatory networks are simplified and viewed as pipes and junctions.



(a) Complex Vessels [3] (b) Pipe-like Domains

Figure 1.2: Converting Circulatory Networks to Pipes and Junctions

1.1 Motivation

In domain decomposition, one solves a problem on a complex domain by splitting it on simpler domains and working iteratively to identify the solution on the original region. In this way, the global cost of the simulation is broken up over the subdomains, along the needed iterations. In this work we plan to use this method in 2D nontrivial domains on the incompressible Navier-Stokes equations, one of the most important problems in applied mathematics and computational fluid mechanics. We use this problem in 2D as a preliminary work for more realistic cases in 3D.

1.2 Contribution

After testing our method on a simple diffusion-reaction problem to validate the accuracy of the method, we present the methodology for the incompressible Navier-Stokes equations, in different geometries with an increasing level of complexity. The results

of simulations on our constructed 2D domains are reasonable and therefore it is possible to extend our problem into 3D realistic cases later. The reliability of our domain decomposition procedure is discussed in details.

1.3 Outline

In Chapter 2, we will discuss our mathematical problems needed for simulation. Then in Chapter 3, we will discuss domain decomposition techniques and simulation procedures. After that we will introduce finite element method and FreeFem++. In Chapter 4, we will present our numerical results from simulations. Chapter 5 includes a conclusion and future directions.

Chapter 2

The Mathematical Problems

In this chapter, we first introduce some basic knowledge of **Hilbert Spaces**. Then we will briefly discuss two PDE problems: one is a **2D Laplace equation**, and the other is the famous **Navier-Stoke Equation**. For the former one, we use it as a test for the accuracy of domain decomposition techniques, and for the latter one, we use it for blood simulations on our constructed models.

2.1 Intro to Hilbert Spaces

A Hilbert space is defined as a complete space with scalar product. In numerical PDEs, we have also defined space of functions $L^2(0,1)$ and $H^1(0,1)$.

Given an arbitrary function f, define:

$f \in L^2(0,1)$ if $|\int_1^0 f^2 dx| < \infty$, which means the function f is bounded above.

$f \in H^1(0,1)$ if f satisfies $|\int_1^0 f^2 dx| < \infty$ as well as $|\int_1^0 (\frac{df}{dx})^2 dx| < \infty$, which means the function f itself and its first order derivative are both bounded above.

Therefore, in general, a function f is in $H^k(0,1)$ if f, $\frac{df}{dx}$, $\frac{d^2f}{dx^2}...\frac{d^kf}{dx^k} \in L^2(0,1)$. Moreover, we can see that $H^k \subset H^{k-1} \subset ... \subset H^1 \subset L^2$, and L^2 is a Hilbert Space.

In numerical PDEs, our functions are usually defined in Hilbert Spaces.

2.2 A simple 2D diffusion-reaction problem

Initially, in order to get familiar with domain decomposition method, we first try to test our model using a 2D elliptic equation in a well defined domain Ω. Our problem is the following:

$$\begin{cases} -\Delta u + 2u = 0 \ \ in \ \ \Omega \\ u = g = e^{x+y} \ \ on \ \ \partial\Omega \end{cases} \tag{2.1}$$

The weak formulation of (2.1) reads: find $\boldsymbol{u} \in H^1(\Omega)$ s.t.

$$\int_\Omega \nabla u \nabla v + 2\int_\Omega uv = 0 \tag{2.2}$$

for all $\boldsymbol{v} \in H^1(\Omega)$. We write down the weak form of our problem because we need to use FreeFem++ (will be discussed in Chapter3) to code the problem later.

In order to achieve the the exact solution of the problem, we assume the solution has the form $u = X(x)Y(y)$ using separation of variables. Moreover, the boundary $g = e^{x+y}$

can also be written as the separated form $g = B_x(x)B_y(y)$. Thus, we can rewrite the initial problem as

$$-X''Y - XY'' + 2XY = 0.$$

Then assuming $X \neq 0$ and $Y \neq 0$, we rearrange the above equation and get

$$\frac{X''}{X} = 2 - \frac{Y''}{Y}$$

Since X and Y are independent variables, both sides should be constants. Hence, we can write

$$\frac{X''}{X} = K$$

for some real constant $K > 0$. Then X can be written as the following:

$$X = C_1 e^{\sqrt{k}x} + C_2 e^{-\sqrt{k}x}.$$

By prescribing the boundary conditions, we obtain $X = e^x$. Similarly, we proceed the same procedure for Y, and we get $Y = e^y$. Therefore, the final exact solution reads:

$$u_{exact} = e^{x+y}$$

2.3 Navier-Stokes Equations for Incompressible Fluids

Navier-Stokes equation is used to describe the motion of viscous liquid in physics world. Since blood is almost incompressible, we will use incompressible Navier-Stokes Equation to do simulations. The equation is in the form:

$$\begin{cases} -\nu\Delta\boldsymbol{u} + (\boldsymbol{u}\cdot\nabla)\,\boldsymbol{u} + \nabla p = \boldsymbol{f} & in \quad \Omega \\ \nabla\cdot\boldsymbol{u} = 0 & in \quad \Omega \\ u = g & on \quad \Gamma \end{cases} \tag{2.3}$$

The system (2.3) is referred to as *incompressible Navier-Stokes equations*. An introduction to these equations can be found in [4]. These balance equations arise from applying Newton's second law to fluid motion, together with the assumption that the stress in the fluid is the sum of a diffusing viscous term and a pressure term. In this equation, $\boldsymbol{u}$ stands for the fluid's velocity; p represents as the pressure divided by density, but it is still called as "pressure" for simplification; and ν is the fluid's viscosity.

The weak formulation of (2.3) reads: find $\boldsymbol{u} \in H^1(\Omega)$ and $p \in L^2(\Omega)$ s.t.

$$\int_\Omega \nu\nabla\boldsymbol{u} : \nabla\boldsymbol{v} + \int_\Omega (\boldsymbol{u}\cdot\nabla)\,\boldsymbol{u}\cdot\boldsymbol{v} - \int_\Omega \nabla\cdot\boldsymbol{v}p + \int_\Omega \nabla\cdot\boldsymbol{u}q - \int_\Omega \boldsymbol{f}\boldsymbol{v} \tag{2.4}$$

for all $\boldsymbol{v} \in H^1(\Omega)$ and $q \in L^2(\Omega)$.

In the next chapter, we will present simulations on pipe-like domains using domain decomposition with Navier-Stokes equation.

Chapter 3

Domain Decomposition Techniques and Finite Element Method

In this chapter, we will first introduce **domain decomposition techniques** together with simulations of our constructed models. Next, we will present a method called **finite element** to solve PDEs numerically. Moreover, in the finite element part, we will also introduce **FreeFem++**, which is a powerful tool for PDE simulations. In the next Chapter, we will present the numerical results of our simulations.

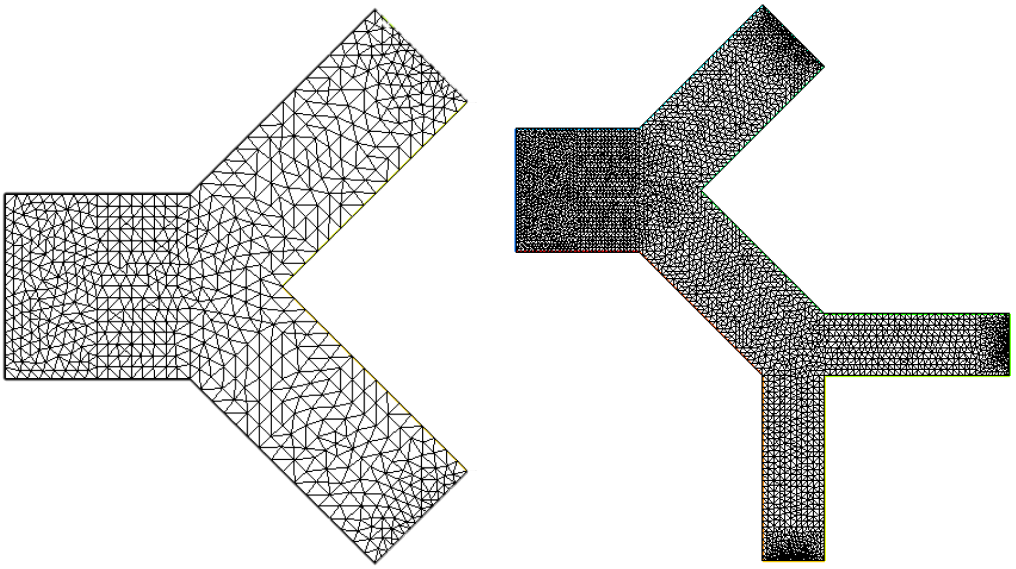
3.1 Domain Decomposition Techniques

Domain Decomposition technique is a well-established method designed for solving the entire solution in a whole domain with boundary conditions from the solution of its subdomains. More specifically, it is like a "divide and conquer" technique which splits one

domain into several small subdomains. We solve each part iteratively and then coordinate those adjacent parts until they converge.

Domain decomposition can be ascribed into two types: one is *The Overlapping Method* and one is *The Non-overlapping Method*. Those two methods will be clarified together with examples of the simulations on our constructed models.

Now, let us introduce our 2D models. In Figure 3.1, one is a single mesh with one bifurcation, and the other is a more complicated model with two bifurcations. We enforce the fluid flowing into the left part and flowing out of the right parts. In the rest of this chapter, we will discuss domain decomposition on those two constructed models and how the simulations work.



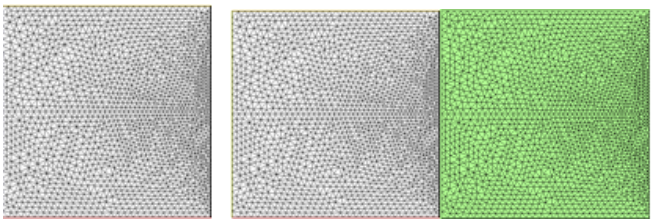
(a) Single Vessel Model (b) Modified Vessel Model

Figure 3.1: Examples of Two Constructed Model

3.1.1 Test Cases Using 2D Elliptic Equation

Before further introduction of domain decomposition techniques, we first want to simulate a test case for the 2D Elliptic Problem which is discussed in the previous chapter.

Let us consider a a unit square domain $\Omega = (0,1) \times (0,1)$ and a rectangular domain $\Omega = (0,2) \times (0,1)$ as listed in Figure 3.2.



(a) Single-Square Domain Ω

(b) Rectangular Domain with Ω_1 on the left and Ω_2 on the right

Figure 3.2: Simple 2D models for Testing

For the square mesh in Figure 3.2, we use FreeFem++ to code the problem and simulate the solution. We finally derive our analytic solution as $u_{numerical}$. Then we compare it with the exact solutions u_{exact} derived in Chapter 2 by taking the 2-norm $\|u_{exact} - u_{numerical}\|_2$.

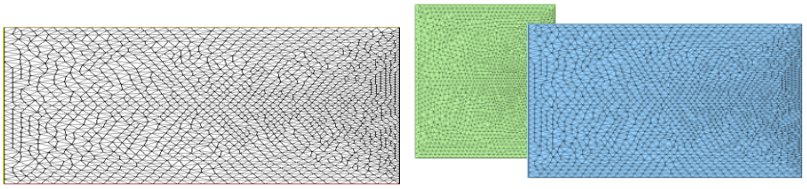
Then we add another domain Ω_2 with the same size into Ω_1 without overlapping (please see section 3.1.3 for further details), and enforce the Dirichlet-Neumann boundary conditions

As expected, after the simulation using FreeFem++, the error of the two-domain problem tends to zero as well. Thus, the results of our test cases are good enough and we may say that the Domain Decomposition technique is accurate and it is a good choice for our computational fluid dynamic problem.

In the rest of the chapter, all simulations that are discussed use Navier-Stokes equation defined in the previous chapter.

3.1.2 The Overlapping Method

One of the domain decomposition techniques is called "The Overlapping Method". In this method, we can divide the whole domain into several subdomains with each adjacent subdomains overlapping more than their interfaces.



(a) A Rectangular Domain (b) Decomposed domain

Figure 3.3: Example of Domain Decomposition With Overlapping Method

Figure 3.3 shows how a regular rectangular domain can be decomposed into two overlapping subdomains.

For overlapping methods, our project uses Schwarz Algorithm. To provide more introduction, let us illustrate a simple 1D example [5]: for $f(x) \in L^2(0,1)$, find $u(x)$ s.t.

$$\begin{aligned} -\tfrac{d^2u}{dx^2} &= f \qquad x \in (0,1) \\ u(0) &= u(1) = 0 \end{aligned} \tag{3.1}$$

As in Figure 3.4, we enforce two points x_l and x_r in the interval (0, 1) such that $0 < x_l < x_r < 1$ and the corresponding domains are [0,x_r] and [x_l,1].

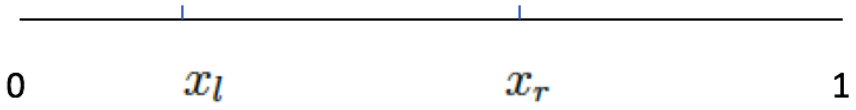


Figure 3.4: 1D Domain Discussed in the Example

Here, Figure 3.5 represents the algorithm which solves the problem in our example above.

Algorithm 2

1. Let $\lambda^{(0)} \in \mathbb{R}$ and $\mu^{(0)} \in \mathbb{R}$ be two arbitrary numbers.
2. Loop: up to convergence, for $i = 1, 2, \ldots$ solve:

a.
$$-\frac{d^2 u_1^{(i+1)}}{dx^2} = f \; x \in (0, x_r) \qquad (3.71)$$
$$u_1^{(i+1)}(0) = 0, \quad u_1^{(i+1)}(x_r) = \lambda^{(i)}.$$

b.
$$-\frac{d^2 u_2^{(i+1)}}{dx^2} = f \quad x \in (0, x_r) \qquad (3.72)$$
$$u_2^{(i+1)}(x_l) = \mu^{(i)}, \; u_2(1) = 0.$$

c.
$$\lambda^{(i+1)} = u_2^{(i+1)}(x_r), \quad \mu^{(i+1)} = u_1^{(i+1)}(x_l). \qquad (3.73)$$

3. End of the loop.

Figure 3.5: Addictive Schwarz Algorithm for Domain Decomposition with Overlapping [5]

However, the above algorithm is designed for 1D problem, and it just provides an inspiration of how domain decomposition works. Now that the goal of our project is to do simulations on 2D pipe-like domains, we want to extend the algorithm to 2D. Since the Schwarz solver can solve for rectangular domains, let us consider an L-shaped domain Ω as illustrated in Figure 3.6. We decompose it into two domains Ω_1 and Ω_2, both has

the same interface, or overlapping part. In order to achieve the convergence, we have the following pseudocode for this 2D problem :

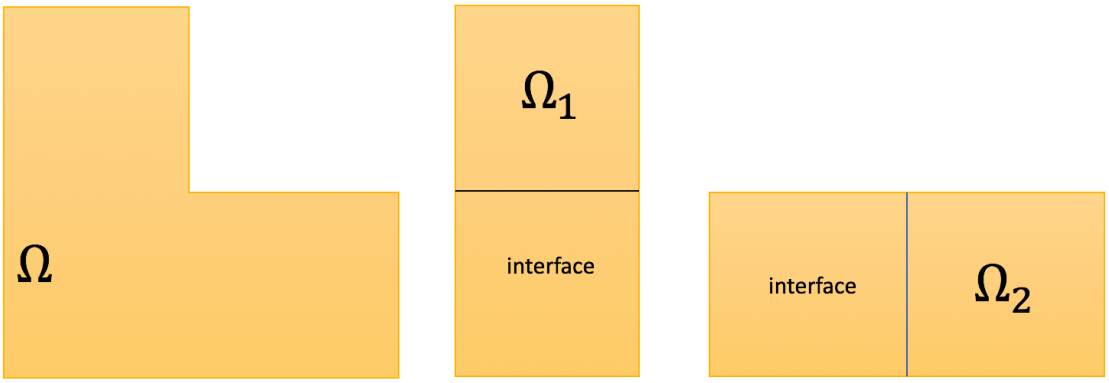


Figure 3.6: L-shape Domain and Its Decomposition with Overlapping

We define B.C. as the simplification of boundary condition and Sol as the simplification of solution.

While error $>$ threshold k $<$ number of maximum iterations

$P^k(\Omega_1) = 0$ with B.C.$(Sol^k(\Omega_2))$

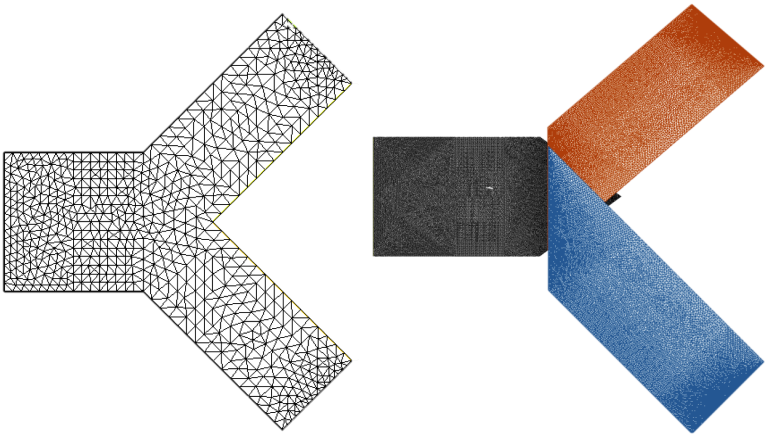
$P^k(\Omega_2) = 0$ with B.C.$(Sol^k(\Omega_1))$

$$\text{error} = (\int_{\Omega_1} (Sol^k(\Omega_1) - Sol^{k-1}(\Omega_1))^2 + \int_{\Omega_2} (Sol^k(\Omega_2) - Sol^{k-1}(\Omega_2))^2)^{\frac{1}{2}}$$

END While

Exploiting the algorithm above, we decompose our single model with overlapping (see Figure 3.7).

As a whole domain, we first simulate Navier-Stokes Equation on it. After using FreeFem++, the numerical solution is accurate enough. Thereafter, we try the same problem on our decomposed model with overlapping domains and we still get a reasonable result.



(a) Example of a vessel model

(b) Decomposed Model With Overlapping Method

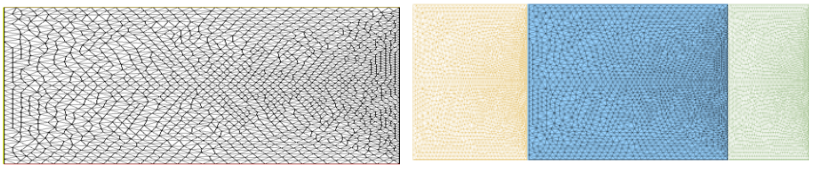
Figure 3.7: Example of Domain Decomposition for a blood vessel with One Bifurcation

However, our decomposed domains with overlapping are not rectangular. Furthermore, the way we decompose our model are not simplified enough. In order to reduce the cost and information needed, we are willing to try domain decomposition with non-overlapping method.

3.1.3 The Non-overlapping Method

Another domain decomposition techniques is called "Non-Overlapping Method". In this method, we divide the whole domain into several subdomains with each adjacent subdomains intersecting only on their interfaces. Figure 3.8 shows how a regular rectangular domain can be decomposed into three non-overlapping subdomains.

Now, let us still focus on the problem in our previous example. The only change is that we define $x_\Gamma (=x_l=x_r)$ be a point in (0,1) [citation] as showed in Figure 3.9 . Then our non-overlapping algorithm reads in Figure 3.10 below.



(a) A Rectangular Domain (b) Decomposed domain

Figure 3.8: Example of Domain Decomposition With Non-Overlapping Method

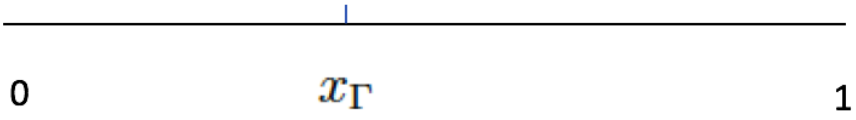


Figure 3.9: 1D Domain Discussed in the Example

Similarly, the algorithm is an example of 1D problem. We still need to extend our case into 2D. Let us consider the following example with a rectangular domain.

In Figure 3.11, the two non-overlapping domains Ω_1 and Ω_2 have interactions to each other. Suppose for the whole domain Ω, we have Laplace equation $\Delta u = f$, and for subdomains Ω_1 we have $\Delta u_1 = f$ and for Ω_2 we have $\Delta u_2 = f$. The pseudocode is the following, which is also known as Dirichlet-Neumann method [citation]:

While error $>$ threshold k $<$ number of maximum iterations

$\Delta u_1^k = f$ with Dirichlet B.C. $u_1^k = u_2^k$

$\Delta u_2^k = f$ with Neumann B.C. $\nabla u_1^k \cdot n = \nabla u_2^k \cdot n$.

$$\text{error} = (\int_{\Omega_1} (u_1^k - u_1^{k-1})^2 + \int_{\Omega_2} (u_2^k - u_2^{k-1})^2)^{\frac{1}{2}}$$

END While

With knowledge of non-overlapping method in domain decomposition, we first decompose our model into four parts: one parent, two children and one triangle connecting

Algorithm 4

1. Let $\lambda^{(0)} \in \mathbb{R}$ be an arbitrary number and ϑ a relaxation parameter.
2. Loop: up to convergence, for $i = 1, 2, \ldots$ solve

   a.
   $$-\frac{d^2 u_1^{(i+1)}}{dx^2} = f \; x \in (0, x_\Gamma) \tag{3.82}$$
   $$u_1^{(i+1)}(0) = 0, \quad u_1^{(i+1)}(x_\Gamma) = \lambda^{(i)}.$$

   b.
   $$-\frac{d^2 u_2^{(i+1)}}{dx^2} = f \qquad x \in (x_\Gamma, 1) \tag{3.83}$$
   $$\frac{du_2^{(i+1)}}{dx}(x_\Gamma) = \frac{du_1^{(i+1)}}{dx}(x_\Gamma), \; u_2^{(i+1)}(1) = 0.$$

   c.
   $$\lambda^{(i+1)} = \vartheta u_2^{(i+1)}(x_\Gamma) + (1 - \vartheta)\lambda^{(i)}. \tag{3.84}$$

3. End of the loop.

Figure 3.10: Algorithm for Domain Decomposition with Non-overlapping [5]

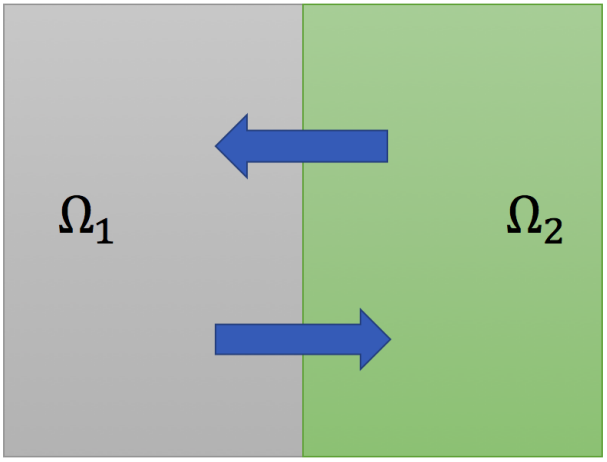
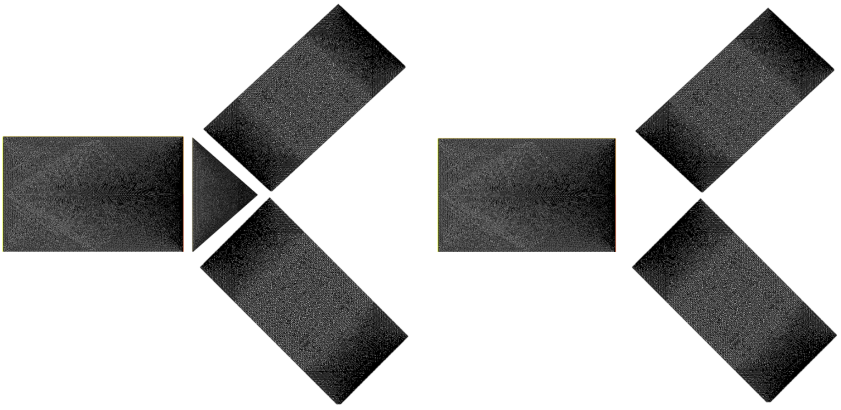


Figure 3.11: Non-overlapping Example

other three subdomains (refer to Figure 3.12). Then similar to previous procedures, we simulate Navier-Stokes equation on those new domains. The result is as good as that of on overlapping domains.



(a) Domain Decomposition With Triangle

(b) Domain Decomposition Without Triangle

Figure 3.12: Single Blood Vessel Model Without Overlapping

Not satisfying our current model, we try to modify our model more. This time, we manage to pseudo gate the triangle in the middle for two reasons (please see the right plot of Figure 3.12): firstly, the domain is not in rectangular shape; secondly, we want to simplify our model more. We apply normal Neumann conditions from two child domains to the left one. Moreover, the volume of the fluid Q from the left domain is divided because of the existence of bifurcations. As those two bifurcated domains have the same size, we assume that the volume of flow $\mathbf{Q}$ will be divided evenly into two parts, each of which has the flow volume $\frac{Q}{2}$. After introducing this new boundary condition, we run simulations on this non-overlapping domain. With only several iterations, the calculated error converges to a reasonable value, which means the way we decompose our model is possible.

Figure 3.13 refers to the plot of errors in each iteration for our decomposed model without triangle domain. We can infer that the solution will finally converge.

Now that the simulations on our single domain with bifurcation are very successful, we would like to modify a little bit to our original model and see whether our FreeFem++

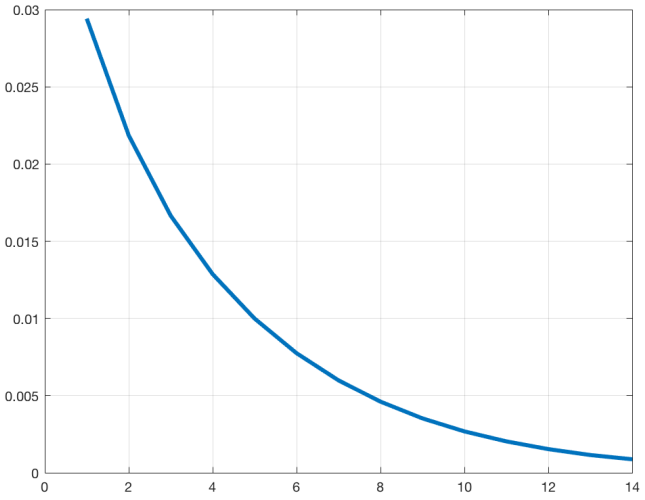
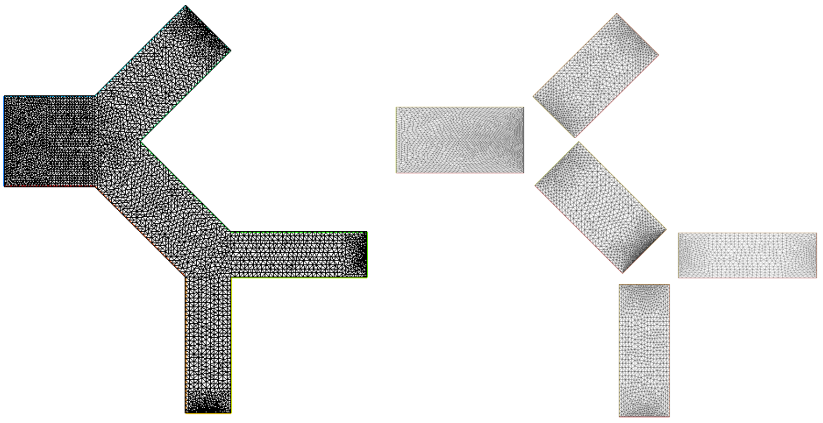


Figure 3.13: Error for Domain Decomposition Without Triangle

are still powerful on the new one. Figure 3.14 represents our new model with its domain decomposition. The modified model has one more child bifurcation after its first bifurcation. In our simulation, we still enforce the blood flowing into the left side and flowing out of the right side. Similar to the above procedure, we divide the volume of main flow **Q** to some ways according to the ratio of the size of the domains. Here in our model, we can also prescribe different values of pressure in order to control the way blood flows.



(a) Model without Domain Decomposition

(b) Model with Domain Decomposition

Figure 3.14: New Blood Vessel Model With Child Bifurcation

The simulations in our new model still work well and the final solutions converges to some points that is tolerable even though there are tiny oscillations (see Figure 3.15).

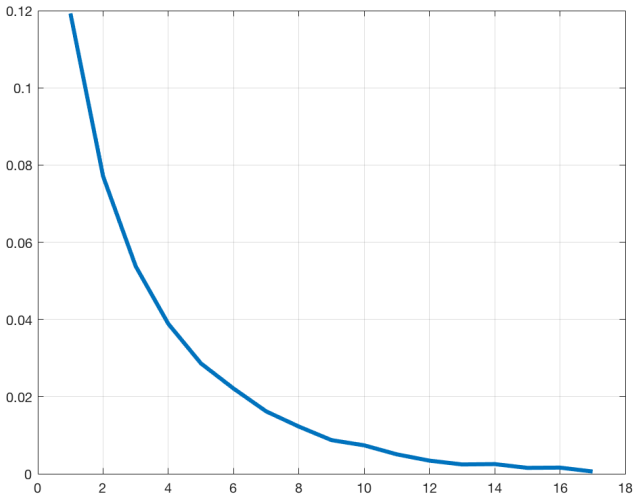


Figure 3.15: Error for Modified Model with Domain Decomposition

3.2 Introduction to Finite Elements

In recent years, the progress of numerical method makes it possible for accurately solving PDEs numerically. The finite elements method is one of the numerical methods capable of solving many industrial problems including blood flow in computational fluid dynamics.

3.2.1 Finite Elements Using Galerkin Method

There are several cases in Finite Element Method, one particular cases of which is called Galerkin Finite Elements Method. Before introducing Galerkin Method, we first recall Lagrange Polynomial which interpolates our nodes piece-wise linearly.

For interpolation, we have several points. i.e. $(1,1),(2,3),(3,7)$.... In general, we denotes those points as (x_i, f_i). We define the Lagrange Polynomial as the following:

$$P(x) = f_1\phi_1(x_2) + f_2\phi_2(x_2) + ... + f_n\phi_n(x_n) = \sum_{j=0}^{n} f_j\phi_j(x_j)$$

where the function $\phi_j(x)$ satisfies

$$\phi_j(x) = \begin{cases} 0 & i = j \\ 1 & i \neq j \end{cases}$$

Hence, for a given Lagrange Polynomial, we have $P(x_1) = f_1$, $P(x_2) = f_2$ ... $P(x_n) = f_n$.

Now, let as consider a generic elliptic problem in a domain Ω [4]:

Find u $\in$ V: a(u,v) = F(v) for any v$\in$ V, where V is an appropriate Hilbert space.

Then let $V_h \subset V$ which depends on a positive parameter h. Now we approximate our original problem as

$$Find\, u_h \in V_h : \;\; a(u_h, v_h) = F(v_h) \;\; \forall v \in V.$$

The above equation is called Galerkin problem and we denote $\{\phi_j, j = 1, 2, 3...N_h\}$ as a basis of V_h. We rewrite the problem as

$$a(u_h, \phi_j) = F(\phi_j), \;\; i = 1, 2, ..., N_h$$

And we can also derive that

$$u_h(x) = \sum_{j=1}^{N_h} u_j \phi_j(x)$$

Then we can write the Galerkin problem as

$$\sum_{j=1}^{N_h} u_j a(\phi_j, \phi_i) = F(\phi_i).$$

Denoting the matrix $a_{ij} = a(\phi_j, \phi_i)$ by A, we have the following linear system:

$$A\mathbf{u} = \mathbf{f}$$

The above procedure is called Galerkin Method. It is often used for solving numerical PDEs.

3.2.2 FreeFem++

FreeFem++ is a very powerful tool for solving numerical partial differential equation. It is written in C++ and the way it solves the problem is using finite element method. The code below is an example of FreeFem++ code using finite elements method to evaluate the solutions.

listings xcolor

```
1 border down1(t=0,1){x=t;y=-0.5;label=1;};
2 border down2(t=1,2){x=t;y=-t+0.5;label=2;};
3 border down3(t=2,2.5){x=t;y=t-3.5;label=3;};
4 border down4(t=2.5,1.5){x=t;y=-t+1.5;label=4;};
5 border up5(t=1.5,2.5){x=t;y=t-1.5;label=5;};
```

```
6  border up6(t=2.5,2){x=t;y=-t+3.5;label=6;};
7  border up7(t=2,1){x=t;y=t-0.5;label=7;};
8  border up8(t=1,0){x=t;y=0.5;label=8;};
9  border left(t=0.5,-0.5){x=0;y=t;label=9;};
10
11 int nm=16;
12 mesh Th1=buildmesh(down1(nm)+down2(nm)+down3(nm)+down4(nm)
13 +up5(nm)+up6(nm)+up7(nm)+up8(nm)+left(nm)); //1 first domain
14 plot(Th1,wait=1);
15 fespace X1h(Th1,P1);
16 X1h u1h,v1h,u1hold;
17
18 bool convergence;
19 int k=0,kmax=100;
20 real thr=0.001,error=1000.0,gamma=0.75,e1,e2;
21
22 u1h=0.0;
23 u1hold=u1h;
24 func ue=exp(x+y);
25 func dxue=exp(x+y);
26 func dyue=exp(x+y);
27
28 problem Problem1(u1h,v1h) =
29     int2d(Th1)(dx(u1h)*dx(v1h) + dy(u1h)*dy(v1h))
30   + int2d(Th1)(2*u1h*v1h)
31   + on(1,2,3,4,5,6,7,8,9,u1h=ue); //2
32
33
34 convergence=false;
35
36 while(convergence==false && k<=kmax)
37 {
38     u1hold=u1h;
39     Problem1;
40     plot(u1h,wait=1,fill=1,value=1);
41     error = sqrt(int1d(Th1)((u1h-ue)^2));
```

```
42      cout << "Error at iteration " << k << " is " << error << endl;
43      if (error<=thr) convergence=true;
44      cout << "Convergence is " << convergence << endl;
45      k++;
46      u1h = gamma*u1h + (1-gamma)*u1hold;
47  }
48  plot(u1h,wait=1,fill=1,value=1);
```

Here we would like to interpret a little more about my example code above. This code is designed for simulating 2D elliptic equations which is discussed in Chapter 2.

First, through line 1 to line 9, we define our 2D model. Then through line 11 to 16, we are building an object called **Th** of **Mesh** corresponding to our 2D model with mesh size 16. At the same time, we plot our mesh and define our finite element space as P1, which means linear interpolation.

We then declare our unknown numerical solution as **u1h** and our generic test function as **v1h**. We also define our exact solution as **ue**, where $ue = e^{x+y}$ as we calculated in Chapter 2.

Starting at Line 28, we define our problem which corresponds to the weak formulation discussed in Chapter 2 expect for notations of variables:

$$a(u,v) = \int_{\Omega} \nabla u \nabla \boldsymbol{v} + 2 \int_{\Omega} u \boldsymbol{v} \tag{3.2}$$

for all $\boldsymbol{v} \in H^1(\Omega)$.

In Line 31, we prescribe Dirichlet boundary conditions as Function ue.

Finally, through Line 36 to Line 47, we use domain decomposition technique to compute our numerical solution **u1h** after several iterations. And then we plot the resulting picture as Figure 3.16.

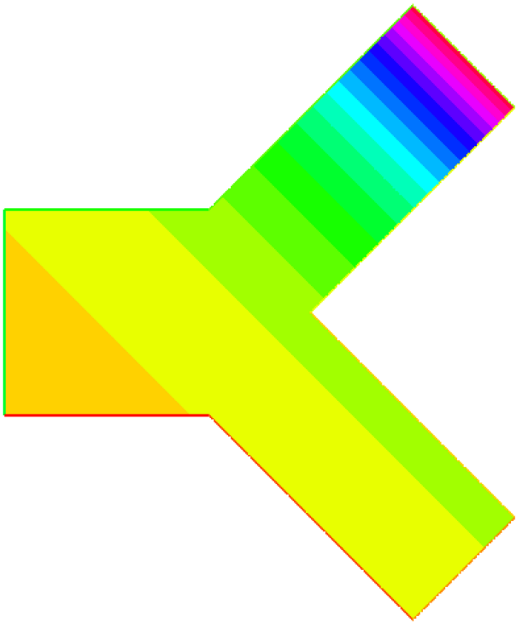


Figure 3.16: Final Result of The Example Code

Chapter 4

Numerical Results

In this chapter, we only present results of simulations with Navier-Stokes equation. Because simulating 2D elliptic equation is trivial and it is just a test case for the accuracy of domain decomposition techniques.

Since we cannot solve Navier-Stokes equation exactly, what we do in each iteration is to compute the norm of its current solution with its previous solution, i.e. error $=\|u_h - u_{hold}\|_2$. Then if error converges to zero or a relatively small value, we stop our simulation.

4.1 Results of a Single Model with Bifurcation

For our single model, the numerical results looks very good. Figure 4.1 refers to the plot of error analysis for domain decomposition on the single model. X-axis stands for the number of iterations and Y-axis represents the value of errors. We can see errors in

the plots converge to a relatively small value. At the first glimpse, we can see that the trend seems to tend to zero.

However, since we do not have time to collect enough data to prove that our solution converges, we can only focus on our plots to make a general comment: the error will finally converge.

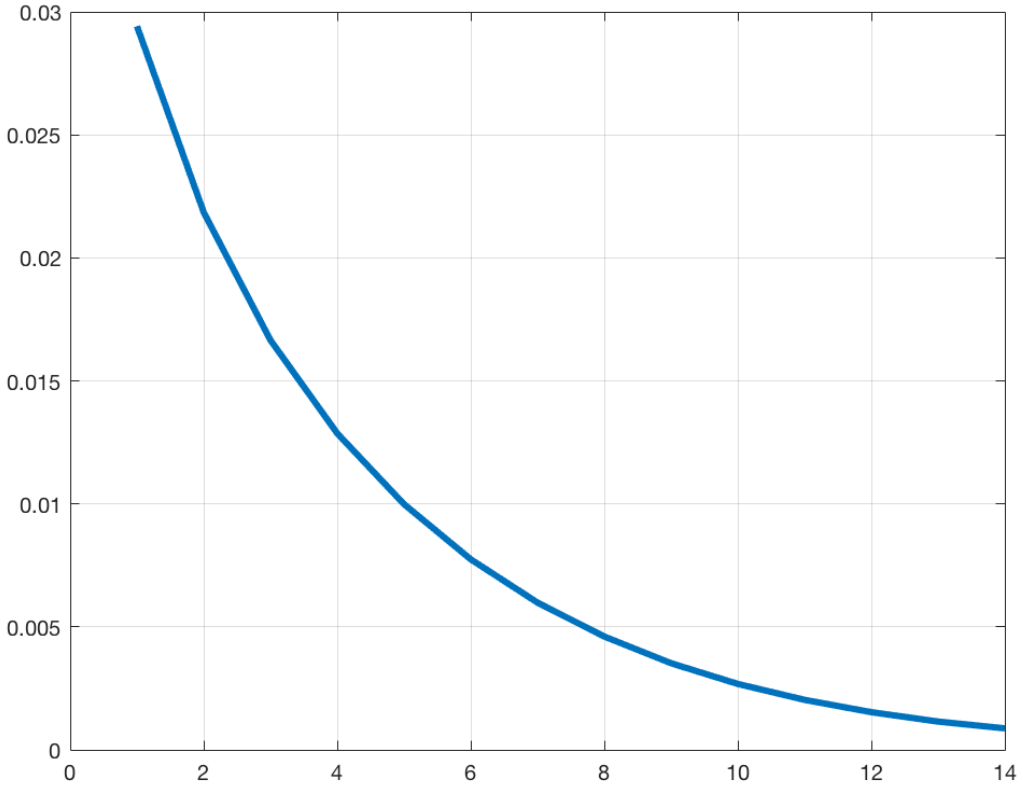
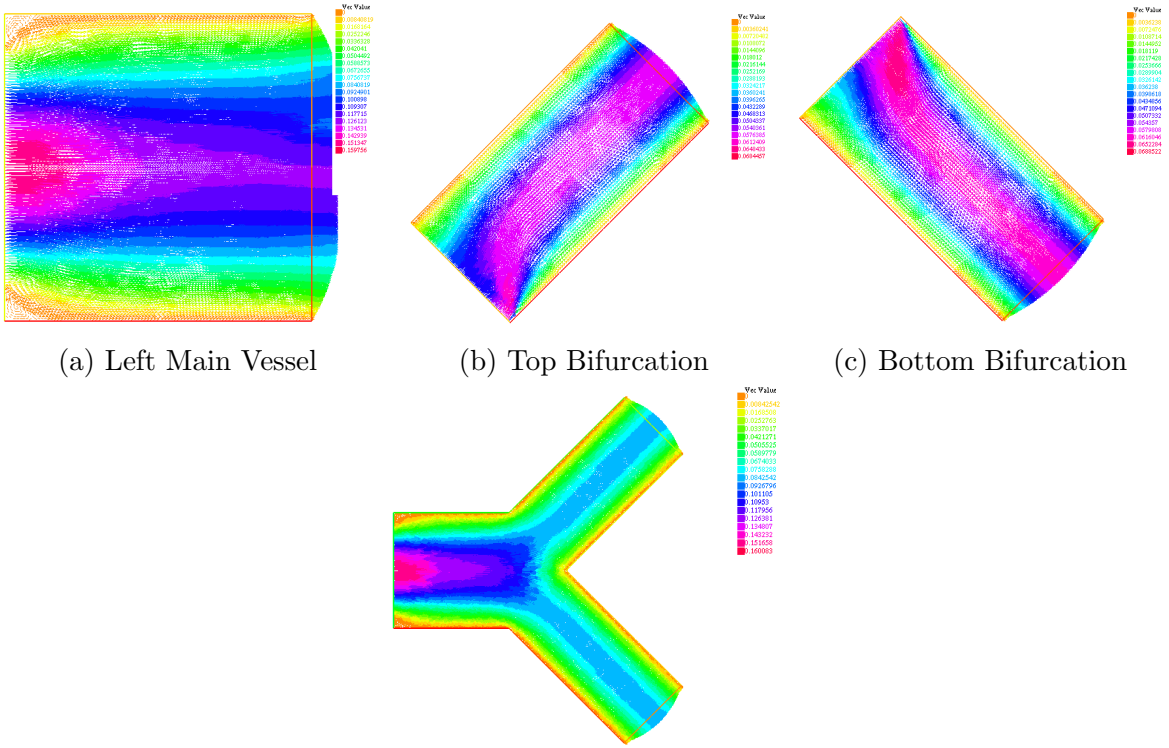


Figure 4.1: Error Plots for a Single Model with Domain Decomposition

Figure 4.2 and Figure 4.3 represent plots of velocity and pressure after FreeFem++ simulation. As said before, there are not enough data. Thus, one possible way is to compare our numerical solution with domain decomposition with that of the whole domain. Comparing each part eye by eye, we can see that with domain decomposition, our



(a) Left Main Vessel (b) Top Bifurcation (c) Bottom Bifurcation

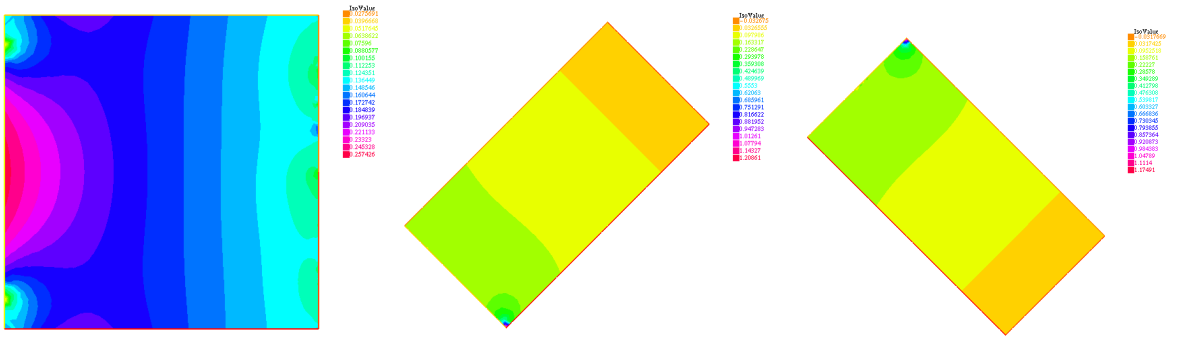
(d) Whole Single Model

Figure 4.2: Velocity Plots by FreeFem++ Using Domain Decomposition

solution is still close to that of the whole domain, which means in general for our single model, domain decomposition works well.

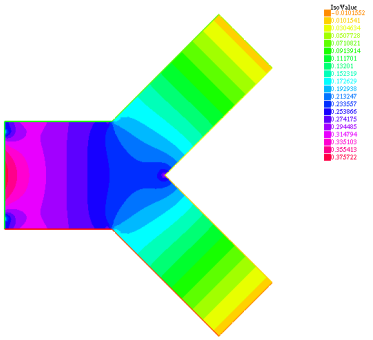
4.2 Results of Modified Model with Child Bifurcation

Let us focus on Figure 4.5. Unlike the error analysis in the previous section, some little oscillations occur in the error plot with domain decomposition on our modified model. However, the trend of the error plot here is still reasonable and tolerable. We may conclude that the error finally converges to some extent.



(a) Left Main Vessel (b) Top Bifurcation (c) Bottom Bifurcation

Figure 4.3: Pressure Plots by FreeFem++ Using Domain Decomposition



(a) Whole Single Model

Figure 4.7 and Figure 4.9 represent plots of velocity and pressure after FreeFem++ simulation for our modified model. Just like the result of previous section for single model, after comparing eye by eye with plots, we find that with domain decomposition, our solution is still close to that of the whole domain.

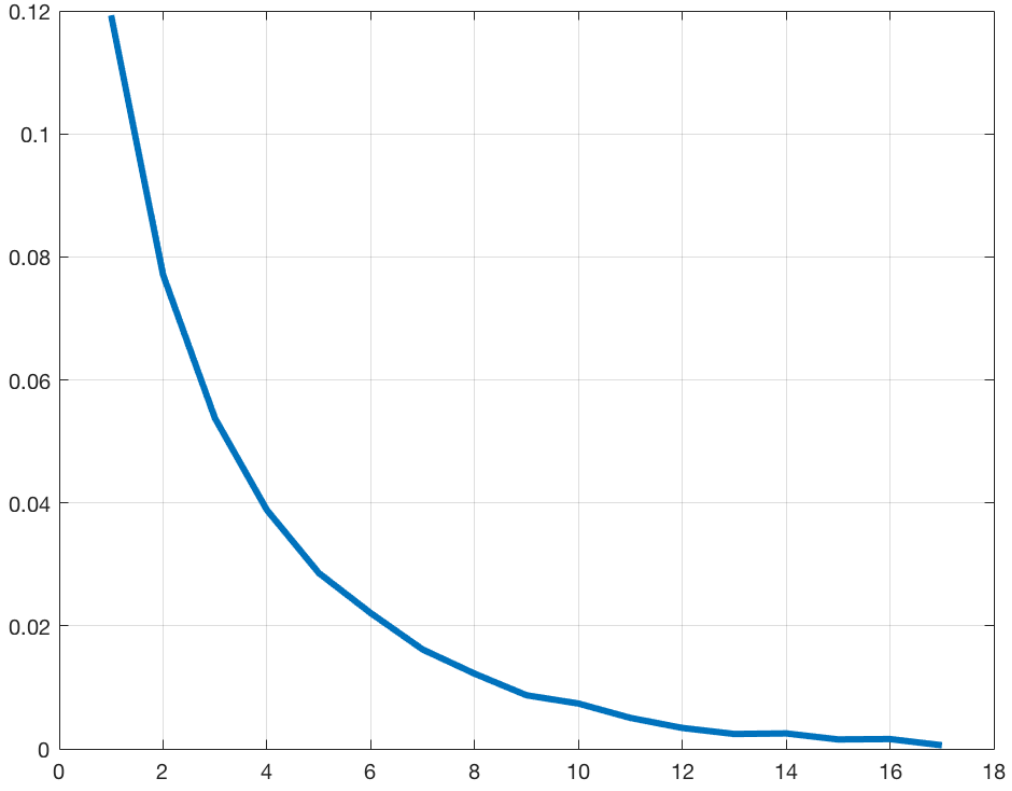
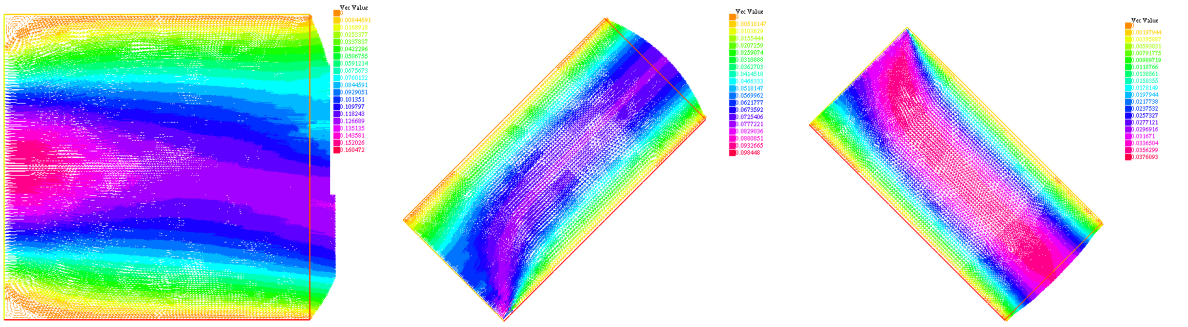
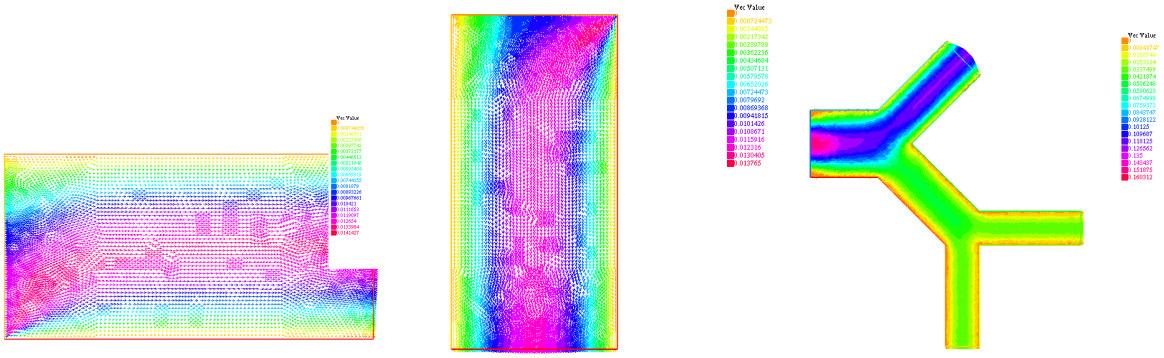


Figure 4.5: Error Plots for the Modified Model with Domain Decomposition



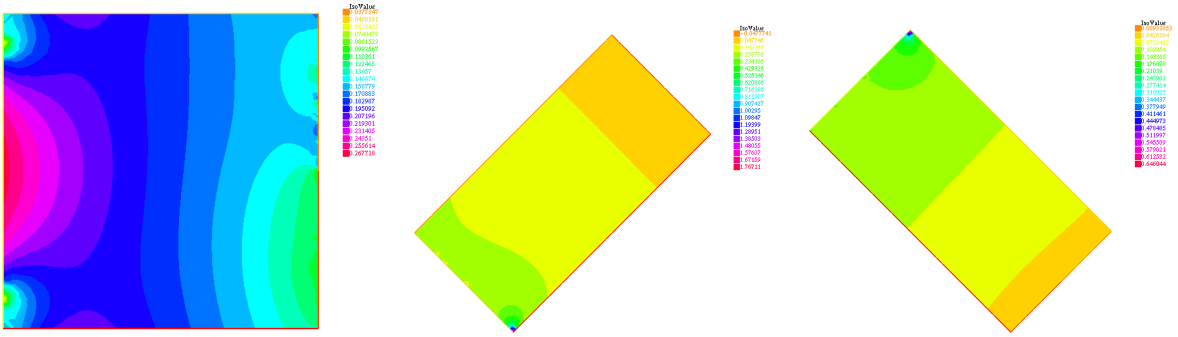
(a) Left Main Vessel (b) Top Bifurcation (c) Bottom Bifurcation

Figure 4.6: Velocity Plots by FreeFem++ Using Domain Decomposition



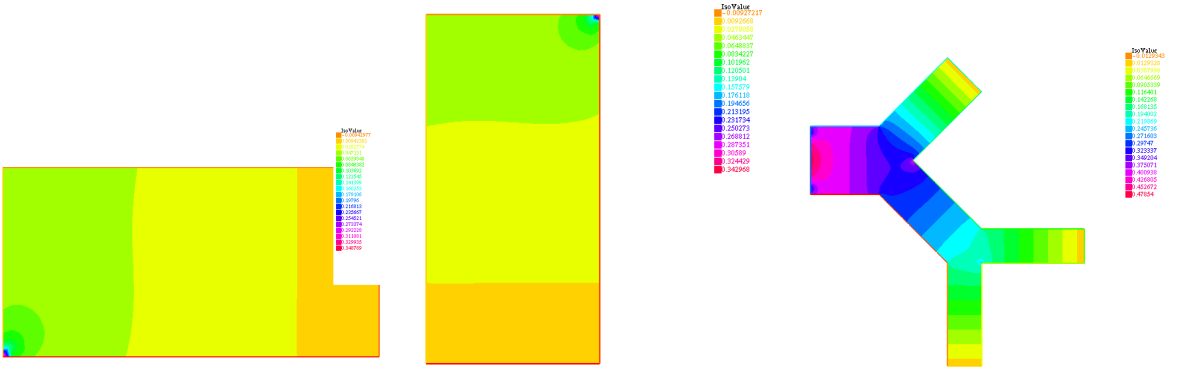
(a) Child Top Bifurcation (b) Child Bottom Bifurcation (c) Modified Whole Model

Figure 4.7: Velocity Plots by FreeFem++ Using Domain Decomposition



(a) Left Main Vessel (b) Top Bifurcation (c) Bottom Bifurcation

Figure 4.8: Velocity Plots by FreeFem++ Using Domain Decomposition



(a) Child Top Bifurcation (b) Child Bottom Bifurcation (c) Modified Whole Model

Figure 4.9: Pressure Plots by FreeFem++ Using Domain Decomposition

Chapter 5

Conclusion

In general, our project successfully explores the domain decomposition techniques in computational fluid dynamics for pipe-like domains in 2D. By domain decomposition, we can split a complex model into several simple parts in 2D on the incompressible Navier-Stokes equations. The results of the simulations are reasonable and to some extent good enough. We may conclude that the way we decompose our model is plausible and accurate. Therefore, it is possible for us to extend our 2D problems to 3D realistic cases for future research.

For future directions, we may explore the ordering of solving problem in domain decomposition. More specifically, during our work, we find that solving different parts in various orders can lead to different number of iterations required to converge and therefore resulting in different converging time. Sometimes, there are huge differences between the converging time using different orders. However, we do not have time to continue and to further explore it. If we could explore more the truth of ordering, then it is possible to

increase the efficiency of simulations in large circulatory networks and reduce the cost a lot.

Appendix A

FreeFem++ Code

listings xcolor

```
1  /* ****************************************************
2                                      *
3                                    *   *
4                                  *       *
5                                *       *
6                              *    2  *
7                            *       *
8                              *   *
9  * * * * * * * * * *  *        *
10 *        1          *
11 *                   *
12 * * * * * * * * * *  *      *
13                           *   *
14                         *       *
15                           *   3   *
16                             *       *
17                               *   *
18                                 *      * * * * * *
19                                        *    4    *
20                                        * * * * * *
```

```
21                                * * *
22                                *   *
23                                * 5 *
24                                *   *
25                                *   *
26                                * * *
27  ***************************************************** */
28  border down1(t=0,1){x=t;y=-0.5;label=1;};
29  border lowerRight1(t=-0.5,0){x=1;y=t;label=2;};
30  border upperRight1(t=0,0.5){x=1;y=t;label=3;};
31  border up1(t=1,0){x=t;y=0.5;label=4;};
32  border left1(t=0.5,-0.5){x=0.0;y=t;label=5;};
33  int nm=32;
34
35  mesh Th1=buildmesh(down1(nm)+lowerRight1(nm)+upperRight1(nm)+up1(nm)+left1(nm)); // first domain
36  plot(Th1,wait=1);
37
38  ofstream merror("materrors.m");
39
40  merror << "error=[ ";
41
42  fespace X1h(Th1,P2);
43  X1h u11,u12,u11old,u12old,v11,v12,k11,k12;
44  fespace Q1h(Th1,P1); //2
45  Q1h p1,q1;
46
47  //-----------------------------------------------------------------
48  border down2(t=1.5,2.5){x=t;y=t-1.5;label=1;};
49  border right2(t=2.5,2){x=t;y=-t+3.5;label=2;};
50  border up2(t=2,1){x=t;y=t-0.5;label=3;};
51  border left2(t=1,1.5){x=t;y=-t+1.5;label=4;};
52
53  mesh Th2=buildmesh(down2(nm)+right2(nm)+up2(nm)+left2(nm)); // second upper-right domain
54  plot(Th2,wait=1);
55
56  fespace X2h(Th2,P2);
```

```
57 X2h u21,u22,u21old,u22old,v21,v22,k21,k22;
58 fespace Q2h(Th2,P1); //2
59 Q2h p2,q2,p2old;
60
61 //----------------------------------------------------------------------
62
63 border down3(t=1,2){x=t;y=-t+0.5;label=1;};
64 border downright3(t=2,2.25){x=t;y=t-3.5;label=2;};
65 border upright3(t=2.25,2.5){x=t;y=t-3.5;label=3;};
66 border up3(t=2.5,1.5){x=t;y=-t+1.5;label=4;};
67 border left3(t=1.5,1){x=t;y=t-1.5;label=5;};
68
69 mesh Th3=buildmesh(up3(nm)+downright3(nm)+upright3(nm)+down3(nm)+left3(nm)); //third lower-right domain
70 plot(Th3,wait=1);
71
72
73 fespace X3h(Th3,P2);
74 X3h u31,u32,u31old,u32old,v31,v32,k31,k32;
75 fespace Q3h(Th3,P1); //2
76 Q3h p3,q3,p3old;
77
78 //----------------------------------------------------------------------
79
80 border down4(t=2.5,3.5){x=t;y=-1.5;label=1;};
81 border right4(t=-1.5,-1){x=3.5;y=t;label=2;};
82 border up4(t=3.5,2.5){x=t;y=-1;label=3;};
83 border left4(t=-1,-1.5){x=2.5;y=t;label=4;};
84
85 mesh Th4=buildmesh(up4(nm)+right4(nm)+down4(nm)+left4(nm)); //right child domain
86 plot(Th4,wait=1);
87
88 fespace X4h(Th4,P2);
89 X4h u41,u42,u41old,u42old,v41,v42,k41,k42;
90 fespace Q4h(Th4,P1); //2
91 Q4h p4,q4,p4old;
92
```

```
93  //-----------------------------------------------------------------
94  border down5(t=2,2.5){x=t;y=-2.5;label=1;};
95  border right5(t=-2.5,-1.5){x=2.5;y=t;label=2;};
96  border up5(t=2.5,2){x=t;y=-1.5;label=3;};
97  border left5(t=-1.5,-2.5){x=2;y=t;label=4;};
98
99  mesh Th5=buildmesh(up5(nm)+right5(nm)+down5(nm)+left5(nm)); //down child domain
100 plot(Th5,wait=1);
101
102 fespace X5h(Th5,P2);
103 X5h u51,u52,u51old,u52old,v51,v52,k51,k52;
104 fespace Q5h(Th5,P1); //2
105 Q5h p5,q5,p5old;
106
107 //-----------------------------------------------------------------
108 u11=0.; u12=0.; u11old = 0.; u12old=0.;
109 u21=0.; u22=0.; u21old = 0.; u22old=0.;
110 u31=0.; u32=0.; u31old = 0.; u32old=0.;
111 u41=0.; u42=0.; u41old = 0.; u42old=0.;
112 u51=0.; u52=0.; u51old = 0.; u52old=0.;
113 p1=0.; p2=0.;p2old=0.; p3=0.;p3old=0.;p4=0.;p4old=0.; p5=0.;p5old=0.;
114 q1=0.; q2=0.;q3=0.;q4=0.;q5=0.;
115 k11=u11;k12=u12;
116 k21=u21;k22=u22;
117 k31=u31;k32=u32;
118 k41=u41;k42=u42;
119 k51=u51;k52=u52;
120
121 func g=(0.4-y)*(y+0.4)*(y<0.4)*(y>-0.4);
122 real  nu=1.e-1;
123 real toll=1.e-4;
124 real gamma=.3;
125 real P11= 0.35;
126 real errL2=0.,errL2max=0.;
127
128 problem Stokes1 ([u11,u12,p1],[v11,v12,q1],solver=UMFPACK) =
```

```
129      int2d(Th1)(//dti*u1*v1 + dti*u2*v2 +
130                nu * ( dx(u11)*dx(v11) + dy(u11)*dy(v11)
131                +      dx(u12)*dx(v12) + dy(u12)*dy(v12) ))
132
133    - int2d(Th1)(p1*dx(v11) + p1*dy(v12))
134    - int2d(Th1)(dx(u11)*q1 + dy(u12)*q1)
135
136        + int2d(Th1)( (k11*dx(u11)+k12*dy(u11))*v11
137                +    (k11*dx(u12)+k12*dy(u12))*v12 )
138
139    + int1d(Th1,3)(((p2-nu*dx(u21))*N.x-nu*dy(u21)*N.y)*v11+
140                 ((p2-nu*dy(u22))*N.y-nu*dx(u22)*N.x)*v12)
141
142    + int1d(Th1,2)(((p3-nu*dx(u31))*N.x-nu*dy(u31)*N.y)*v11+
143                 ((p3-nu*dy(u32))*N.y-nu*dx(u32)*N.x)*v12)
144
145    + on(5,u11=g,u12=0.)
146    + on(1,4,u11=0.,u12=0.);
147
148
149  problem Stokes2 ([u21,u22,p2],[v21,v22,q2],solver=UMFPACK) =
150      int2d(Th2)(//dti*u1*v1 + dti*u2*v2 +
151                nu * ( dx(u21)*dx(v21) + dy(u21)*dy(v21)
152                +         dx(u22)*dx(v22) + dy(u22)*dy(v22) ))
153    - int2d(Th2)(p2*dx(v21) + p2*dy(v22))
154    - int2d(Th2)(dx(u21)*q2 + dy(u22)*q2)
155      + int2d(Th2)( (k21*dx(u21)+k22*dy(u21))*v21
156                +    (k21*dx(u22)+k22*dy(u22))*v22 )
157    + on(4,u21=-u11*N.x*3.0/4.0,u22=-u11*N.y*3.0/4.0)
158    + on(1,3,u21=0.,u22=0.);
159
160  problem Stokes3 ([u31,u32,p3],[v31,v32,q3],solver=UMFPACK) =
161      int2d(Th3)(//dti*u1*v1 + dti*u2*v2 +
162                nu * ( dx(u31)*dx(v31) + dy(u31)*dy(v31)
163                +         dx(u32)*dx(v32) + dy(u32)*dy(v32) ))
164    - int2d(Th3)(p3*dx(v31) + p3*dy(v32))
```

```
  - int2d(Th3)(dx(u31)*q3 + dy(u32)*q3)
    + int2d(Th3)( (k31*dx(u31)+k32*dy(u31))*v31
            +   (k31*dx(u32)+k32*dy(u32))*v32 )
  + int1d(Th3,3)(((p4-nu*dx(u41))*N.x-nu*dy(u41)*N.y)*v31+
             ((p4-nu*dy(u42))*N.y-nu*dx(u42)*N.x)*v32)
  + int1d(Th3,2)(((p5-nu*dx(u51))*N.x-nu*dy(u51)*N.y)*v31+
             ((p5-nu*dy(u52))*N.y-nu*dx(u52)*N.x)*v32)
  + on(5,u31=-u11*N.x*1.0/4.0,u32=-u11*N.y*1.0/4.0)
  + on(1,4,u31=0.,u32=0.);

problem Stokes4 ([u41,u42,p4],[v41,v42,q4],solver=UMFPACK) =
    int2d(Th4)(//dti*u1*v1 + dti*u2*v2 +
            nu * ( dx(u41)*dx(v41) + dy(u41)*dy(v41)
            +        dx(u42)*dx(v42) + dy(u42)*dy(v42) ))
  - int2d(Th4)(p4*dx(v41) + p4*dy(v42))
  - int2d(Th4)(dx(u41)*q4 + dy(u42)*q4)
    + int2d(Th4)( (k41*dx(u41)+k42*dy(u41))*v41
            +   (k41*dx(u42)+k42*dy(u42))*v42 )
  + on(4,u41=-u31*N.x*0.5,u42=-u31*N.y*0.5)
  + on(1,3,u41=0.,u42=0.);

problem Stokes5 ([u51,u52,p5],[v51,v52,q5],solver=UMFPACK) =
    int2d(Th5)(//dti*u1*v1 + dti*u2*v2 +
            nu * ( dx(u51)*dx(v51) + dy(u51)*dy(v51)
            +        dx(u52)*dx(v52) + dy(u52)*dy(v52) ))
  - int2d(Th5)(p5*dx(v51) + p5*dy(v52))
  - int2d(Th5)(dx(u51)*q5 + dy(u52)*q5)
   + int2d(Th5)( (k51*dx(u51)+k52*dy(u51))*v51
   +   (k51*dx(u52)+k52*dy(u52))*v52 )
  + on(3,u51=-u31*N.x*0.5,u52=-u31*N.y*0.5)
  + on(4,2,u51=0.,u52=0.);

real[int]xx(50); real[int]yy(50);
real thr=0.001;
real Error=10^5;
int nmax = 200;
```

```
201  int j=0;
202 while(Error>thr && j<nmax )
203 {
204  cout << "Iteration " << j << endl;
205  u11old=u11; u12old=u12;
206  u21old=u21; u22old=u22; p2old=p2;
207  u31old=u31; u32old=u32; p3old=p3;
208  u41old=u41; u42old=u42; p4old=p4;
209  u51old=u51; u52old=u52; p5old=p5;
210
211  Stokes3;
212  plot([u31,u32],value=true,nbiso=10,wait=false);//,ps="es10_vel_a.eps");
213  plot(p3,fill=1,value=true,wait=false);//,ps="es10_pre_a.eps");
214
215  Stokes4;
216 plot([u41,u42],value=true,nbiso=10,wait=false);//,ps="es10_vel_a.eps");
217 plot(p4,fill=1,value=true,wait=false);//,ps="es10_pre_a.eps");
218
219  Stokes5;
220 plot([u51,u52],value=true,nbiso=10,wait=false);//,ps="es10_vel_a.eps");
221 plot(p5,fill=1,value=true,wait=false);//,ps="es10_pre_a.eps");
222
223  Stokes1;
224 plot([u11,u12],value=true,nbiso=10,wait=false);//,ps="es10_vel_a.eps");
225 plot(p1,fill=1,value=true,wait=false);//,ps="es10_pre_a.eps");
226
227  Stokes2;
228 plot([u21,u22],value=true,nbiso=10,wait=false);//,ps="es10_vel_a.eps");
229 plot(p2,fill=1,value=true,wait=false);//,ps="es10_pre_a.eps");
230
231 real Error1 = int2d(Th1)((u11-u11old)^2+(u12-u12old)^2);
232  Error1 = sqrt(Error1);
233 cout << "Error1 " << Error1 << endl;
234  real Error2 = int2d(Th2)((u21-u21old)^2+(u22-u22old)^2);
235  Error2 = sqrt(Error2);
236 cout << "Error2 " << Error2 << endl;
```

```
237  real Error3 = int2d(Th3)((u31-u31old)^2+(u32-u32old)^2);
238  Error3 = sqrt(Error3);
239 cout << "Error3 " << Error3 << endl;
240 real Error4 = int2d(Th4)((u41-u41old)^2+(u42-u42old)^2);
241  Error3 = sqrt(Error4);
242 cout << "Error4 " << Error4 << endl;
243 real Error5 = int2d(Th5)((u51-u51old)^2+(u52-u52old)^2);
244  Error5 = sqrt(Error5);
245 cout << "Error5 " << Error5 << endl;
246  Error = int2d(Th1)((u11-u11old)^2+(u12-u12old)^2)+int2d(Th2)((u21-u21old)^2+(u22-u22old)^2) + int2d(Th
247               int2d(Th4)((u41-u41old)^2+(u42-u42old)^2)+ int2d(Th5)((u51-u51old)^2+(u52-u52old)^2);
248  Error = sqrt(Error);
249  cout << "Convergence " << Error << endl;
250
251 u11 = gamma*u11 + (1-gamma)*u11old;
252 u12 = gamma*u12 + (1-gamma)*u12old;
253
254 u21 = gamma*u21 + (1-gamma)*u21old;
255 u22 = gamma*u22 + (1-gamma)*u22old;
256 p2 = gamma*p2 + (1-gamma)*p2old;
257
258 u31 = gamma*u31 + (1-gamma)*u31old;
259 u32 = gamma*u32 + (1-gamma)*u32old;
260 p3 = gamma*p3 + (1-gamma)*p3old;
261
262 u41 = gamma*u41 + (1-gamma)*u41old;
263 u42 = gamma*u42 + (1-gamma)*u42old;
264 p4 = gamma*p4 + (1-gamma)*p4old;
265
266 u51 = gamma*u51 + (1-gamma)*u51old;
267 u52 = gamma*u52 + (1-gamma)*u52old;
268 p5 = gamma*p5 + (1-gamma)*p5old;
269
270 k11=u11;k12=u12;
271 k21=u21;k22=u22;
272 k31=u31;k32=u32;
```

```
k41=u41;k42=u42;
k51=u51;k52=u52;
 xx[j]=Error;
 yy[j]=j;
merror << xx[j] << "; \n";


j++;
}

merror << "]; \n";
plot([xx,yy],value=true,nbiso=10,wait=true);
plot(p1,[u11,u12],value=true,wait=false);
plot(p2,[u21,u22],value=true,wait=false);
plot(p3,[u31,u32],value=true,wait=false);
plot(p4,[u41,u42],value=true,wait=false);
plot(p5,[u51,u52],value=true,wait=false);
plot([u11,u12],value=true,nbiso=10,wait=1);//,ps="es10_vel_a.eps");
plot(p1,fill=1,value=1,wait=1);//,ps="es10_pre_a.eps");
errL2 = int2d(Th1)((u11-g)*(u11-g)+u12*u12);
errL2 = sqrt(errL2);
cout << "Error vs Exact " << errL2 << endl;
plot([u21,u22],value=true,nbiso=10,wait=1);//,ps="es10_vel_a.eps");
plot(p2,fill=1,value=1,wait=1);//,ps="es10_pre_a.eps");
errL2 = int2d(Th2)((u21-g)*(u21-g)+u22*u22);
errL2 = sqrt(errL2);
cout << "Error vs Exact " << errL2 << endl;
plot([u31,u32],value=true,nbiso=10,wait=1);//,ps="es10_vel_a.eps");
plot(p3,fill=1,value=1,wait=1);//,ps="es10_pre_a.eps");
errL2 = int2d(Th3)((u31-g)*(u31-g)+u32*u32);
errL2 = sqrt(errL2);
cout << "Error vs Exact " << errL2 << endl;
plot([u41,u42],value=true,nbiso=10,wait=1);//,ps="es10_vel_a.eps");
plot(p4,fill=1,value=1,wait=1);//,ps="es10_pre_a.eps");
errL2 = int2d(Th4)((u41-g)*(u41-g)+u42*u42);
errL2 = sqrt(errL2);
```

```
309 cout << "Error vs Exact " << errL2 << endl;
310 plot([u51,u52],value=true,nbiso=10,wait=1);//,ps="es10_vel_a.eps");
311 plot(p5,fill=1,value=1,wait=1);//,ps="es10_pre_a.eps");
312 errL2 = int2d(Th5)((u51-g)*(u51-g)+u52*u52);
313 errL2 = sqrt(errL2);
314 cout << "Error vs Exact " << errL2 << endl;
```

Bibliography

[1] Lucas O. Müller, Pablo J. Blanco, Sansuke M. Watanabe, and Raúl A. Feijóo. A high-order local time stepping finite volume solver for one-dimensional blood flow simulations: application to the adan model. *International Journal for Numerical Methods in Biomedical Engineering*, 32(10):e02761. doi: 10.1002/cnm.2761. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cnm.2761.

[2] Pablo J Blanco, Sansuke M Watanabe, Marco Aurélio RF Passos, Pedro A Lemos, and Raúl A Feijóo. An anatomically detailed arterial network model for one-dimensional computational hemodynamics. *IEEE Transactions on biomedical engineering*, 62(2): 736–753, 2015.

[3] Belle Dumé. Nanoparticles control blood-vessel growth. June 2013. URL https://physicsworld.com/a/nanoparticles-control-blood-vessel-growth/.

[4] Alfio Quarteroni and Silvia Quarteroni. *Numerical models for differential problems*, volume 2. Springer, 2009.

[5] Luca Formaggia, Fausto Saleri, and Alessandro Veneziani. *Solving numerical PDEs: Problems, applications, exercises*. Springer Science & Business Media, 2012.