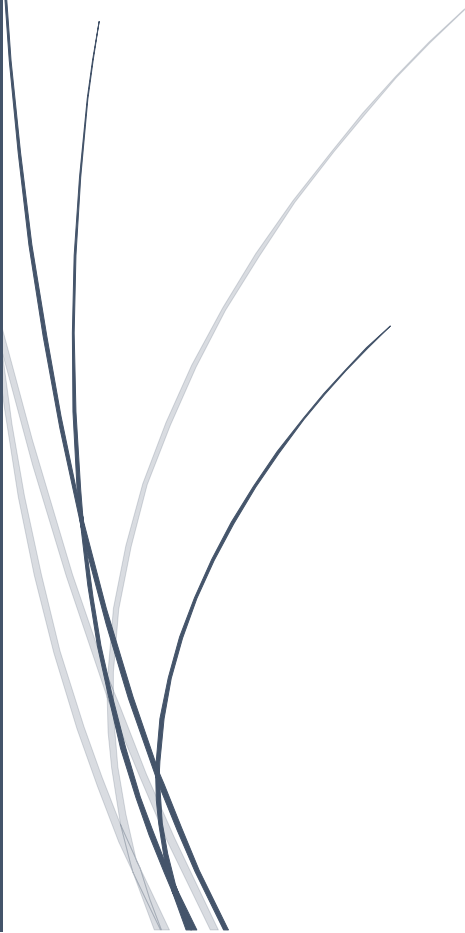


A dark blue vertical bar is on the left. A blue arrow points right from it, containing the date.

7/3/2021

ELIXIR

A PYTHON PACKAGE TO SOLVE 1D
BLOOD FLOW PROBLEMS

- V Abhijith Narayan Rao
 - Sanaul Malik Shaheer
- 
- Several thin, curved lines in dark blue and light grey originate from the bottom left and curve upwards and to the right.

Contents

ABSTRACT	2
INTRODUCTION.....	2
WHY THIS PROJECT?.....	4
IMPLEMENTATIONAL DETAILS	4
SOLVING DIFFERENTIAL EQUATIONS USING NEURAL NETWORKS	9

ABSTRACT

Modeling blood flow and especially the propagation of the pulse wave in systemic arteries is a topic that is interesting to the medical society since the shape of the pressure profiles has diagnostic significance. We build a package that can simulate blood flow and pressure in large arteries by solving a nonlinear one-dimensional model based on the incompressible Navier-Stokes equations for a Newtonian fluid in an elastic tube. Our method, however, does not require the usage of discretized methods for solving differential equations such as the popular Lax-Wendroff method (LW) but instead uses automatic differentiation to achieve a similar result.

INTRODUCTION

Partial differential equations (PDEs) and Ordinary Differential Equations (ODEs) are extensively used in mathematical modeling of various physics, engineering, finance, and healthcare problems.

A **differential equation** is an **equation** that contains one or more terms and the derivatives of one (i.e., dependent variable) concerning the other variables (i.e., independent variables) $dy/dx = f(x)$. Here, "x" is an independent variable and "y" is a dependent variable. Differential equations are everywhere, mostly because we can model complex systems as functions, with inputs of the system as the independent variable and the outputs of the system as the response or the dependent variable of the system.

However, in practical situations, these equations typically lack analytical solutions or are only too challenging to solve and are hence solved numerically. In current practice, most numerical approaches to solve ODEs/PDEs like finite element method (FEM), finite difference method (FDM), and finite volume method (FVM) are mesh-based. A typical implementation of a mesh-based approach involves three steps:

1. Grid Generation
2. Discretization of governing equation, and
3. The solution of the discretized equation using some iterative approach.

However, there are limitations to these approaches. Some of the limitations of these methods are as follows:

1. They cannot solve PDEs in complex computational domains because grid generation (step 1) becomes infeasible.
2. The discretization process (step 2) creates a discrepancy between actual ODE/PDE's mathematical nature and its approximate difference equation. Sometimes this can lead to quite serious problems.

One of the prospects to fix these concerns is to use neural networks. In this approach, the data set consists of randomly selected points in the domain and on the boundary, called the collocation points. That dataset then dictates the tuning of the NNs parameters.

There are two primary motivations for this approach. First, being universal approximators, neural networks can potentially represent any ODE/PDE. So, this avoids the discretization step and thus discretization-based physics errors too. Second, it is mesh-free, and therefore complex geometries can be easily handled.

Initial work in this direction can be credited to Lagaris et al. Firstly, they solved the initial boundary value problem using neural networks and later extended their work to handle irregular boundaries. Since then, the field has grown exponentially. In particular, we refer to the physics-informed neural networks (PINN) approach by Raissi and Karniadakis and Raissi et al. This approach has produced promising results for a series of benchmark nonlinear problems.

Now, as far as the current standards in the simulation of blood flow in systemic arteries are concerned, the standard procedure would be to discretize the governing equations, followed by reaching its solution using popular mesh-based approaches such as the Lax-Wendroff method or the Galerkin Finite element method. Despite its immense success in performing such simulations, these methods, as stated previously, lack the generation of a continuous solution that the medical society may benefit from significantly.

Keeping that in mind, we have developed Elixir, a one-dimensional blood flow simulation package that utilizes the endowment of neural networks to develop a continuous solution for the simulation, which leads to smoother and more accurate graphs.

We have also ensured that Elixir's solver is extremely simple to understand and modify, which helps develop a more complex solution, leading to better results in general.

WHY THIS PROJECT?

Modeling blood flow and pressure in the systemic arteries has been a topic of interest both to theoretical and clinical investigators. Thus, research in this area has a vital interdisciplinary aspect. This project aims to develop a package capable of performing such simulations using the models we develop to treat cardiovascular diseases better. This is important since most deaths in developed countries result from cardiovascular diseases, mostly associated with abnormal flow in the arteries.

The original project's inspiration arose from previous and present efforts to develop an anaesthesia simulator based on mathematical models. An important part of which is having a good model for the cardiovascular system.

However, as stated previously, the traditional focus of such projects generally is developing a good model.

The choice of the method to be used to solve the associated equations generally comes from a standard list of such methods.

We present a new method that, unlike its predecessors, offers a continuous solution, along with other benefits such as GPU support, a massive community, and such like.

IMPLEMENTATIONAL DETAILS

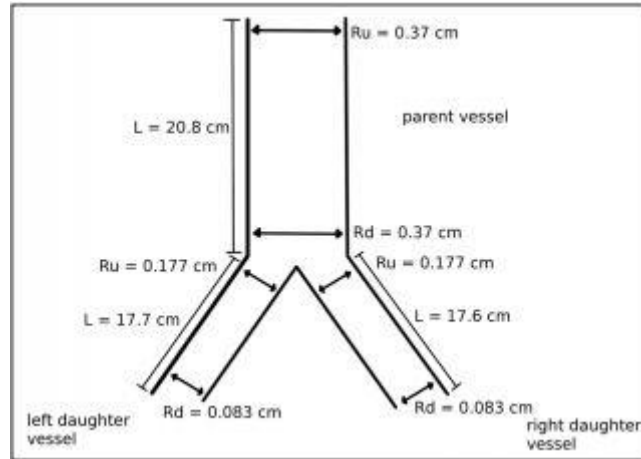
We now discuss the capabilities of our package, along with the mathematical development of the same.

We have made a few assumptions about the structural nature of the arteries we simulate, which we now explain.

We considered arteries to be elastic axisymmetrical tubes of initial radius $r_0(z)$ in a cylindrical coordinate system. The radius at rest is allowed to taper exponentially for an arterial segment, and the following equation explicates it:

$$r_0(z) = R_u \cdot \exp\left(\log\left(\frac{R_d}{R_u}\right) \frac{z}{L}\right)$$

The following figure demonstrates the bifurcation geometry for a common carotid artery, which we use to validate the solution calculated by Elixir.



Note: Currently, Elixir only supports the simulation of large arteries, i.e., arteries with radii greater than 100 μ m.

We now describe the motion of blood through the artery. Now, blood is a very complicated plasma of elastic suspensions; examples of these suspensions include red blood cells (erythrocytes) and white blood cells (leucocytes). This renders blood flow simulation a problematic task. By definition, a fluid is Newtonian if the viscosity coefficient is constant at all shear rates. This applies to most homogeneous liquids, including blood plasma, but in a liquid with a suspension of particles, the mechanical behaviour can deviate such that the liquid becomes non-Newtonian. These deviations become particularly significant when the particle size becomes appreciably large compared to the channel's dimension in which the fluid is flowing.

Hence, in large vessels, it is reasonable to regard the blood viscosity as constant, both because the vessel diameters are large compared with the individual cell diameters, and shear rates are high enough for viscosity to be independent of them.

Since, as stated previously, it is safe to regard blood as Newtonian, at least in the case of large vessels, we can use the Navier-Stokes equation to describe the motion of blood.

In physics, the **Navier–Stokes equations** are a set of partial differential equations describing the motion of viscous fluid substances, named after the French engineer and physicist Claude-Louis Navier and Anglo-Irish physicist and mathematician George Gabriel Stokes.

The Navier–Stokes equations mathematically express conservation of mass and momentum for Newtonian fluids. An equation of state relating to pressure, temperature, and density generally accompanies the equations. They arise from applying Isaac Newton's second law to fluid motion, together with the assumption that the fluid's stress is the sum of a diffusing viscous term (proportional to the gradient of velocity) and a pressure term, hence describing the viscous flow.

$$\frac{\partial u_z(r, z, t)}{\partial z} + \frac{1}{r} \frac{\partial(r u_r(r, z, t))}{\partial t} = 0$$

$$\frac{\partial u_z(r, z, t)}{\partial t} + u_z(r, z, t) \frac{\partial u_z(r, z, t)}{\partial z} + u_r(r, z, t) \frac{\partial u_z(r, z, t)}{\partial r} + \frac{1}{\rho} \frac{\partial p(z, t)}{\partial z} = \frac{\nu}{r} \frac{\partial}{\partial r} \left(r \frac{\partial u_z(r, z, t)}{\partial r} \right)$$

where $u = (u_z(r, z, t), u_r(r, z, t))$ denotes blood flow velocity, $p(z, t)$ denotes blood pressure, which is assumed to be uniform across r and the parameters ρ and ν denote blood density and viscosity, respectively. By integrating the governing equations over cross-sectional area, $A(z, t) = \pi R(z, t)^2$, the 1D conservation law can be derived.

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial z} = S$$

Where,

$$U = \begin{pmatrix} A(z, t) \\ q(z, t) \end{pmatrix}, F = \begin{pmatrix} q(z, t) \\ \frac{q(z, t)^2}{A(z, t)} + f(r_0) \sqrt{A_0(z) A(z, t)} \end{pmatrix}$$

$$S = \begin{pmatrix} 0 \\ S_1 \end{pmatrix}$$

With,

$$S_1 = -\frac{2\pi R(z, t)}{\delta_b Re} \frac{q(z, t)}{A(z, t)} + (2\sqrt{A(z, t)}) \left(\sqrt{\pi} f(r_0) + \sqrt{A_0(z)} \frac{df(r_0)}{dr_0} \right) - A(z, t) \frac{df(r_0)}{dr_0} \frac{dr_0(z)}{dz}$$

Here, the unknowns are the vessel cross-sectional area $A(z, t)$ and flux $q(z, t)$. The elasticity of the vessel is described by the quantity $f(r_0)$ with relaxed vessel radius

$r_0(z)$, $A_0(z)$ is the relaxed cross-sectional vessel area, $R(z, t)$ the vessel radius, δ_b is the boundary layer thickness, and Re is Reynold's number.

Note: The flux $q(z, t)$ is simply the arterial pulse velocity multiplied by the cross-sectional area.

The solution to these equations is already accessible publicly via the project VaMpy, a Python simulation package similar to Elixir. However, the difference lies in Elixir deploying a meshless solution to these equations, whereas VaMpy utilizing Richtemeyer's 2-step Lax-Wendroff method to accomplish the same.

We now discuss the inflow and outflow conditions for our artery.

For the inflow conditions, at the aortic valve, practically all previous models specify the flow, i.e., $q(0, t)$, emanating from the aortic valve. This is done either from direct measurement or by deriving a function based on simpler models. Since we are interested in qualitative behaviour, we have chosen the latter. Such a function can be the following periodic function based on the cardiac output parameters and the cardiac period's length.

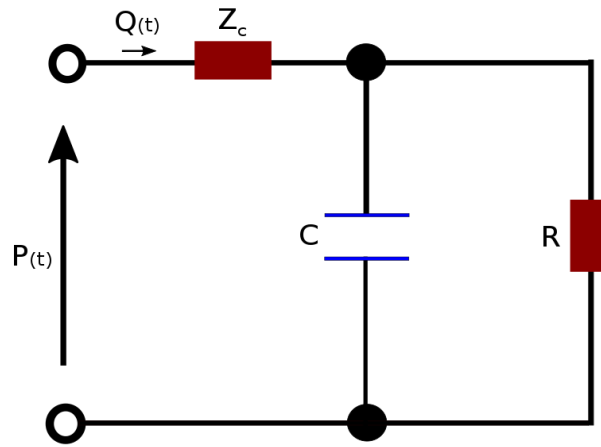
$$q(0, t) = \overline{q_0} t / \tau^2 \exp(-t^2 / (2\tau^2))$$

$$q(0, t + jT) = q(0, t), \quad j = 1, 2, 3 \dots$$

Here $\overline{q_0}$ is the cardiac output, τ is when the maximal cardiac output is reached, and T is the length of the cardiac period. The function has a jump at $t = T$, but it is of order 10^{-18} and can be neglected. Furthermore, it should be noted that the backflow into the left ventricle is not included.

To describe the outflow boundary condition, however, is slightly more complicated. For creating such a condition, we would have to find a straightforward model of a daughter artery that our current artery connects to. This modeling can be achieved in several ways, with the popular method being simulation using the 3-element Windkessel model.

In the windkessel model, we describe the daughter artery as an electrical circuit with the pressure representing the voltage and the flow representing the current. We then model the resistances and compliance offered by the artery as impedances and capacitance in the electrical circuit.



The daughter artery's peripheral resistance modeling is accomplished by the two impedances, as shown in the figure. However, we now need to consider the elasticity of the artery, which plays a role in creating the diastolic pressure. This is achieved by placing a capacitor in parallel to the second resistor.

The walls of large elastic arteries contain elastic fibres, formed of elastin. These arteries distend when the blood pressure rises during systole and recoil when the blood pressure falls during diastole. The electrical analogue of such an action is in natural choice, the capacitor. A capacitor charges when a potential is applied across it and discharges when the potential is removed, and hence we use it model the elasticity of the artery.

We again omit the derivation of the relation of the different variables present within the 3WK model.

$$\frac{\partial p(z, t)}{\partial t} = Z_c \frac{\partial q(z, t)}{\partial t} - \frac{p(z, t)}{RC} + \frac{q(z, t)(Z_c + R)}{RC}$$

where Z_c , R , and C are impedances and compliance parameters. However, this equation now poses another problem. The original Navier Stokes equation regard only the cross-sectional area and the flow as unknowns. We do not have any functional form for pressure. Hence, we need to derive a necessary relationship for the same. This is achieved by the state equation that relates the pressure with the cross-sectional area:

$$p(r_0, A) = \frac{4 Eh}{3 r_0} \left(1 - \sqrt{\frac{A_0}{A}}\right)$$

Here, E is the Young's modulus and h , the wall thickness. For more information regarding the state equation, please refer to Lighthill (1984).

Now, simulating a superficial artery is not enough if we would like to study the distribution of drugs or anaesthesia in the bloodstream, but instead requires us to model the entire arterial tree. We assume that the arterial tree can be modeled as a simple binary tree, implying that we assume that the arterial tree only contains bifurcations. The bifurcation conditions are,

$$q = q_{d1} + q_{d2}$$

$$p_p = p_{d1} = p_{d2}$$

The two bifurcation conditions are pretty straightforward to understand. The first condition states that the flow gets distributed at the bifurcation and that the sum of inflow into the daughter artery is equal to the outflow from the parent. The second condition is simply that the pressure at the bifurcation for all the arteries are equal.

SOLVING DIFFERENTIAL EQUATIONS USING NEURAL NETWORKS

We now introduce neural networks to solve differential equations – neural networks trained to solve supervised learning tasks while respecting any given law of physics described by general nonlinear partial differential equations.

The idea behind the solution is simple. We assume the function of interest within the differential equation; in our case, the flow and area, to be a neural network whose hyperparameter choice is left to the user's judgment.

We then utilize the differential equation as the optimization condition to train the neural network.

For example, for Berger's equation,

$$f := u_t + uu_x - (0.01/\pi)u_{xx}$$

We begin by approximating $u(t, x)$ as a neural network. To highlight the simplicity in implementing this idea, we have included a Python code snippet using Tensorflow; currently one of the most popular and well documented open-source libraries for machine learning computations. To this end, $u(t, x)$ can be defined as:

```
def u(t, x):  
    u = neural_net(tf.concat([t,x],1), weights, biases)  
    return u
```

And hence,

```
def f(t, x):  
    u = u(t, x)  
    u_t = tf.gradients(u, t)[0]  
    u_x = tf.gradients(u, x)[0]  
    u_xx = tf.gradients(u_x, x)[0]  
    f = u_t + u*u_x - (0.01/tf.pi)*u_xx  
    return f
```

Note: This code snippet was part of the original paper by Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Credits to the authors, shown here for demonstration purpose only.

The shared parameters between the neural networks $u(t, x)$ and $f(t, x)$ can be learned by minimizing the mean squared error loss,

$$MSE = MSE_u + MSE_f$$

With this, we can train a neural network to learn the parameters that best help it approximate the function under consideration.