4D MR Phase and Magnitude Segmentations with GPU Parallel Computing

By

Robert V. Bergen

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Department of Physics and Astronomy

University of Manitoba

Winnipeg

Dedication

To Robyn, who is always there for me.

To my father and grandfather, to whom my education was always very important.

Acknowledgments

I would like to thank my co-supervisors, Dr. Chris Bidinosti and Dr. Murray Alexander, for the support given and interest taken in this thesis. I am very grateful for the advice given in regards to the research, writing and editing of this work, as well as helping me on the way to submitting my research for publication for the first time.

I would like to thank both Dr. Murray Alexander and Dr. Randy Kobes for giving me my first research opportunity as an undergraduate many years ago. It proved to be a valuable experience, and the work I continued to do over the next few summers with Dr. Alexander were rewarding ones.

To Dr. Hung-yu Lin, Dr. Stephen Pistorious and Dr. Gabriel Thomas, thank you for agreeing to read my thesis as reviewers for my defense. I would also like to thank Dr. Lin for providing me the data used in this thesis, and for his time in answering any questions I had.

Thank you to Dr. Simon Liao and the Department of Applied Computer Science at the University of Winnipeg for the workspace which they provided for me. Many thanks to James Deng, who troubleshot the seemingly endless amount of computer glitches that arose over the course of my research.

I would also like to acknowledge the funding from this project from The University of Manitoba and the Faculty of Graduate Studies.

Abstract

The increasing size and number of data sets of large four dimensional (three spatial, one temporal) MR cardiac images necessitates efficient segmentation algorithms. Analysis of phase-contrast MR images yields cardiac flow information which can be manipulated to produce accurate segmentations of the aorta. New phase contrast segmentation algorithms are proposed that use simple mean-based calculations and least mean squared curve fitting techniques. The initial segmentations are generated on a multi-threaded CPU in 10 seconds or less. All CPU algorithms are sped up by a factor of 2 on the GPU, except for a more complex algorithm which fits flow data to Gaussian waveforms, and produces an initial segmentation in 0.5 seconds. This is a massive 2760x speedup over the CPU. The CPU computation time suggests that a complex segmentation of large 4D data sets are only practical with GPU computing. Level sets are applied to a magnitude image, where the initial conditions are given by the previous CPU and GPU algorithms. A qualitative comparison of results show that the GPU algorithm appears to produce the most accurate segmentation. After segmentation, particle trace simulations are run to visualize flow patterns in the aorta using two different representations of the flow field. One simulation plots streamlines which are lines tangent to the instantaneous flow field. The other plots pathlines which represent the path a virtual particle takes in the time-varying field. A procedure for the definition of analysis/emitter planes is proposed from which virtual particles can be emitted or collected within the vessel, which are useful for future quantification of various flow parameters.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cardiac magnetic resonance imaging (MRI) is a clinically-proven imaging modality that shows high levels of accuracy and reproducibility in a wide spectrum of cardiovascular diseases [1]. An important problem in MRI is detecting abnormalities in cardiac flow, which is currently done through echocardiography in three dimensions (two spatial, and one time). However, with the advance of 4-dimensional (three spatial, one time) MRI techniques, it is now possible to improve upon the previous methods to give more accurate, reliable and comprehensive clinical diagnosis of vascular diseases which can be life-threatening and difficult to detect. To study the dynamics of motion or flow during the cardiac cycle, a complete series of 3D datasets acquired at different times during the cardiac cyle is required. Phase contrast MRI can measure all three directions of blood flow velocity relative to all four spatio-temporal dimensions of the heart and great vessels which can then be analyzed to quantify regional flow and velocity parameters such as flow rate, peak velocity and retrograde flow [2, 3, 4, 5, 6]. Other hemodynamic parameters such as pressure differences and vessel wall shear stress can also be calculated [7, 8, 9, 10, 11]. Segmented data can aid in the early detection of congenital aortic disease leading to aortic aneurysms and dissections.

There have been a variety of proposed solutions to the four dimensional segmentation of the aorta. A common method for segmentation is the level set technique [12, 13]. This method creates a contour, and deforms it around an object by solving partial differential equations based on image intensity and the curvature of the contour itself. The contour is represented as the zero level set of a higher dimensional function, called a level set function, whose motion is formulated as an evolution equation. The main advantages of using level sets is that arbitrarily complex shapes can be modeled and topological changes such as merging and splitting are handled implicitly.

The most general level set, called the geodesic active contour (GAC) model, updates its level set function based on image intensity, image edges, and the curvature of the contour itself. However, in many cases the aorta boundary can be very difficult to detect because of low contrast within the image. In these cases, additional filters may be used along with the level set. One such solution is an Adapting Active Shape Model [14] which uses a level set method with a tubular filter to prevent the contour from expanding into areas of the image where the contrast is low. Others have acquired high resolution images of the aortic arch and renal arteries using phase-contrast information weighted by the image intensity of the magnitude image [15].

In this thesis we explore five different algorithms to segment the aorta from 4D cardiac MR images. The first algorithm uses only the magnitude information by preprocessing the magnitude image for application of the GAC level set. The second, third, and fourth algorithms use phase-contrast information exclusively. In the second algorithm, the cardiac cycle is separated into four different time regions, in which mean and standard deviations of the blood flow are calculated at each voxel for each directional component of the blood flow. The voxels whose values fall into an accepted range are segmented, while others are thrown away. The third algorithm instead fits Gaussian waveforms to the flow curves at each voxel, and segments the voxels whose Gaussian parameters fall into acceptable ranges. The fourth algorithm yields a similar segmentation by comparing the flow curves at each voxel to one standard Gaussian. Finally, the last algorithm uses both phase and magnitude information by initializing the GAC level set with the initial phase-based segmentations. This level set is then evolved on the magnitude image to conform the segmentation boundary to nearby magnitude image edges and accurately segment the aorta in three dimensions. All the segmentation algorithms are compared in terms of accuracy and processing time.

Because of the large amount of data obtained from 4D MRI scans and the computational needs of image processing, rapid methods of segmentation, 3D surface model construction and visualization are needed. Even the most accurate segmentations lose practicality once their run times stretch into hours or even days. Recently, Graphics Processing Units (GPUs) have become increasingly popular in the field of image processing, due to their ability to parallelize computations among its many cores, in a way that cannot be done with Central Processing Units (CPUs). For this reason, we optimize our segmentation methods for the GPU architecture. GPUs have already been shown to be useful in the medical image processing field; for example, the level set and other methods have already been implemented on GPU architecture [16, 17]. We show that for certain algorithms where calculations are sufficiently parallelizable, the GPU greatly outperforms the CPU to produce accurate results at speeds which would otherwise be impossible.

In the following chapter, the theory behind nuclear magnetic resonance and how it can be manipulated to produce MR images is reviewed, and the specific MR encoding and imaging sequences used to generate the 4D cardiac data analyzed in this thesis are presented. Chapter 3 explains each of the five segmentation methods, and discusses de-noising techniques for the initial segmentations. The level set is also discussed in detail, and it is shown how the various parameters in the level set equation controls the growth of the level set contour. Chapter 4 introduces applications of segmentation, specifically the visualization of blood flow in three dimensions and the definition of analysis planes within a segmented volume, which are useful for the quantifications of a variety of flow parameters. Chapter 5 discusses the idea of parallel computing and explains the advantages and drawbacks of GPUs in scientific computing. CPU vs. GPU runtimes are then compared for all our algorithms, and further possibilities for optimization are discussed. In Chapter 6 we discuss the advantages and limitations of our algorithms, and suggest possible improvements for future work.

Chapter 2

Principles of Magnetic Resonance Imaging

This chapter presents an overview of the theory behind MRI. First, the idea of nuclear magnetic resonance is introduced and we discuss how a net magnetization may be induced in an object and how it is mathematically formulated. We discuss how a signal may be detected that originates from this net magnetization, and how it may be manipulated through the application of various radio-frequency pulses and/or gradient fields (or imaging sequences) to yield spatial and velocity information of the object itself. As the following overview is meant to be brief, a more complete explanation of the processes involved can be found in "Principles of Magnetic Resonance Imaging" by Z.-P. Lhiang and P. C. Lauterbur, which was used as the main source for the content of this chapter [18].

2.1 Nuclear magnetic resonance

MRI is based on the phenomenon of nuclear magnetic resonance (NMR), which was first observed independently by Felix Bloch [19] and Edward Purcell [20]. As its name implies, NMR involves a resonance effect arising from the interactions between magnetic fields and the nuclei of the object being imaged. Then to describe the theory, we need to start at a nuclear level. Fortunately, most MRI principles can be accurately described using classical vector models because we are only concerned with the collective behaviour of the huge number of nuclei involved.

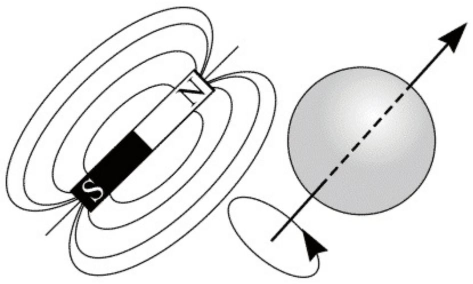


Figure 2.1: A charged particle spins about an axis and creates a magnetic field similar to that of a bar magnet. (Image from M. Puddephat [21].)

Nuclear magnetism describes the behaviour of a nucleus placed in an external magnetic field. For simplicity, consider the hydrogen nucleus. The nucleus rotates about its own axis if it has nonzero spin. Like any charged object, this spin creates a magnetic field around it, analogous to a bar magnet (Figure 2.1). This field can be represented by a vector quantity $\vec{\mu}$, which is called the nuclear magnetic dipole moment. This can be related to spin angular momentum by

$$\vec{\mu} = \gamma \vec{J}, \tag{2.1}$$

where γ is the gyro-magnetic ratio, whose value depends on the nucleus. Before going further, it is important to note that this description of magnetic moments is just a conceptual model, and cannot explain the magnetic moments of uncharged particles like the neutron. Ultimately, a quantum mechanical explanation is necessary for a complete description of spin and atomic magnetic moments.

Quantum mechanics tells us that the energy associated with a dipole moment in a magnetic field B_0 is

$$E = -\vec{\mu} \cdot \vec{B}_0 = -\mu_z B_0 = -\gamma \hbar m_I B_0, \tag{2.2}$$

where m_I is the spin quantum number and $\hbar$ is Planck's constant. For a hydrogen nucleus, $m_I =$

$\pm 1/2$. We refer to spins pointing along B_0 as spin"up". For spins pointing up, m_I is positive and

$$E_\uparrow = -\frac{1}{2}\gamma\hbar B_0, \tag{2.3}$$

while for spins aligned opposite to B_0, or spin "down" nuclei

$$E_\downarrow = \frac{1}{2}\gamma\hbar B_0, \tag{2.4}$$

and their energy difference is

$$\Delta E = E_\downarrow - E_\uparrow = \gamma\hbar B_0. \tag{2.5}$$

The difference between the energy levels of the two spin states is known as the Zeeman splitting phenomenon. This is illustrated in Figure 2.2. The difference in population of spins in each energy level is related to the energy difference between them. According to the Boltzmann equation, we get

$$\frac{N_\uparrow}{N_\downarrow} = \exp\left(\frac{\Delta E}{k_B T}\right) = \exp\left(\frac{\gamma\hbar B_0}{k_B T}\right), \tag{2.6}$$

where N is the population of spins pointing up ($\uparrow$) or down ($\downarrow$), T is the absolute temperature of the system, and k_B is the Boltzmann constant.

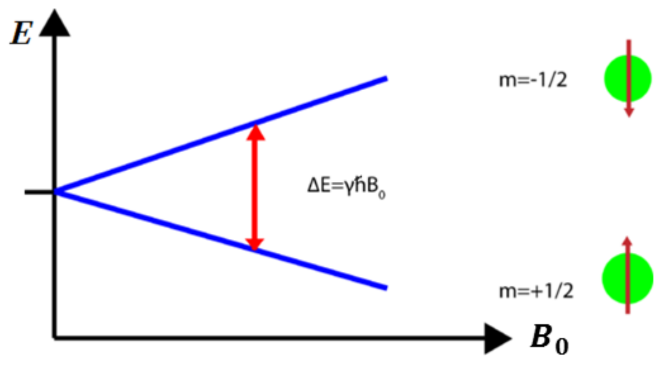


Figure 2.2: Zeeman splitting for a spin 1/2 particle, in an external field B_0. (Image modified from [22].)

Figure 2.3 shows what an entire object might look like before and after being subjected to an external magnetic field, where the number of spin up and spin down particles follows the Boltzmann distribution (Equation (2.6)).

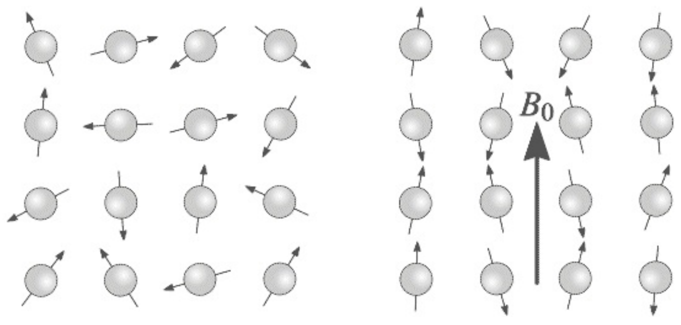


Figure 2.3: Spins are randomly aligned in an object when there is no field present. Application of a magnetic field aligns the spin parallel or anti-parallel to the field depending on the magnetic spin quantum number of the particle. (Image from M. Puddephat [21].)

When placed in an external magnetic field, an object with a magnetic moment experiences a torque. The torque, τ, can be written as

$$\vec{\tau} = \vec{\mu} \times \vec{B_0}. \tag{2.7}$$

It is conventional to choose a coordinate system so that $\vec{B_0} = B_0\hat{k}$. Classical mechanics tells us that the torque experienced by a dipole moment is equal to the rate of change of its angular momentum, so

$$\begin{aligned} \frac{d\vec{J}}{dt} &= \vec{\mu} \times B_0\hat{k} \\ \text{or } \frac{d\vec{\mu}}{dt} &= \gamma\vec{\mu} \times B_0\hat{k} \text{ (see Equation (2.1)).} \end{aligned} \tag{2.8}$$

The solution to such a set of differential equations is

$$\begin{aligned} &\mu_{xy}(t) = \mu_{xy}(0)e^{-i\gamma B_0 t} \\ &\mu_z(t) = \mu_z(0), \\ &\text{where } \mu_{xy} = \mu_x + i\mu_y. \end{aligned} \tag{2.9}$$

Figure 2.4 illustrates Equations (2.8) and (2.9). Normally, the orientations of the nuclear magnetic moments in an object are random, but when the external field is applied to a hydrogen nucleus the magnetic moment vectors are oriented in one of two directions (parallel or anti-parallel). Equation (2.9) shows that the nucleus precesses about the z-axis, which is simliar to that of a spinning top precessing about the gravitational axis. Note that in Equation (2.9), the vector precesses at a frequency $\omega_0 = \gamma B_0$, which is known as the Larmor frequency.

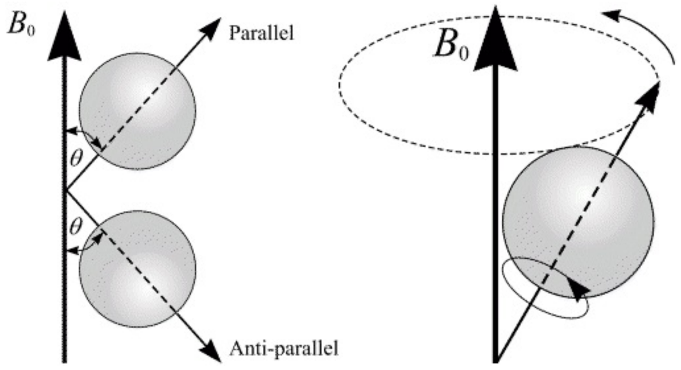


Figure 2.4: Equation (2.9) shows that the spins precess about the magnetic field axis, similar to a spinning top. (Image from M. Puddephat [21].)

2.2 Net magnetization

The net magnetization resulting from the addition of all the dipole moments can be calculated after they have been aligned with the B_0 field. The net magnetization vector $\vec{M}$ for N spins can be calculated as

$$\vec{M} = \left(\sum_{n=1}^{N} \vec{\mu_n}\right) = \left(\sum_{n=1}^{N} \mu_{x,n}\right)\hat{x} + \left(\sum_{n=1}^{N} \mu_{y,n}\right)\hat{y} + \left(\sum_{n=1}^{N} \mu_{z,n}\right)\hat{z}. \tag{2.10}$$

Since there is no preferred orientation of the dipoles in the x and y -directions, they average to zero and we are left with

$$\vec{M} = \left(\sum_{n=1}^{N} \mu_{z,n}\right)\hat{z} = \left(\sum_{n=1}^{N} \gamma\hbar(m_I)_n\right)\hat{z}, \tag{2.11}$$

$$\text{where} \qquad \sum_{n=1}^{N} (m_I)_n = \frac{1}{2}(N_\uparrow - N_\downarrow). \tag{2.12}$$

Then for hydrogen nuclei,

$$\vec{M} = \gamma\hbar \left[\frac{1}{2}(N_\uparrow - N_\downarrow)\right] \hat{z}. \tag{2.13}$$

From Equation (2.6), we make the observation that $\Delta E << k_B T$, so we can use a first order approximation to write

$$(N_\uparrow - N_\downarrow) \approx \frac{N\gamma\hbar B_0}{2k_B T}, \tag{2.14}$$

which can be substituted into Equation (2.13) to obtain

$$\vec{M} = \frac{N\gamma^2\hbar^2 B_0}{4k_B T}\hat{z}. \tag{2.15}$$

This is the net magnetization in static equilibrium, and is usually denoted $\vec{M}_0$. The magnetization is proportional to both the number of spins N and the external magnetic field strength B_0 and inversely proportional to T. Since T and N are fixed for a given biological sample, increased magnetization can only be achieved by increasing B_0. This has been the primary motivation for the increase in field strength of MRI scanners over the years.To acquire an image, we will need to perturb this spin system with additional fields.

2.3 RF pulse

To acquire an NMR signal, we need to excite the magnetization vector $\vec{M}$ so that we induce a coherent transition of spins from one energy state to another. A short pulse of time-varying magnetic field, $\vec{B}_1$, is applied perpendicular to the magnetic field $\vec{B}_0$ with $|\vec{B}_1| << |\vec{B}_0|$. The radiation energy of this field must be equal to the energy difference of the spin states, so

$$\begin{aligned} \hbar\omega_{rf} &= \Delta E = \hbar\gamma B_0, \\ \omega_{rf} &= \omega_0. \end{aligned} \tag{2.16}$$

This is called the resonance condition. Note that the pulse frequency typically lies within the radio frequency range and for this reason, $\vec{B}_1$ is called a radio frequency (RF) pulse. A typical RF pulse can be written as

$$\vec{B}_1(t) = B_1^e(t)e^{-i(\omega_0 t+\phi)}, \tag{2.17}$$

where ϕ is the initial phase angle of the RF field, and $B_1^e(t)$ is called the envelope function. Before describing the types of envelope functions used in RF pulses, we can simplify the mathematics involved by switching reference frames.

2.4 Rotating reference frame

For simplification, we can switch to the rotating reference frame with the Larmor frequency, ω_0. Mathematically, the transformation can be written as

$$\begin{pmatrix} \hat{x}' = cos(\omega_0 t)\hat{x} - sin(\omega_0 t)\hat{y} \\ \hat{y}' = sin(\omega_0 t)\hat{x} + cos(\omega_0 t)\hat{y} \\ \hat{z}' = \hat{z} \end{pmatrix} \tag{2.18}$$

Setting $\vec{M}_{1,\text{rot}} = \vec{M}$ and $\vec{B}_{1,\text{rot}} = \vec{B}_1$, the transformations of $\vec{M}$ and $\vec{B}_1$ into the rotating frame are then given by

$$\begin{bmatrix} M_x' \\ M_y' \\ M_z' \end{bmatrix} = \begin{bmatrix} cos(\omega_0 t) & -sin(\omega_0 t) & 0 \\ sin(\omega_0 t) & cos(\omega_0 t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} \tag{2.19}$$

$$\begin{bmatrix} B'_{1x} \\ B'_{1y} \end{bmatrix} = \begin{bmatrix} cos(\omega_0 t) & -sin(\omega_0 t) \\ sin(\omega_0 t) & cos(\omega_0 t) \end{bmatrix} \begin{bmatrix} B_{1x} \\ B_{1y} \end{bmatrix} \tag{2.20}$$

The motion of the bulk magnetization can be seen in Figure 2.5. In the rotating frame, the $\vec{B_1}$ field tips the magnetization vector at an angle α away from the z' axis. This angle is called the flip angle.

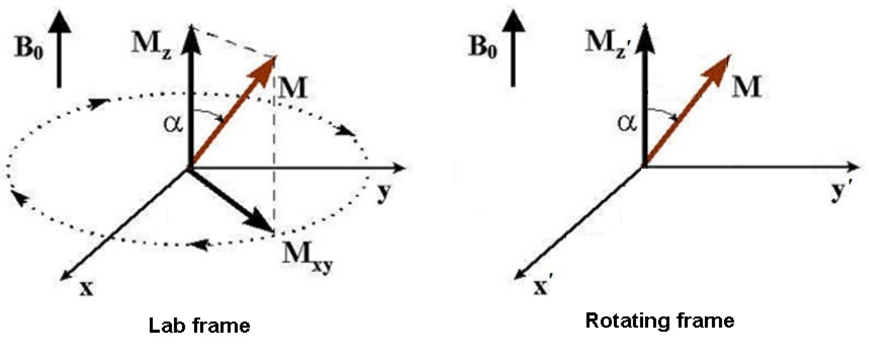


Figure 2.5: The magnetization vector M in the lab frame will precess in the xy-plane, at an angle from the z-axis given by α. In the rotating frame, M appears stationary and tipped along the y'-axis. (Image modified from [23].)

It is also of interest to define the rate of change of $\vec{M}$ with respect to the rotating frame. Taking the derivative of $\vec{M}$ gives:

$$\begin{aligned} \frac{d\vec{M}}{dt} &= \frac{dM_{x'}}{dt}\hat{x}' + \frac{dM_{y'}}{dt}\hat{y}' + \frac{dM_{z'}}{dt}\hat{z}' + M_{x'}\frac{d\hat{x}'}{dt} + M_{y'}\frac{d\hat{y}'}{dt} + M_{z'}\frac{d\hat{z}'}{dt}, \\ &= \left(\frac{d\vec{M}}{dt}\right)_{\text{rot}} + M_{x'}\frac{d\hat{x}'}{dt} + M_{y'}\frac{d\hat{y}'}{dt} + M_{z'}\frac{d\hat{z}'}{dt}, \\ &= \left(\frac{d\vec{M}}{dt}\right)_{\text{rot}} + \vec{\omega} \times \vec{M}_{\text{rot}}, \end{aligned} \tag{2.21}$$

since the time derivatives of the unit vectors are of the general form $d\hat{x}'/dt = \vec{\omega}\times\hat{x}'$. Since $\vec{M} = \vec{M}_{\text{rot}}$, we can simply write

$$\left(\frac{d\vec{M}}{dt}\right)_{\text{lab}} = \left(\frac{d\vec{M}}{dt}\right)_{\text{rot}} + \vec{\omega} \times \vec{M}. \tag{2.22}$$

2.5 Bloch equation

We can now examine the time-dependent behaviour of $\vec{M}$ in the presence of the field $\vec{B}_1$ which is described by the Bloch equation,

$$\frac{d\vec{M}}{dt} = \gamma\vec{M} \times \vec{B} - \frac{M_x\hat{x} + M_y\hat{y}}{T_2} - \frac{(M_z - M_z^0)\hat{z}}{T_1}, \tag{2.23}$$

where M_z^0 is the thermal equilibrium value for $\vec{M}$ in the presence of $\vec{B}_0$ only, and T_1 and T_2 are time constants characterizing the relaxation process of the spins after they have been perturbed from their equilibrium states. This is explained in more detail in Section 2.6. Since the RF pulse is short compared to these times, we can ignore the last two terms in Equation (2.23) for now.

Substituting Equation (2.22) into Equation (2.23) gives

$$\left(\frac{d\vec{M}}{dt}\right)_{\text{rot}} = \gamma\vec{M} \times \left(\vec{B} + \frac{\vec{\omega}}{\gamma}\right), \tag{2.24}$$

where the effective magnetic field in the rotating frame is

$$\vec{B}_{\text{eff}} = \vec{B}_{\text{rot}} + \frac{\vec{\omega}}{\gamma}. \tag{2.25}$$

In the case that $\vec{B}_{\text{rot}} = B_0\hat{z}$, and $\vec{\omega} = -\gamma B_0\hat{z}$, then Equation (2.25) gives $\vec{B}_{\text{eff}} = B_0\hat{z} - \gamma\frac{B_0\hat{z}}{\gamma} = 0$. Therefore, in the rotating frame, the apparent magnetic field vanishes and $\vec{M}_{\text{rot}}$ appears to be stationary.

2.6 Flip angle and transverse magnetization

Recall that the angle that $\vec{M}$ is tipped from the z-axis is α, the flip angle. The flip angle can be calculated by

$$\alpha = \int_0^\tau \gamma B_1^e(t)dt, \tag{2.26}$$

where τ is the duration of the pulse. Choosing a rectangular pulse for the envelope function B_1^e is common, because Equation (2.26) simplifies to

$$\alpha = \gamma B_1^e \tau, \tag{2.27}$$

so that the flip angle is easily controlled by varying the strength of the B_1 field and its duration. The amount of transverse magnetization can then be found by

$$M_{xy} = |\vec{M}|\sin(\alpha). \tag{2.28}$$

After the spin system has been perturbed from its thermal equilibrium by the RF pulse, it will return to this state provided that the external RF pulse is removed and sufficient time is allowed. Two processes occur during this time: a recovery of the longitudinal magnetization M_z, called longitudinal relaxation, and the destruction of the transverse magnetization M_{xy}, called transverse relaxation. The exact mechanisms of these processes are beyond the scope of this text, but we can describe the evolution of the transverse and longitudinal magnetizations with the Bloch equation:

$$\begin{aligned} \frac{dM_{z'}}{dt} &= -\frac{M_{z'} - M_z^0}{T_1}, \\ \frac{dM_{xy'}}{dt} &= -\frac{M_{xy'}}{T_2}, \end{aligned} \tag{2.29}$$

where the first term in the Bloch equation has been dropped since $\vec{B}_{\text{eff}} = 0$ in the Larmor frame. The solutions for these differential equations are exponential decays and are given by

$$\begin{aligned} M_{xy'}(t) &= M_{xy'}(0)e^{-t/T_2}, \\ M_{z'}(t) &= M_z^0(1 - e^{-t/T_1}) + M_{z'}(0)e^{-t/T_1}. \end{aligned} \tag{2.30}$$

Figure 2.6 graphs the longitudinal and transverse magnetizations over time. The values of T_1 and T_2 depend on the tissue composition, structure, and surroundings. In general, T_1 is always longer than T_2. Usually, T_1 ranges from 300 to 2000 ms, and T_2 ranges from 30-150 ms [18].

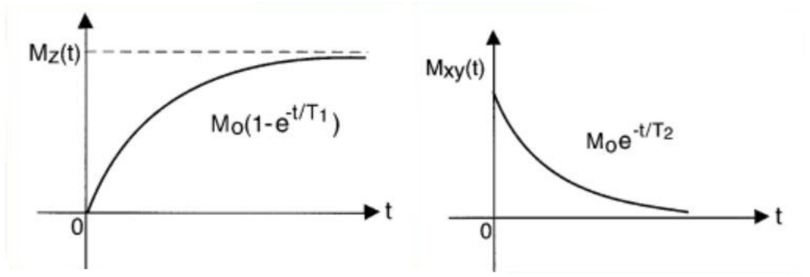


Figure 2.6: The longitudinal relaxation (left) after the magnetization vector has been tipped 90deg by an RF pulse into the xy-plane. Simultaneously, the transverse magnetization exponentially decays (right).(Image from S. Day [24].)

To summarize, we have described how placing an object (whose constituent particles have 1/2 integer nuclear spin) in an external magnetic field $\vec{B}_0$, and perturbing the system with another alternating field $\vec{B}_1$ induces a macroscopic rotating magnetization vector. This phenomenon is called NMR or nuclear magnetic resonance. In the rotating frame, the magnetization vector M is tipped at an angle α. After the B_1 field is removed, the net magnetization in the xy-plane exponentially decays with a time constant T_2. The change in magnetization will generate an RF signal which we will need to detect and interpret.

2.7 Signal detection

According to Faraday's law of electromagnetic induction, the voltage generated in a coil is proportional to the rate at which the magnetic flux through a coil changes. If $\vec{B}(\vec{r})$ is the lab frame magnetic field at location $\vec{r}$ produced by a unit direct current flowing in the detection coil, the magnetic flux can be calculated by

$$\Phi(t) = \int_{\text{object}} \vec{B}(\vec{r}) \cdot \vec{M}(\vec{r}, t) d\vec{r}. \tag{2.31}$$

By the law of induction, the voltage is then

$$V(t) = -\frac{\partial \Phi(t)}{\partial t} = -\frac{\partial}{\partial t} \int\limits_{\text{object}} \vec{B}(\vec{r}) \cdot \vec{M}(\vec{r}, t) d\vec{r}. \tag{2.32}$$

This is often regarded as the rawest form of the NMR signal. A detection scheme known as quadrature detection is commonly used in MRI systems. In this scheme, two coils are orthogonal to each other and their outputs are combined in complex form so that

$$S(t) = S_1(t) + iS_2(t). \tag{2.33}$$

In practice, the voltage detected is usually moved to a low-frequency band using phase-sensitive detection or signal demodulation. The details of this process are not given here, but may be found in [18]. An inhomogeneous field $\vec{B}(\vec{r})$ can be expressed as the summation of its homogeneous and inhomogeneous parts by $\vec{B}(\vec{r}) = \vec{B}_0 + \Delta\vec{B}(\vec{r})$, so that $\Delta\omega(\vec{r}) = \gamma \Delta B(\vec{r})$. The resulting signal from such a field can be expressed by

$$S(t) = \int\limits_{\text{object}} M_{xy}(\vec{r}, 0) e^{-i\Delta\omega(\vec{r})t} d\vec{r}. \tag{2.34}$$

Note that we assume the field inhomogeneities are not time-varying in this formulation.

By introducing the spin spectral density function $\rho(\omega)$, we can characterize the frequency distribution of a heterogeneous spin system. In terms of the bulk magnetization,

$$M = \int\limits_{-\infty}^{\infty} \rho(\omega) d\omega. \tag{2.35}$$

Our signal can then be rewritten as

$$S(t) = \sin(\alpha) \int\limits_{-\infty}^{\infty} \rho(\omega) e^{-t/T_2(\omega)} e^{-i\omega t} d\omega, \tag{2.36}$$

which is called a free induction decay (FID) signal. FID signals are the most basic form of transient signals from a spin system after a pulse excitation. The FID signal from a spin system with one spectral component resonating at a frequency ω_0 is

$$S(t) = M_z^0 \sin(\alpha) e^{-t/T_2} e^{-i\omega_0 t} \qquad t \geq 0, \tag{2.37}$$

where we see that the amplitude of the signal depends on the thermal equilibrium and flip angle values. The FID signal also has a characteristic T_2 decay, as can be seen in Figure 2.7. In the case of field inhomogeneities, T_2 is replaced by T_2^* in a manner approximated by

$$\frac{1}{T_2^*} = \frac{1}{T_2} + \frac{1}{T_2'}, \tag{2.38}$$

where T_2' takes into account any static field inhomogeneities that cause spins to dephase faster.

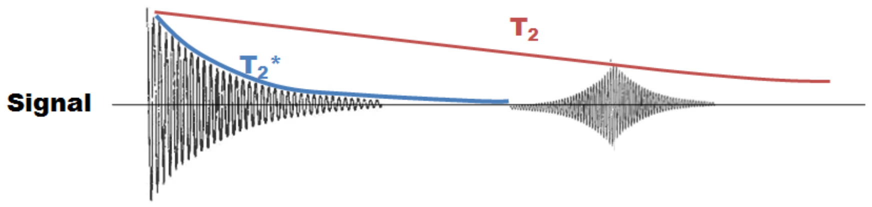


Figure 2.7: The decay of an FID signal induced by an RF-pulse. The echo pulse after the initial FID pulse is discussed in section 2.8. Note the oscillations of the signal, which can be traced back to the $e^{-i\omega_0 t}$ term in Equation (2.37).

2.8 Gradient echoes

Echoes are another type of signal used in MRI. An echo can be generated by multiple RF pulses or by magnetic field gradient reversal. The former type of echo is called an RF echo, while the latter is called a gradient echo. Either method works by refocusing the transverse magnetization after the initial FID signal to produce a second echo signal. The images used for analysis within this thesis used gradient echoes, so only they will be discussed here.

Gradient echoes require another set of coils to produce gradient fields. These fields can dephase and rephase the transverse magnetization in a controlled fashion so that multiple echo signals can be created. A gradient field $\vec{B}_G$ is defined so that the z-component of the field varies linearly along a specific direction. In most cases, the z-component of the gradient field is so strong that the other components may be ignored. For this reason, $\vec{B}_G$ and $\vec{B}_{G,z}$ are used interchangeably. $\vec{B}_G$ is called an x-gradient field if

$$B_{G,z} = G_x x, \tag{2.39}$$

where G_x is the x-gradient. We can define a y or z-gradient in a similar fashion. If we define $\vec{G} = (G_x, G_y, G_z)$, called the gradient direction of $\vec{B}_G$, we can find the value of B_G by

$$B_G = \vec{G} \cdot \vec{r}. \tag{2.40}$$

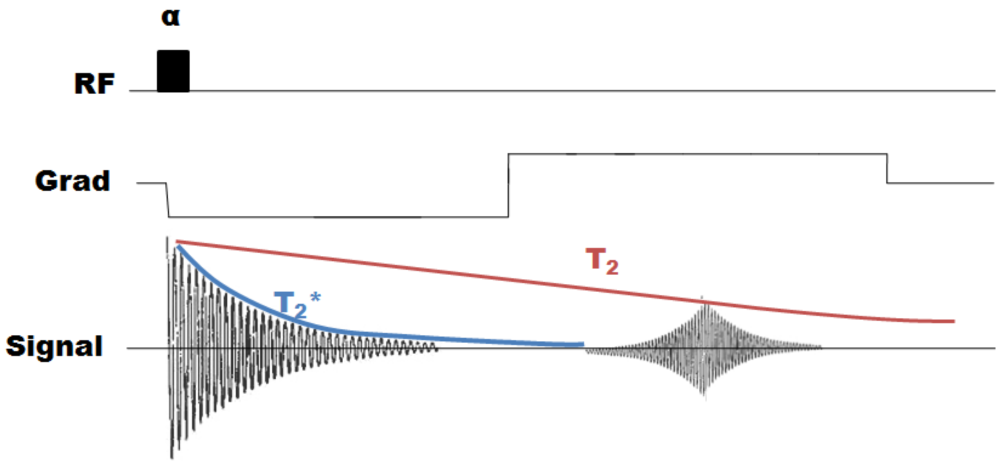


Figure 2.8: Gradient-echo pulse sequence. After an RF pulse, a negative gradient is turned on to dephase spins which are then rephased by a positive gradient.

Now, consider the spin sequence in Figure 2.8. After an initial RF pulse, a negative x-gradient is turned on. Spins in different x-positions will acquire different phases, expressed by

$$\phi(x,t) = \gamma \int_0^t -G_x x dt' = -\gamma G_x x t \qquad 0 \leq t \leq \tau, \tag{2.41}$$

where the loss of spin coherence becomes worse over time. After a time τ, the duration of the first gradient, the signal goes to zero. However, if another positive gradient of the same strength is applied, the magnetization gradually rephases which results in the regrowth of the signal, or an echo. The phase angle after the second gradient is

$$\begin{aligned}\phi(x,t) &= -\gamma G_x x t + \gamma \int_{\tau}^{t} G_x x dt' \\ &= -\gamma G_x x t + \gamma G_x x(t-\tau) \qquad \tau \le t \le 2\tau.\end{aligned} \tag{2.42}$$

so that at $t = 2\tau$, the total phase equals zero and the spin system has regained coherence.

2.9 Slice-selectivity

In MRI, gradient fields are also useful for spatially selecting one or more slices of an object to image. A slice (Figure 2.9), can be mathematically defined as

$$|\vec{\mu}_s \cdot \vec{r} - s_0| < \Delta s/2, \tag{2.43}$$

where $\vec{\mu}_s$ specifies the slice orientation, Δs is the slice thickness in the direction of $\vec{\mu}_s$, and s_0 is the distance of the slice from the origin. For example, by letting $\vec{\mu}_s = \hat{z}$, and $s_0 = z_0$, we define a slice along the z-direction. To excite a slice along the z-axis during an RF pulse, we apply a linear

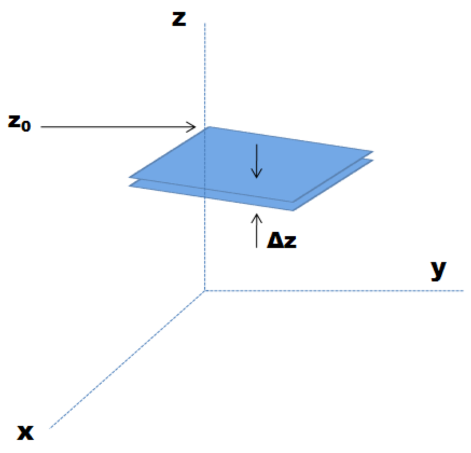


Figure 2.9: A slice oriented perpendicular to the z-axis, with thickness Δz.

gradient field along the z-axis,

$$\vec{G}_{ss} = (0, 0, G_z) = G_z \hat{z}. \tag{2.44}$$

Now, in the presence of this gradient, the Larmor frequency of a spin at position z is

$$\omega(z) = \omega_0 + \gamma G_z z, \tag{2.45}$$

and its frequency is

$$f(z) = f_0 + \gamma G_z z / 2\pi. \tag{2.46}$$

Based on the slice equations already given, we can define a spatial selection function,

$$p_s(z) = \Pi\left(\frac{z - z_0}{\Delta z}\right), \tag{2.47}$$

which is a boxcar function of width Δz centered at z_0. Its corresponding frequency would then be

$$\begin{aligned} & p_s(f) = p_s\left(\frac{2\pi f}{\gamma G_z}\right) = \Pi\left(\frac{f - f_c}{\Delta f}\right), \\ \text{where} \qquad & f_c = f_0 + \gamma G_z z_0 / 2\pi, \\ & \Delta f = \gamma G_z \Delta z / 2\pi. \end{aligned} \tag{2.48}$$

Recall that an RF pulse is characterized by an envlope function B_1^e and an excitation frequency ω_{rf} by

$$B_1(t) = B_1^e(t) e^{-i\omega_{rf} t}. \tag{2.49}$$

We can determine the proper envelope and frequency by assuming that B_1 is related to $p(f)$ by the Fourier transform. Specifically,

$$B_1(t) \propto \int_{-\infty}^{\infty} p(f) e^{-i2\pi f t} df. \tag{2.50}$$

By inserting Equation (2.47) into Equation (2.50), we can derive

$$B_1(t) \propto \Delta f \text{sinc}(\pi \Delta f t) e^{-i2\pi f_c t}. \tag{2.51}$$

We can now determine the values of ω_{rf} and B_1^e:

$$\omega_{rf} = 2\pi f_c = \omega_0 + \gamma G_z z_0, \qquad B_1^e(t) = A\text{sinc}(\pi \Delta f t), \tag{2.52}$$

where A is a constant determined by the flip angle. In practice, these pulses are not physically realizable and must be truncated, but for brevity the effects of pulse truncation are not discussed here.

2.10 Frequency and phase-encoding

After a signal has been activated by a slice-selective pulse, we can extract the spatial information from the signal during the free precession period. Two methods are used to achieve this: frequency encoding and phase encoding.

Consider a one-dimensional object with spin distribution $\rho(x)$. If the magnetic fields present are the homogeneous B_0 field and a linear gradient field $G_x x$, the Larmor frequency will be given as

$$\omega(x) = \omega_0 + \gamma G_x x. \tag{2.53}$$

The FID signal generated by the spins from an interval dx at point x will then be

$$dS(x,t) \propto \rho(x) dx e^{-i\gamma(B_0 + G_x x)t}, \tag{2.54}$$

where the proportionality constant is dependent on the flip angle α. This signal is called frequency-encoded because its frequency is linearly related to the spatial location. Therefore, G_x is called the frequency-encoding gradient. If we ignore the constant of proportionality for the moment, integrating over the entire object gives

$$S(t) = \int_{\infty}^{\infty} \rho(x) e^{-i\gamma(B_0 + G_x x)t} dx = \left(\int_{\infty}^{\infty} \rho(x) e^{-i\gamma(G_x x)t} dx \right) e^{-i\omega_0 t}. \tag{2.55}$$

We can drop the term $e^{-i\omega_0 t}$ by demodulation, and extend our argument to multiple dimensions so that

$$S(t) = \int_{\text{object}} \rho(\vec{r}) e^{-i\gamma(\vec{G}_f \cdot \vec{r})t} d\vec{r}, \tag{2.56}$$

where $G_f = (G_x, G_y, G_z)$ is the frequency encoding gradient.

Consider the one-dimesional case again, but this time we turn on a gradient G_x for only a short interval T_{pe} and then turn it off. The local signal is given as

$$dS(x,t) = \begin{cases} \rho(x) e^{-i\gamma(B_0 + G_x x)t} dx & 0 \leq t \leq T_{pe} \\ \rho(x) e^{-i\gamma G_x x T_{pe}} e^{-i\gamma B_0 t} dx & T_{pe} \leq t \end{cases} \tag{2.57}$$

During the first interval, the local signal is frequency encoded so that spins at different x positions have different frequencies. If the gradient is turned off at T_{pe}, these spins will again precess at the same rate, but will have acquired different phase angles

$$\phi(x) = -\gamma G_x x T_{pe}. \tag{2.58}$$

It is now clear that the signal is phase-encoded. Similar to frequency encoded signals, the one-dimensional argument can be generalized and demodulated so that the final signal can be expressed by

$$S(t) = \int_{\text{object}} \rho(\vec{r}) e^{-i\gamma \vec{G}_p \cdot \vec{r} T_{pe}} d\vec{r}. \tag{2.59}$$

2.11 k-space

From Equation (2.56), we can show the Fourier transform relationship between $S(t)$ and $\rho(x)$ if we write

$$S(\vec{k}) = \int_{-\infty}^{\infty} \rho(\vec{r}) e^{-i2\pi\vec{k}\cdot\vec{r}} d\vec{r}, \tag{2.60}$$

where $\vec{k} = \gamma\vec{G}_f t/2\pi$. In two dimensions, we may rewrite this as

$$S(k_x, k_y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \rho(x, y) e^{-i2\pi(k_x x + k_y y)} dx dy. \tag{2.61}$$

Note that $S(k)$ is only available for a limited set of points in k-space. The set of these points is called the sampling trajectory of k-space. The trajectory can be found by

$$\begin{aligned} &k_x = \gamma G_x t/2\pi, \\ &k_y = \gamma G_y t/2\pi, \\ &\text{or} \\ &k_x = \gamma G_x (t - T_E)/2\pi, \\ &k_y = \gamma G_y (t - T_E)/2\pi, \end{aligned} \tag{2.62}$$

depending on whether the signal is an FID or echo signal, respectively. The echo time, T_E, is the time between the echo pulse and the initial FID signal. We can also define the angle ϕ from the origin at which we would like to sample k-space (see Figure 2.10):

$$\phi = \tan^{-1}\left(\frac{G_y}{G_x}\right). \tag{2.63}$$

This can easily be generalized to three dimensions using G_x, G_y, and G_z to define the spherical azimuthal and zenith angles. Further pulse sequences at different gradient strengths can then provide a more complete coverage of k-space, allowing us to reconstruct our image. Although there are many strategies to map k-space completely, they are not discussed in detail here. Figure 2.11

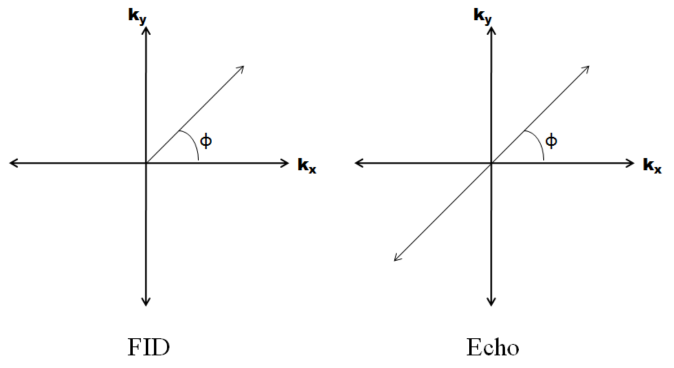


Figure 2.10: The trajectory at which k-space is sampled at angle ϕ for an FID and echo signal. The two-sidedness of the echo signal doubles the sampling size of k-space relative to the FID.

combines the concepts introduced so far, and shows a complete gradient-echo imaging sequence to obtain a signal that can be converted into a 2D image. It should be understood that the sequence is taken multiple times with different values of G_y, to obtain a full coverage of k-space, while G_z and G_x remain constant for slice and frequency selectivity, respectively.

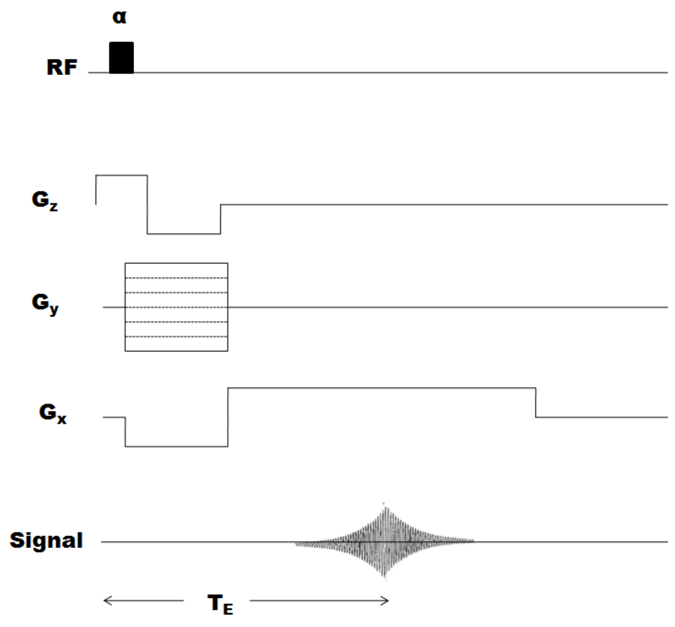


Figure 2.11: Generic gradient-echo imaging sequence. G_z is used for slice-selectivity. G_y is varied for each iteration of the sequence to map different k-space trajectories. G_x is used for frequency encoding. T_E is the echo time. The time between successive sequences T_R is such that $T_R >> T_E$.

2.12 Image Reconstruction

Ultimately, our goal is to acquire a 2D-image slice of a three-dimensional object. To this end, we can denote our image function by $I(x, y)$ and relate it to the underlying function $\rho(x, y, z)$ by

$$I(x,y) = \int_{z_0-\Delta z/2}^{z_0+\Delta z/2} \rho(x,y,z)dz. \tag{2.64}$$

The basic imaging equation is

$$S(k_x, k_y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} I(x,y)e^{-i2\pi(k_x x+k_y y)}dxdy. \tag{2.65}$$

A scheme that uses n sequences to cover k-space could be defined by

$$\begin{aligned} G_{n,x} &= G\cos\phi_n, \\ G_{n,y} &= G\sin\phi_n, \end{aligned} \tag{2.66}$$

where ϕ is defined by Equation (2.63). The acquired signal for the n-th cycle is defined as

$$S_n(t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} I(x,y)e^{-i\gamma Gt(x\cos\phi_n+y\sin\phi_n)}dxdy. \tag{2.67}$$

Image reconstruction methods such as a Fast Fourier Transform (FFT) can then solve Equation (2.67) for $I(x, y)$. The techniques of this section can be applied successively along the slice-selective axis to yield a stack of 2D images, forming a 3D image.

2.13 Velocity Encoding and Flow Imaging

Most MR-sequences demonstrate sensitivity to flow and motion, which can lead to artifacts in many applications [25]. However, this motion sensitivity can also be manipulated to quantify blood flow and motion of tissue. The quantification of flow relies on the fact that the local spin magnetization is a vector quantity. Using appropriate velocity encoding gradients, motion dependent phase effects can be extracted from the signal. Velocity gradients can be encoded in all three directional components to acquire three separate spin phase signals that can be used to quantify motion in each direction.

The motion dependency of spin phase signals can be derived by examining the precession frequency of spins in a local magnetic field. The Larmor frequency ω_L of a spin at a spatial location $\vec{r}$ in a static magnetic field B_0 with a local field inhomogeneity ΔB_0 and a magnetic field gradient $\vec{G}$ is given by

$$\begin{aligned} \omega_L(\vec{r}, t) &= \gamma B_z(\vec{r}, t) \\ &= \gamma B_0 + \gamma \Delta B_0 + \gamma \vec{r}(t) \cdot \vec{G}(t), \end{aligned} \tag{2.68}$$

where $\gamma B_0 = \omega_{L,0}$. The acquired FID signal is demodulated with respect to the Larmor frequency in the static field B_0 so that the static field contribution can be ignored (see Equations (2.56) and (2.59)). Integrating Equation (2.68) gives the phase of the precessing magnetization and thus the phase of the measured MR signal at echo time $t = T_E$ after an excitation pulse at $t = t_0$:

$$\begin{aligned} \phi(\vec{r}, T_E) - \phi(\vec{r}, t_0) &= \int_{t_0}^{T_E} \omega_L(\vec{r}, t)\, \mathrm{d}t, \\ &= \gamma \Delta B_0 (T_E - t_0) + \gamma \int_{t_0}^{T_E} \vec{G}(t) \vec{r}(t)\, \mathrm{d}t. \end{aligned} \tag{2.69}$$

This can be further expanded in the following Taylor series:

$$\begin{aligned} \phi(\vec{r}, T_E) &= \phi(\vec{r}, t_0) + \gamma \Delta B_0 (T_E - t_0) + \sum_{n=0}^{\infty} \phi_n(\vec{r}^{(n)}, T_E), \\ &= \phi_0 + \sum_{n=0}^{\infty} \frac{\gamma}{n!} \int_{t_0}^{T_E} \vec{r}^{(n)} \vec{G}(t)(t - t_0)^n, \mathrm{d}t. \end{aligned} \tag{2.70}$$

Here, $\vec{r}^{(n)}$ is the n^{th} derivative of the time dependent spin position and ϕ_n the corresponding

n^{th} order phase. If the change of flow is small with respect to the temporal resolution of data acquisition the velocities can be approximated to be constant during data acquisition. Then $\vec{r}(t)$ can be approximated as a first order displacement $\vec{r}(t) = \vec{r_0} + \vec{v}(t - t_0)$ with a constant velocity $\vec{v} = \vec{v}(\vec{r_0})$. Equation (2.70) then becomes

$$\phi(\vec{r}, T_E) = \phi_0 + \gamma \vec{r_0} \int_0^{T_E} \vec{G}(t)\,\mathrm{d}t + \gamma \vec{v} \int_0^{T_E} \vec{G}(t) t\,\mathrm{d}t, \tag{2.71}$$

where the three terms correspond to an unknown background phase ϕ_0, and the effects of the gradient field on static and moving spins, respectively. The integrals in Equation (2.71) are also known as n^{th} order gradient moments M_n so that

$$\phi(\vec{r}, TE) = \phi_0 + \gamma \vec{r_0} M_0 + \gamma \vec{v} M_1, \tag{2.72}$$

where M_1 determines the velocity induced signal phase. As a result, appropriate control of the first gradient moment can be used to specifically encode spin flow or motion.

A bipolar gradient signal can be used to eliminate M_0 from Equations (2.71) and (2.72). Equation (2.71) then gives

$$\phi = \phi_0 + \gamma G v \left(\frac{T}{2}\right)^2, \tag{2.73}$$

where G is the gradient amplitude and T is the gradient duration. It is now easily seen that velocity induced phase shifts can be controlled by adjusting either of these two parameters.

To remove background phase effects ϕ_0, two measurements with different first moments ($M_1^{(1)}$ and $M_1^{(2)}$) are necessary. The simplest case would be to then use two bipolar gradient signals with inverted polarity (shown in Figure 2.12). These moments can be subtracted from each other to result in a first gradient moment $\Delta M = M_1^{(1)} - M_1^{(2)}$ which is proportional to v. By using the gradient signals described above, Equations (2.72) and (2.73) give

$$\begin{aligned} \Delta\phi &= \gamma v \Delta M_1, \\ v &= \frac{\Delta\phi}{\gamma \Delta M_1}. \end{aligned} \tag{2.74}$$

Figure 2.12: Bipolar gradient signals with inverted polarites are used to eliminate M_0 and M_1 from Equation (2.72).

Note that any phase shift that exceeds π will result in phase wrapping. Therefore, some prior knowledge of the maximum velocities is required. A velocity sensitivity v_{enc} is defined so that $v_{enc} = \pi/\gamma\Delta M_1$. ΔM_1 should then be adjusted so that v_{enc} corresponds to the highest expected velocity. Only velocities along the direction of the gradient contribute to the MR-signal, so multiple measurements are needed for a complete three-directional velocity dataset. Figure 2.13 shows the pulse sequence for one such measurement

Noise in a phase contrast image is defined as the standard deviation σ_ϕ of the phase differences in a homogeneous region with no flow. It can be shown that this is inversely related to the signal-to-noise ratio (SNR) in the corresponding magnitude images [25]. This noise can be estimated by

$$\sigma_\phi = \frac{\sqrt{2}}{\pi(SNR)} v_{enc}. \tag{2.75}$$

Therefore, v_{enc} should be chosen as small as possible to minimize noise in the phase images.

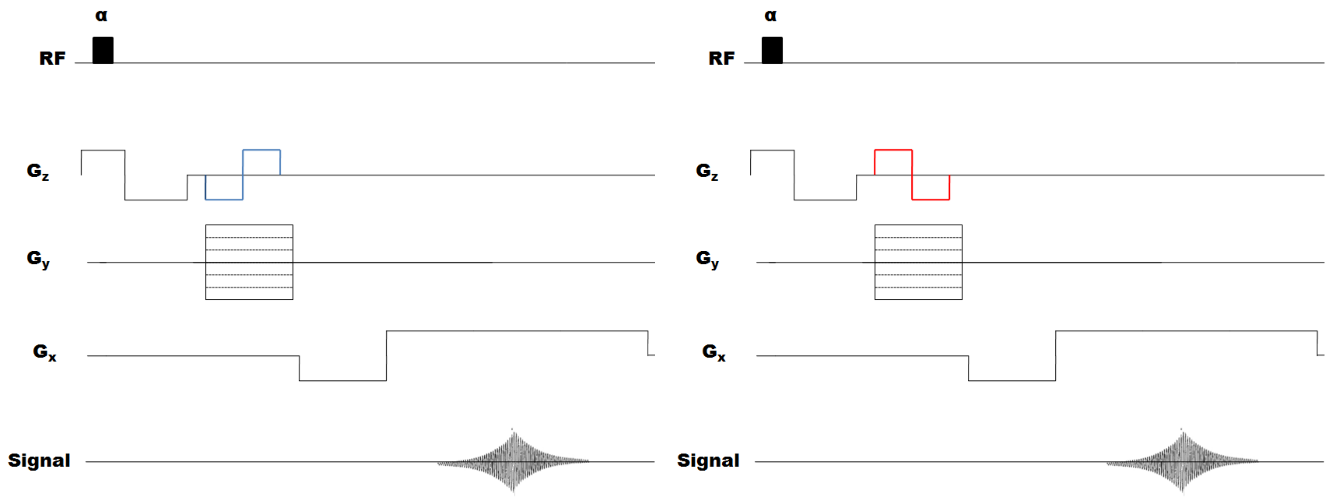


Figure 2.13: Pulse sequence for velocity encoding. After the blue pulse, the net magnetization is $M_1^{(1)}$. The red pulse gives us a net magnetization of $M_1^{(2)}$. Subtraction of these terms leaves us with a term proportional to the velocity of moving spins.

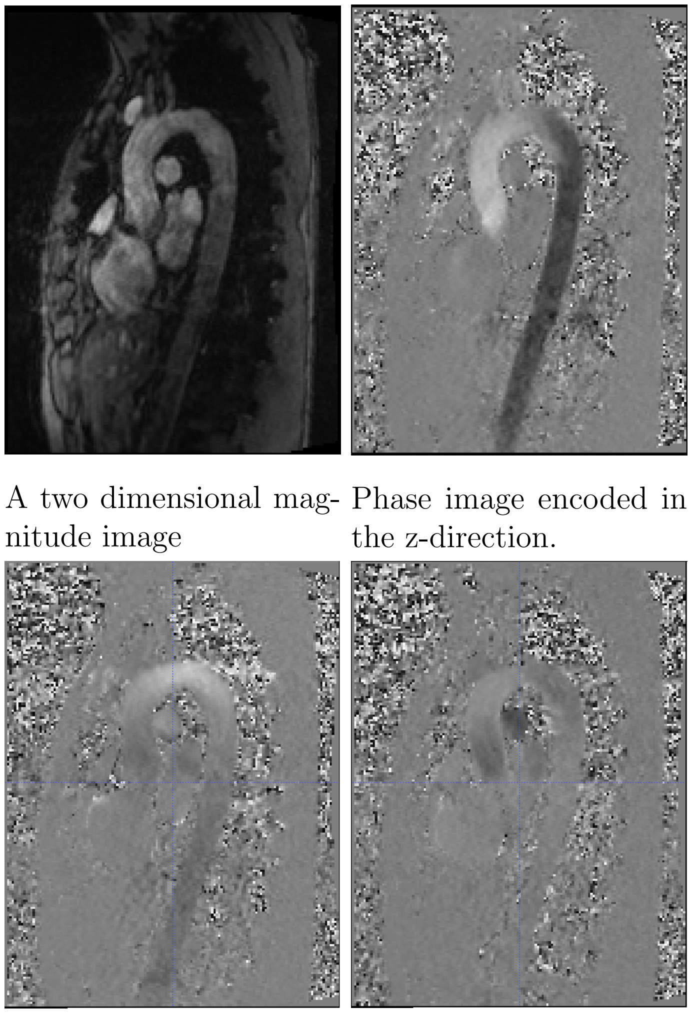
2.14 MR Acquisitions

The work on image segmentation presented in this thesis was done using retrospective data for post analysis. The MRI 4D in vivo flow data were acquired using a sagittal oblique 3D volume covering the entire thoracic aorta. Three-dimensional MRI volume scans with three-directional flow encodings were implemented on a 3.0 T MRI system (Trio, Siemens Healthcare, Malvern, PA USA) with ECG signal triggering and free breathing. A segmented spoiled gradient-echo flow sequence with three-directional encodings was performed with echo time (TE) of 2.5 ms, repetition time (TR) of 5.5 ms, flip angle of 10 degree, temporal resolution of 40ms, spatial resolution of $1.7 \times 3.3 \times 1.7\text{mm}^3$, and a velocity sensitivity of 150 cm/s along three directions.

2.15 Magnitude and Phase-contrast Images

We have shown in Section 2.12 that a Fourier transform can transform our measured k-space data into image space. In Equation (2.67), the image $I(x, y)$ is complex, so we have the option of manipulating this data in different ways. A magnitude image result from combining the real and imaginary parts in quadrature.

While the magnitude image is useful in brightly displaying objects of a certain tissue composition, we have also seen in Section 2.13 that a phase-image is useful for quantifying movement and flow. Each pixel in the complex image has a phase which may be calculated by $\tan^{-1}(\mathrm{Im}/\mathrm{Re})$, where Re and Im refer to the real and imaginary parts of the complex image. Figure 2.14 shows an example of a magnitude image and its corresponding phase images. Recall from Section 2.13 that the phase-contrast images are encoded in three directions, yielding three separate images.



A two dimensional magnitude image

Phase image encoded in the z-direction.

Phase image encoded in the x-direction.

Phase image encoded in the y-direction.

Figure 2.14: A 2D sagittal slice of a magnitude image with its corresponding phase images. This image is taken during the point in the heart cycle where blood is flowing fastest. In the phase images, a bright spot indicates positive flow in the direction it is encoded, while dark indicates negative.

Chapter 3

Image Analysis Methods and Techniques

This chapter introduces the five segmentation algorithms we use in this thesis. The first algorithm uses a level set, a powerful tool used in image segmentation, on the magnitude image. A description of the level set parameters and how they control the evolving level set contour is given. We also introduce various preprocessing techniques for the magnitude image that allows the application of the level set to segment the aorta. Next we introduce three segmentation algorithms to be used on phase-contrast images, which work by analyzing blood flow velocity patterns at voxels within the image and determining whether those corresponding voxels are likely to be inside or outside the aorta. These phase segmentations are further refined by de-noising techniques such as Gaussian filtering. The level set is then used as a means to extract the aorta from any surrounding artefacts that have been picked up by the initial segmentation. Finally, in the last segmentation algorithm we combine the level set and phase-contrast techniques by initializaing a level set on the magnitude image with the phase-based segmentation results. We discuss some post-processing calculations and applications of segmentation including velocity interpolation, the definition of analysis planes, and the visualization of flow data. Images were displayed using ITK-SNAP, an image processing application built upon the open source Insight Toolkit (ITK) [26]. This module has some built in image-processing features which were useful for creating the images of this and later chapters.

3.1 Level Sets

The level set method was devised by S. Osher and J. A. Sethian [27] as a versatile way for computing and analyzing the evolution of a curve Γ which bounds a region of interest Ω in two or more dimensions. We wish to compute the motion of Γ under a velocity field v, which can depend on position, time, the curvature of the curve itself, or any other external physics. To this end, we define the level set function so that

$$\Phi(\vec{x}, t) > 0 \text{ for } \vec{x} \in \Omega,$$

$$\Phi(\vec{x}, t) < 0 \text{ for } \vec{x} \notin \Omega,$$

$$\Phi(\vec{x}, t) = 0 \text{ for } \vec{x} \in \partial\Omega = \Gamma(t),$$

where $\vec{x}$ is a n-dimensional vector. Since we have defined Φ such that $\Gamma(t)$ is the set where $\Phi(\vec{x}, t) = 0$, we have reduced the segmentation problem to locating the set $\Gamma(t)$ for which Φ vanishes. The equation of motion is then

$$\frac{\partial \Phi}{\partial t} + \vec{v} \cdot \nabla \Phi = 0, \tag{3.1}$$

which can be rewritten as

$$\frac{\partial \Phi}{\partial t} + C(\vec{N})|\nabla \Phi| = 0, \tag{3.2}$$

where $\vec{N}$ is the normal vector, $\vec{N} = \nabla\Phi/|\nabla\Phi|$, and C is a function containing features of the image that will determine the speed at which the curve will evolve.

The advantage of the level set method of segmentation is that arbitrarily complex shapes can be modeled and topological changes such as merging and splitting are handled implicitly. The level-set equations need be solved only at voxels near the boundary Γ, so the computation time will be proportional to the size of the surface (in 3 dimensions). This can result in slow segmentations in four-dimensional images where datasets are very large since the number of voxels on a surface increases with image resolution. Slow segmentations can also result from poor placement of the initial contour, so that many iterations of the level set are needed. However, since all points on the evolving contour are described by the same equation, the level set method is also highly parallelizable on a Central Processing Unit (CPU) or Graphics Processing Unit (GPU), which can significantly reduce computation times [28]. This will be further discussed in Chapter 4.

The level set used in the following segmentations is a Geodesic Active Contour (GAC) model [29], taken from the Insight Toolkit [30], which defines C as

$$C = ((\alpha - \beta\kappa)f - \gamma(\nabla f \cdot \vec{N}))\vec{N}, \tag{3.3}$$

where f is the feature image, and κ is the mean curvature. Here, α, β and γ are scalar constants which can be used to weight the influence of the propagation, curvature and advection terms, respectively. In our case, the feature image will be the magnitude of the gradient at each point in the magnitude image. The idea is that the edges of structures in an image will vary greatly in contrast with the background of the image. Therefore, a gradient magnitude image has high image intensity at structural edges, and low intensities elsewhere. Using this as a feature image will allow us to slow contour growth near image edges by equating the speed of the contour with the intensity of the gradient. In general, level set convergence is not guaranteed, and the contour may expand beyond image edges if they have either been poorly defined in the feature image or the algorithm runs for too long. This is discussed in further detail in section 3.2. In Chapter 5, we will discuss how the scalar constants in Equation (3.3) are chosen for each use of the level set.

In the following sections, we will discuss three different segmentation methods: First, the level set is used on a single time slice of the magnitude image to illustrate the weakness of relying on only magnitude information. Second, the phase data are analyzed over time to rule out voxels that are noisy, have no signal, or have a blood flow velocity in a direction not consistent with the aorta shape. Flow curves are also fit to Gaussian waveforms and segmented, based on the fitting parameter values. These initial segmentations are refined by noise reduction techniques and extracted from surrounding erroneously segmented vessels with the GAC level set. Lastly, the results from the phase segmentations will be used to provide an initial contour for a level set used on the magnitude image, to incorporate both phase and magnitude information into the segmentation. A summary of this procedure is shown in in Figure 3.1.

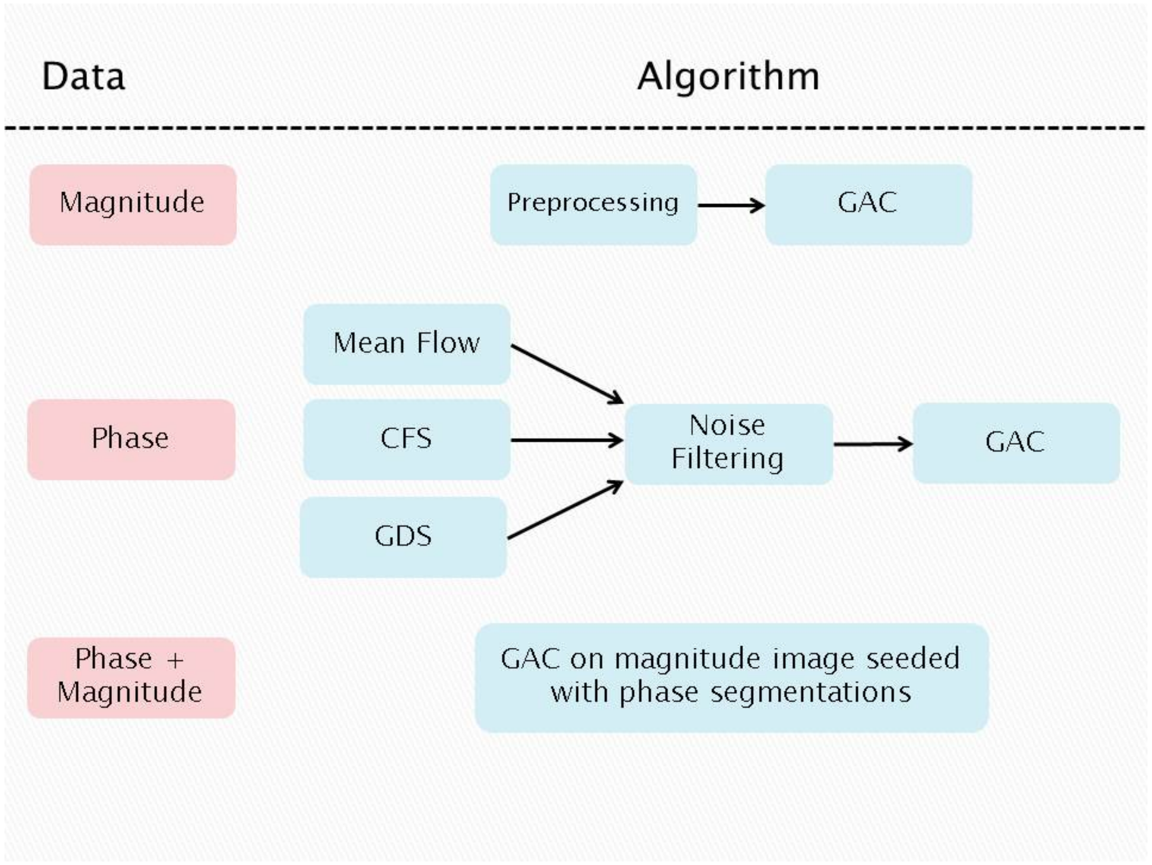


Figure 3.1: A flowchart which overviews the segmentation algorithms to be discussed in more detail later in this Chapter.

3.2 Magnitude Segmentation

For a segmentation of the magnitude image, we choose from our 4D data the 3D time slice that has the best contrast (which corresponds to the time at which average flow speed is the largest). Some 2D slices of the magnitude image can be seen in Figure 3.2. It has been shown that changes in aortic dimensions over time are at scales below our image resolution, so a 3D segmentation taken from the clearest data slice should in theory be valid throughout the cardiac cycle [31].

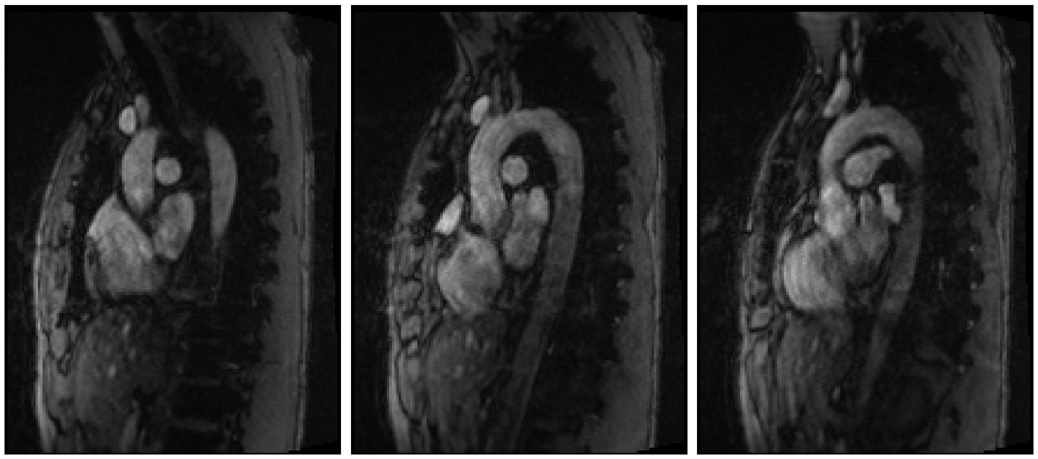


Figure 3.2: Three 2D sagittal slices of the magnitude image used. The goal is to segment the aorta, the large hook-shaped vessel in the center of the images.

Before applying the GAC level set method discussed above, a few pre-processing steps were taken. The magnitude image is first smoothed using a Curvature Anisotropic Diffusion filter, which makes use of a modified curvature diffusion equation (MCDE). This technique was developed by Perona and Malik as a way to smooth an image while preserving edges [32]. The MCDE equation for a two-dimensional image is given as the evolution equation:

$$f_t = \mid \nabla f \mid \nabla \cdot \left[c(\mid \nabla f \mid) \frac{\nabla f}{\mid \nabla f \mid} \right], \tag{3.4}$$

where $f = f(x, y, t)$ and $f(x, y, 0)$ is the image $I(x, y)$. Progressively smoothed images are obtained by choosing greater values of t from the solution. The conductance function c is defined as

$$c(|\nabla f|) = k^2/(k^2 + |\nabla f|^2). \tag{3.5}$$

The filter is sensitive to a conductance parameter k, which is analogous to the width of a Gaussian filter, and also the number of iterations to be computed. The intensity values of the aorta within the image are non-homogeneous which will result in the detection of false edges if passed to an edge-detecting filter before smoothing. Generally, a higher k value is appropriate for lower contrast images, since this will widen the filter and smooth over larger areas.

After smoothing, the gradient magnitude image is calculated with an Infinite Impulse Response (IIR) filter which approximates the convolution of the image with the derivative of the Gaussian kernel [33, 34]. The Gaussian kernel width σ can be increased, which can help eliminate any false edges picked up by the MCDE. The gradient image intensities are then rescaled with a Sigmoid transform to better define edges [26]. In terms of voxel intensity, the Sigmoid transform is defined as

$$I' = (\text{Max - Min}) \cdot \frac{1}{1 + e^{-(\frac{I-\beta_s}{\alpha_s})}} + \text{Min}, \tag{3.6}$$

where I is the input voxel intensity, I' the output voxel intensity, α_s defines the width of the input intensity range, β_s defines the intensity around which the range is centred, and max and min refer to the maximum and minimum values of the output image. Examples of the gradient magnitude and sigmoid filters can be seen in Figures 3.3 and 3.4.

Due to the high average intensity in the upper aorta versus the low average intensity in the lower aorta, we found that applying the sigmoid separately to the two halves of the image allowed for better edge distinction. Well defined edges are important because they define our feature image in Equation (3.3) and slow the propagation of the level set. The two sigmoid images are passed as inputs for two GAC level sets, whose outputs are recombined to give us our final segmentation. Figure 3.5 shows an example of the evolution of a level set using the sigmoid as a feature image,

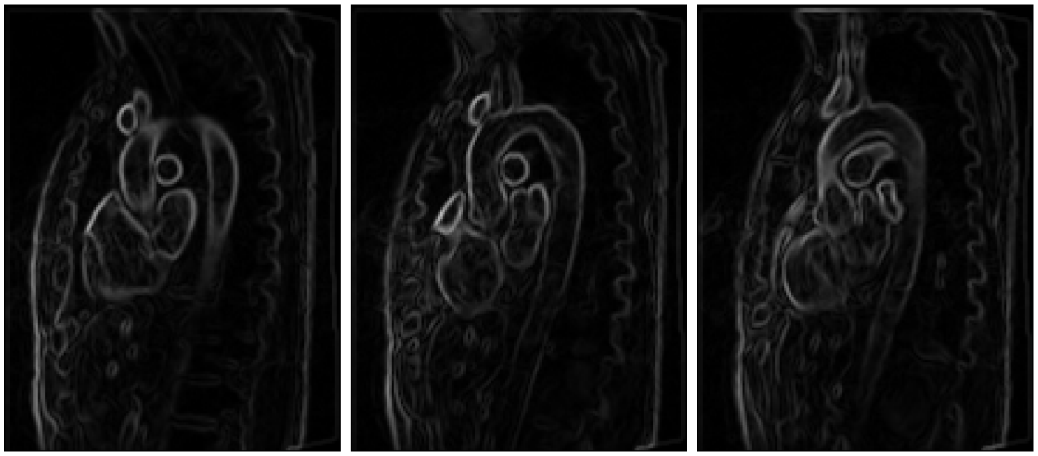


Figure 3.3: Some sagittal 2D slices of the gradient magnitude image. Notice the large difference in intensities between upper and lower aorta. A sigmoid filter can only be made sensitive to the intensity ranges of the upper or lower aorta, but not both. Some blurring of image edges can also be seen. We overestimate edges, so that when the level set is applied it does not escape the aorta and engulf the image.

and demonstrates the importance of good edge detection.

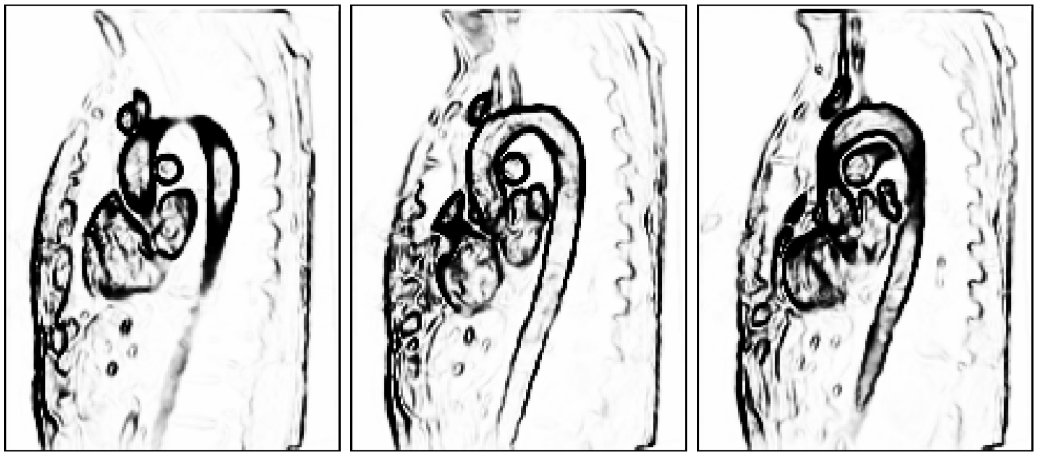


Figure 3.4: Some sagittal 2D slices of the sigmoid output. The outermost slices along the y-axis (Figures 3.4a and 3.4c) have heavily blurred edges, but the central image shows good edge resolution. This is a result of the relatively poor resolution along the y-axis. (See Section 2.14). Notice that the intensity relative to the gradient image is inverted as a result of choosing a negative value for α_s.

Figure 3.6 shows an example of the full pre-processing pipeline that is performed on a magnitude image before a segmentation can take place. Notice that in Figures 3.4, 3.5, and 3.6, changes in Sigmoid parameters greatly effect the edge detection of the lower or abdominal aorta but the edges

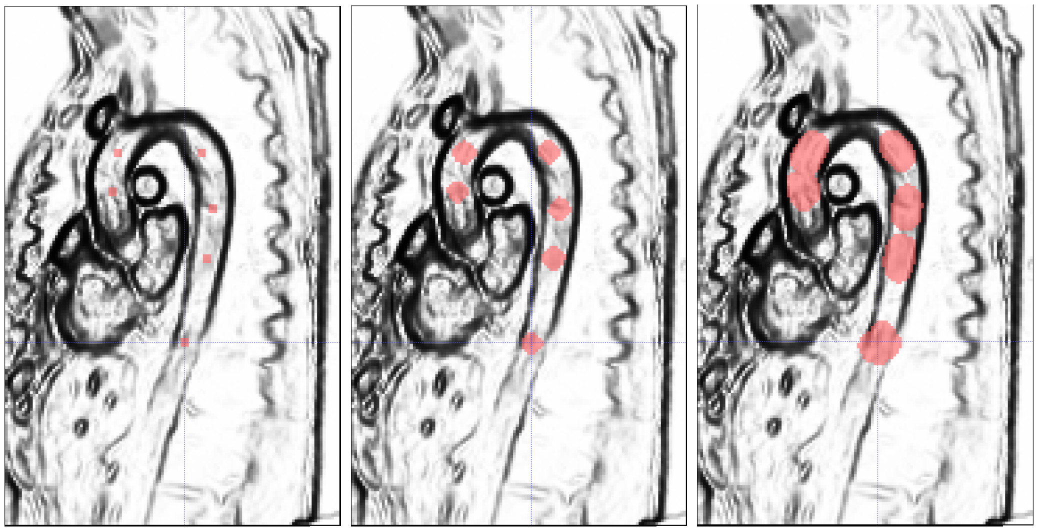


Figure 3.5: Example of the evolution of a level set on a sub-optimal sigmoid image. The sigmoid is passed as the feature image to the GAC algorithm. Initially, it is seeded with user-defined points. The middle image shows the level set after 30 iterations of the algorithm. The final image shows the level set after 50 iterations. Notice that in the lower descending aorta, because the edges are poorly defined, the contour begins to escape the aortic walls, while in the ascending aorta the contour is wrapped to the stronger defined image edges.

detected in the upper aorta do not vary nearly as much. This is due to the poor image contrast in the lower part of the original image. Results of the magnitude segmentation are presented and discussed in Chapter 5.

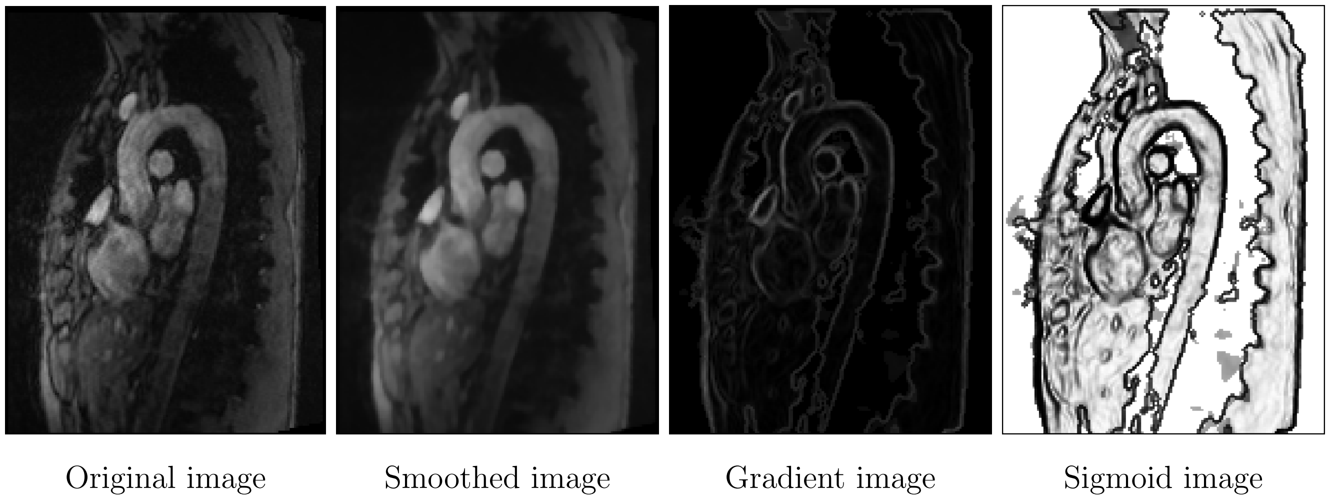


Figure 3.6: Pre-processing pipeline for the magnitude image. The original image is smoothed first to obtain a more homogeneous intensity across the object. This allows the gradient image to detect edges optimally, and the Sigmoid simply transforms the intensities.

3.3 Phase Segmentation

Poor image contrast in the magnitude images can sometimes lead to poor edge detection, so that the level set contour expands beyond the vessel walls. For most accurate results, some additional information is needed for segmentation. This suggests a method which utilizes flow data or phase-contrast imaging. We propose 3 methods which analyze flow data over time at each 3D voxel location. These will be discussed here and in the next section.

All three segmentation algorithms make use of phase images encoded for flow in 3 different directions, so it is necessary to define some coordinate system in which to work. Figure 3.7 defines the coordinate system used in this thesis. Some 2D slices of a phase image encoded in the z-direction can be seen in Figure 3.8 for reference. The voxel intensity, which is proportional to flow speed, can be plotted in each phase-encoded direction as a function of time to obtain three intensity curves at each 3D location. We expect well-defined curves in voxels which are inside the aorta, and noise or no signal in voxels that are not. The aorta is oriented in such a way that large flow velocities are constrained to the x- and z-directions, and peak velocities in the y-direction will be comparatively small. For this reason, we focus primarily on the phase data encoded in the x- and z-directions. Figures 3.9a - 3.9d show typical flow curves for voxels within the aortic arch, lower aorta and outside the aorta, respectively. With limited flow in the y-direction, we only expect to measure curves like Figures 3.9a and 3.9b or a combination of both (with flow in the z-direction switching sign in the ascending aorta).

It should be noted that in the case of an unhealthy patient with turbulent flow, these assumptions may not hold. In this case, peak velocities will also sometimes lie in the $\pm y$ directions. The algorithm described below could be extended to also search for flow in these directions. However, further testing of these algorithms on new data sets are needed before any further discussions on this subject can be made.

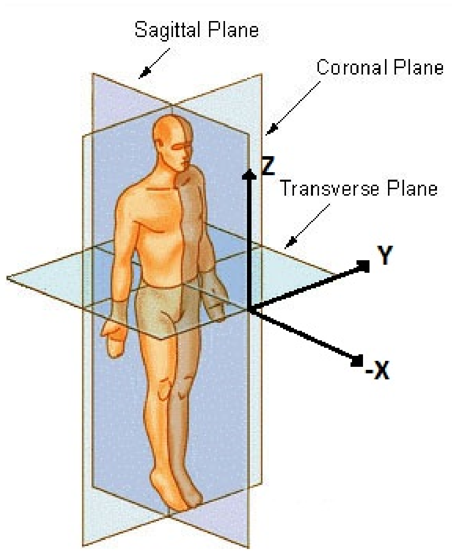


Figure 3.7: Coordinate system used for flow analysis. (Picture modified from a public domain image of the body planes).

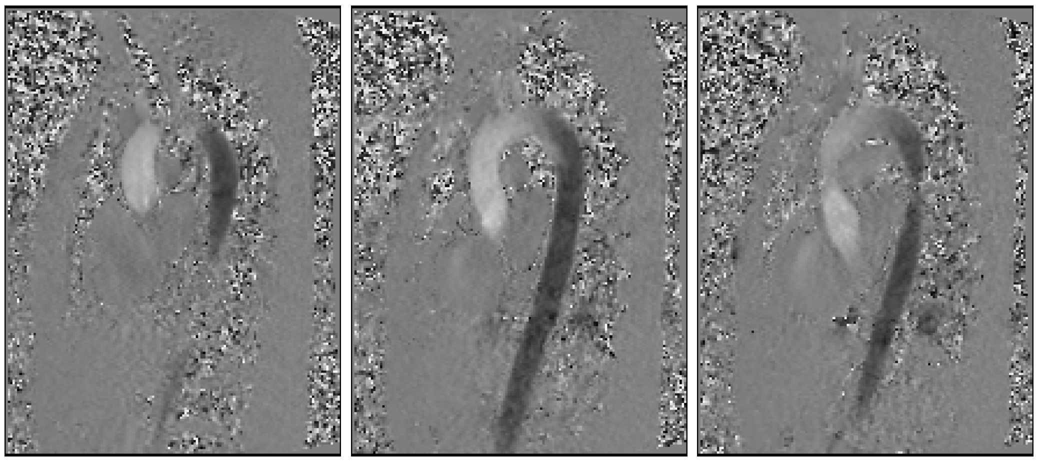
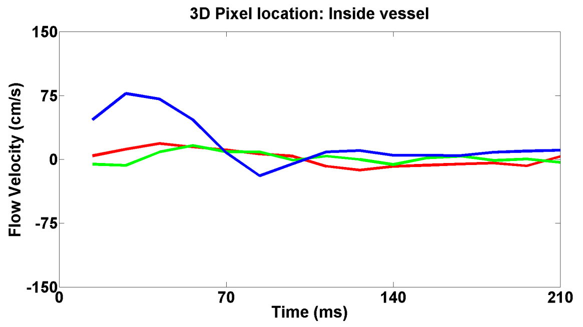
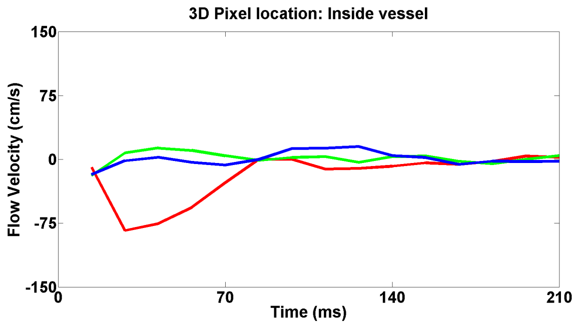


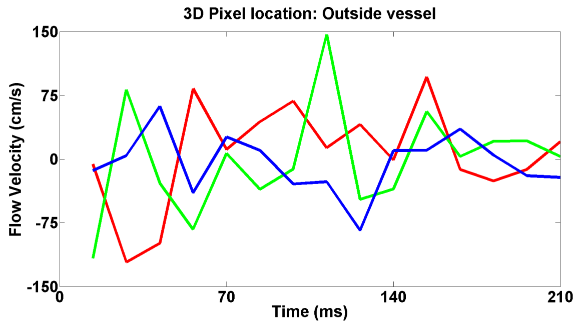
Figure 3.8: Three 2D sagittal slices of a phase image encoded in the z-direction. Intensity of voxels denotes direction and magnitude of flow speed. A linear relationship exists between the pixel intensity value and the flow speed. Darker pixels indicate downwards flow, while bright pixels indicate upwards flow.



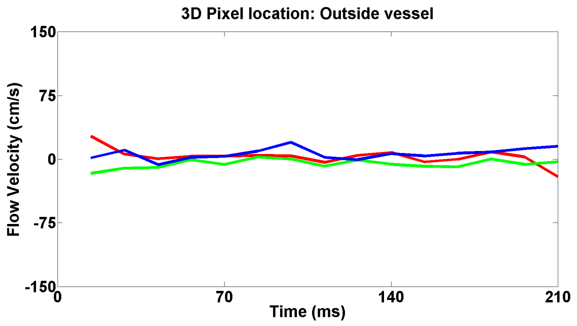
a) Aortic arch



b) Lower aorta



c) Outside aorta



d) Outside aorta

Figure 3.9: The curves show the typical directional speed of blood flow over time within voxels, depending on their location in the image. x-direction: blue, y-direction: green, z-direction: red. In a), the aortic arch is oriented with respect to our coordinate system so that we can expect large x-velocities here during systole. This can be seen clearly as the blue curve spikes in the first 70 ms of the heart cycle. A similar argument can be made for the lower aorta, where we expect blood to flow in the $-z$-direction. c) and d) show noise and no signal, respectively. We can identify whether a voxel belongs in our segmentation by comparing its flow curves to the patterns we expect to find.

In the first segmentation method, which we simply call the phase-segmentation, we can classify a 'good' voxel from a noisy or no signal voxel (Figures 3.9c and 3.9d) by examining the mean and standard deviation of flow velocity for various data regions on the curve. For example, if a voxel has a large mean or deviation of intensity on the tail of its flow curve, we can confidently eliminate it from the segmentation because it does not resemble Figures 3.9a and 3.9b, or any combination thereof. Similarly, we can eliminate any voxel that displays large mean flow speeds in the y-direction at any point. Figure 3.10 depicts the data regions used in calculations. Mean (m) and standard deviation (σ) values are labeled with the subscripts es (early systole), s (systole), and d (diastole) depending on the portion of the heart cycle we wish to examine. Calculations done on the entire curve will use the subscript c (curve).

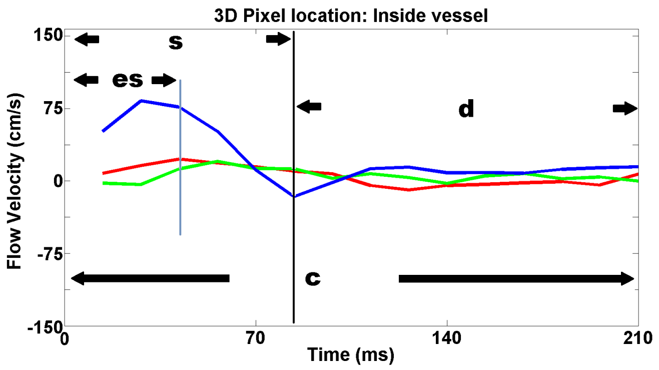


Figure 3.10: Regions of interest within a flow curve. Mean and standard deviations can be calculated for each region depicted here, as well as on the entire curve. These calculations can then be used to form inclusion criteria for the segmentation algorithm.

By creating a range of acceptable mean and standard deviation values, we can narrow the criteria for a voxel to be accepted into the segmentation. This ensures that flow curves resemble the shapes we see in Figures 3.9a and 3.9b. Every range of acceptable values can be estimated based on the v_{enc} and the predominant direction of flow, and then adjusted by the user if needed, based on the outputs produced. Although there are many parameters to be considered, they are independent of each other. Therefore, once an optimal range has been found for the first parameter, the second parameter can be adjusted without the need to readjust the first, and so on.

With as many as fifteen million voxels in a 4D image, even the strictest inclusion criteria will fail to perfectly filter out noise voxels whose flow curves have randomly taken on a shape similar to the one we expect to find in the aorta. To reduce the amount of noise, three separate segmentation algorithms are run to segment flow in the $+x$ and $\pm$ z-directions. Figure 3.11 shows the output of one such algorithm. Notice the noise present in the segmentation. The three segmentations are noise corrected separately and then recombined to form a final segmentation. This is preferable since the combined noise of all three images may result in artefacts that are more difficult to filter than the individual images themselves.

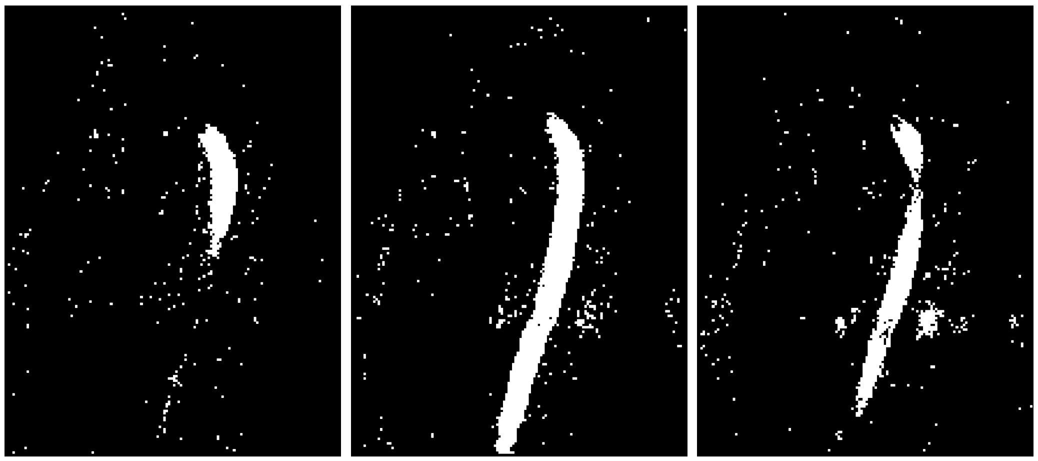


Figure 3.11: 2D sagittal slices of phase segmentations for flow in the –z direction before noise correction.

Two different noise reduction techniques were used which are discussed below. The first is obtained by examining six nearest neighbours of a voxel and determining the status of the majority (either they are included in the segmentation, or they are not). If there is a majority, that voxel is assigned the same status as its neighbours. This process can be iterated until the desired result is obtained. Figure 3.12 illustrates how this filter operates. We call this filter the nearest neighbour (NN) filter.

The second type of noise reduction employed here is the gaussian filter. The filter convolves the image voxels with a Gaussian kernel, which has the effect of transforming the voxel intensity

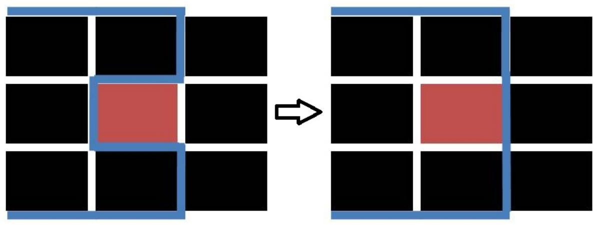


Figure 3.12: In the 2D example above, the blue line represents the segmentation before the nearest neighbour filter is applied. The red pixel has 5 out of 8 neighbours included in the segmentation, so the filter has the effect of changing the segmentation boundary to include the red pixel. This process may be iterated if necessary, so that more pixels are included or excluded from the segmentation.

to a weighted average of the voxels neighbours. The output of a Gaussian filter on a binary image gives values ranging between 0 and 1, so we threshold the result at 0.5 to obtain a noise-corrected binary segmentation (see Chapter 5). Although the results of the Gaussian filter are smoother it has the effect of eliminating smaller arteries, which may be undesirable depending on the intended application of the segmentation.

Unfortunately, this method of segmentation and noise reduction is still susceptible to erroneous segmentation of nearby structures that have similar flow characteristics to the aorta. A connectivity based algorithm could potentially extract the aorta from other structures, but in our case the resolution was too poor and the pulmonary artery appeared to be connected to the aorta in several areas.

3.3.1 GAC extraction

In order to solve the problem we encountered of segmenting other vessels with similar flow characteristics, we use a geometrical approach, and rely on the previously described GAC algorithm and its ability to maintain a low curvature contour. To initialize the GAC, a seed point is manually placed inside the aorta. Recall from Equation (3.3) that a feature image is used to define the speed at which the contour expands or contracts. Our initial segmentation serves as a binary feature image which allows the contour to wrap around the segmentation we have already produced. However, by increasing the weighting of the curvature term in the GAC evolution equation, we stop the contour from bleeding into nearby structures by maintaining a smooth boundary. Figure 3.13 illustrates the GAC extraction on one of the phase segmentations. This segmentation will be discussed later in Section 5.2.

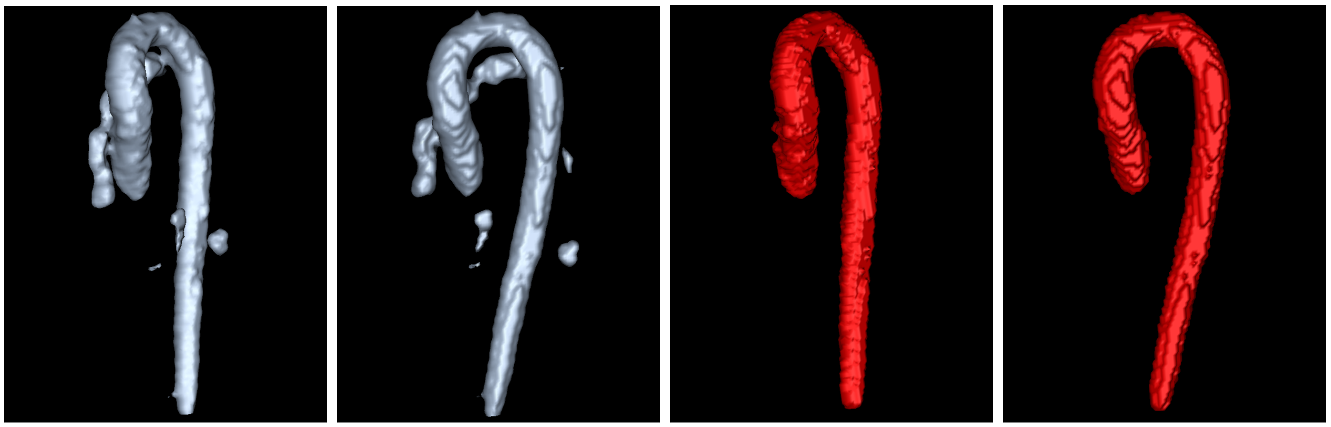


Figure 3.13: Two views of the same segmentation are on the left, where we see that some nearby structures were erroneously segmented. The pulmonary artery (behind the aorta), is being partially segmented, which is the main source of error. The poor resolution of our image makes it appear as if there is no physical separation between the pulmonary artery and the aorta, and their flow characteristics are very similar. By applying the techniques of this section, the aorta is extracted from the pulmonary artery and other structures. The GAC-extracted counterparts may be seen on the right.

Since the feature image is binary, the location of the initial seed (or small sphere) inside the aorta does not affect final results. However, placing a seed at either end of the aorta will result in a slower convergence of the level set since the boundary must travel the entire length of the artery. The user may also choose to place more than one seed for faster convergence.

Although this proposed phase-based technique of segmentation has advantages in speed (see Chapter 5.7) and ease of use (it only requires a minimum of one user defined seed, and its calculations are very simple), it may be desirable to reduce the number of variables needed to run the algorithm. Such schemes are presented in the next section. Ideally, magnitude image information would be used in tandem with phase-contrast information to make use of all available information. These issues are further addressed in section 3.5.

3.4 Curve-fitting pulsatile flow

We have demonstrated how mean and standard deviation calculations on the flow curves of the phase images can be good indicators of whether a voxel is inside the aorta. We now propose a more elegant curve-fitting approach which may possibly yield better results. To the best of our knowledge, there is no standard waveform that has been shown to fit measurements of blood flow through the aorta at any given 3D spatial location, although a theoretical waveform for patients with an abdominal aortic aneurysm has been proposed [35]. Determining a good waveform would help considerably by reducing the number of parameters needed to check for a 'good' flow curve. Below, we discuss the methods in which we fit the flow curves of our segmented voxels to a Gaussian waveform to determine whether the Gaussian curve could be considered a good candidate for modeling blood flow through the aorta. Two different fitting methods are used:

i) Flow curves at each 3D voxel go through a least-squares curve fitting routine to find the Gaussian with mean μ and deviation σ that best fits the flow curve. A range of μ and σ values are then defined which determine which voxels are allowed in the segmentation. This can be described succinctly by

$$x \in S(x) \text{ iff } G(x) = G(\sigma, \mu), \tag{3.7}$$

given the conditions

$$\sigma_{min} \leq \sigma \leq \sigma_{max}, \tag{3.8}$$

$$\mu_{min} \leq \mu \leq \mu_{max}, \tag{3.9}$$

where x is the voxel of interest, G is the Gaussian fit to the flow curve at voxel x, S is the segmentation region and $\sigma_{min,max}$, $\mu_{min,max}$ are chosen by trial and error. We will refer to this method as the curve-fitting segmentation, or CFS.

ii) Each flow curve is compared to a single standard Gaussian curve with $\mu = 28$ ms, $\sigma = 21$ ms. These values were chosen by trial and error. By a visual inspection, these values appear to segment the largest portion of the aorta while minimizing the number of erroneously segmented voxels. An error function is calculated as $\chi = (G(\sigma, \mu) - f(t))^2$ where G is the Gaussian function, and $f(t)$ is a measured flow curve. The error function χ produces a continuous image in each encoded direction, and an error threshold is defined to create a segmentation. We will refer to this method as the Gaussian deviation segmentation (GDS). Figure 3.14 shows an example of a Gaussian fit to the z-component of the MR velocity data at a voxel in the ascending aorta.

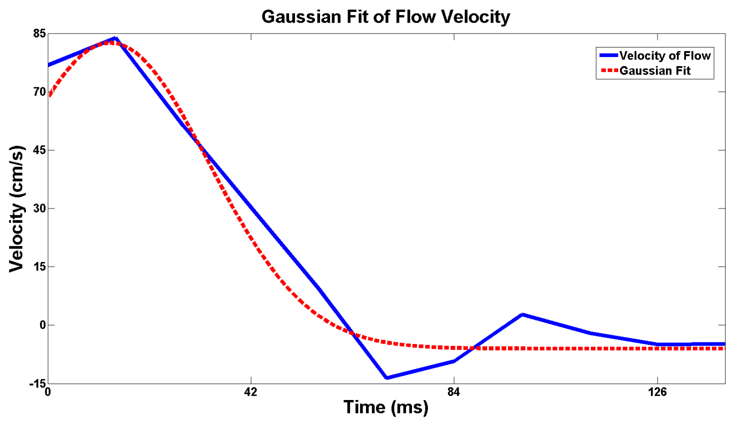


Figure 3.14: An example of a Gaussian fit to the z-component of flow at a voxel in the ascending aorta.

3.5 Combined Phase and Magnitude Segmentation Method

We have already discussed a number of segmentation techniques that take in magnitude image information or phase-contrast information as input. However, a segmentation technique which makes use of *both* data sets could be advantageous. This is especially true in our case, since the magnitude image we work with has poor contrast in the abdominal aorta, and good contrast in the aortic arch.

Conversely, the phase-contrast segmentations we will see in Section 5.2 show strong accuracy in the abdominal aorta, and poorer results in the aortic arch. A segmentation which makes use of the strengths of both methods might be more accurate.

To initialize our phase-magnitude (PM) segmentation , we use the GDS and CFS segmentations as initializations for the GAC level set algorithm. Instead of the chain of pre-processing filters previously used to create a feature image for the GAC, we define our new feature image by thresholding the voxel intensity of the magnitude image. Recall that our magnitude images have good contrast in the aortic arch and poor contrast in the abdominal aorta. Therefore, the feature image, being just a thresholded version of the magnitude image, encourages contour expansion/contraction in the arch while discouraging it in the abdominal aorta. In this way the evolution of the level set from its initial conditions is fastest where the magnitude image is the clearest and negligible where contrast is poor.

We use a thresholding algorithm built into the ITK-SNAP module which performs well when image contrast is good. The threshold defines a lower bound and a smoothness parameter which tapers the intensity of voxels near the lower bound. In our case, a small smoothing value seemed to improve results. Otherwise, the GAC algorithm is unmodified from section 3.2.

3.6 Applications and Flow Visualization

3.6.1 Euler method and interpolation of velocity data

Efficient methods of visualization and quantification are needed to convey the large amounts of data contained within the 4D phase-contrast images in a meaningful way. For the analysis of multidirectional blood flow in a 3D volume, visualization methods such as 2D vector-fields, 3D streamlines and 3D pathlines are common [36, 37, 38, 39]. These choices of visualization can potentially offer different and complementary representations of the same flow field.

The phase-contrast images define a time-varying velocity field under which we would like to track the path of a particle over time given some initial conditions. We define our variables as

$$\vec{y}'(t) = f(t, \vec{y}(t)), \qquad \vec{y}(t_0) = \vec{y}_0, \tag{3.10}$$

where $f(t, \vec{y}(t))$ is the velocity field and $\vec{y}_0$ is the initial position of the particle(s) under inspection. Our goal then is to solve this equation for $\vec{y}(t)$, which in our case is a three-dimensional vector. From the definition of the derivative, we can derive the Euler method of solving this differential equation. Given

$$\vec{y}'(t) = \frac{\vec{y}(t+h) - \vec{y}(t)}{h}, \tag{3.11}$$

$\vec{y}(t)$ can be recursively solved for by

$$\vec{y}(t+h) = \vec{y}(t) + hf(t, \vec{y}(t)). \tag{3.12}$$

The accuracy of such a solution is dependent on the step size h. Other methods such as Runge-Kutta [40] have also been used in particle traces which have been shown to converge faster than its Euler counterpart. For simplicity, we use the Euler solution and a step size h that is much less than the temporal resolution of our image ($h << 14$ ms). Notice that each particle in the trace is governed by the same equation and is therefore highly parallelizable. The majority of computing time will be spent updating the solution at each iteration.

Recall from Equation (3.12) that the velocity field $f(t, \vec{y}(t))$ is derived from the phase-contrast images. To evaluate $\vec{y}(t)$ at points between voxels, some type of velocity interpolation must be used. We use a quadrilinear interpolation in the case of the time-varying field. With a static field $f(t, \vec{y}(t)) = f(\vec{y})$, only trilinear interpolation is needed.

Trilinear interpolation can be described as follows: given a point p on some three-dimensional regular grid, let x_0, x_1 be the grid points nearest p in the x-direction, with x_0 being the smaller of the two. y_0, y_1, z_0, and z_1 are similarly defined. Let α, β, γ be the difference of p from x_0, y_0 and z_0, respectively. For example, if p lies at (2.3, 3.7, 3.3), $\alpha = 0.3, \beta = 0.7, \gamma = 0.3$. Then a linear interpolation of a function $f(p)$ in the x-direction can be written as

$$f(p)_x = (1 - \alpha)f(x_0) + \alpha f(x_1), \tag{3.13}$$

$$f(p)_x = 0.7f(2) + 0.3f(3). \tag{3.14}$$

Similarly, f can be interpolated in the y and z -directions by substituting the appropriate constants in Equation (3.14). Figure 3.15 illustrates linear interpolation in two dimensions, which can easily be extended to the three dimensional case. Equation (3.14) is also easily extended to four-dimensions by introducing another offset for the time dimension.

$$f(p)_t = (1 - \omega)f(t_0) + \omega f(t_1). \tag{3.15}$$

With this formulation, a point near the segmentation boundary will be weighted by velocity data outside the segmentation. For this reason, any velocities outside the segmentation are set to zero before any interpolation. Once these steps have been taken, we can solve Equation (3.12) given some initial conditions y_0.

3.6.2 Analysis planes

For flow simulations (discussed later in Section 3.6.5), we would like to place some particles into our segmentation and track their progress through the velocity field by the interpolation method just described. While we could easily place the initial particle locations at random locations within

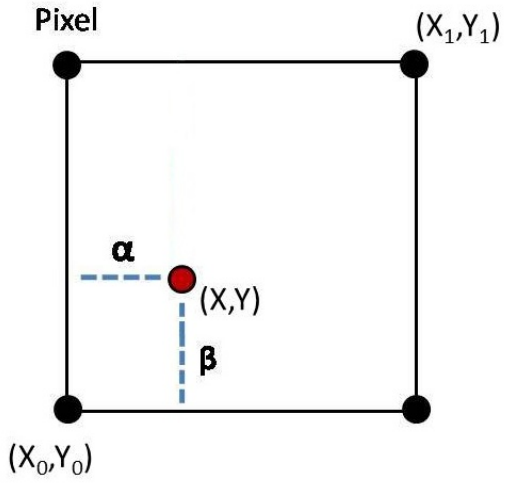


Figure 3.15: If we wish to interpolate for some value $F(X, Y)$, we use Equation (3.14) to define α and β as shown above. The black circles represent the image pixels, while the red circle shows the point which we would like to interpolate.

the aorta, it would be more desirable to define some analysis plane that is perpendicular to the longitudinal axis of the vessel. Velocity data acquired in this 2-D plane is useful since it can be used to quantify a number of parameters including flow rate and shear stress [41]. The analysis plane can also be referred to as an emitter plane, since it is a both a plane in which we can emit particles for simulation, and from which we can measure flow parameters.

In our definition of the analysis planes, we assume that the aorta can be locally approximated by a cylinder. Therefore, an ideal analysis plane can be defined at any point within the aorta as the plane parallel to the cylindrical face. Since a plane may be defined by $\vec{n} \cdot (\vec{r} - \vec{r_0}) = 0$, where $\vec{r_0}$ is a point on the plane, our problem is then reduced to finding the normal vector $\vec{n}$ and the center point of the cylinder (see Figure 3.16).

If we were to find the center line of the entire aorta, we could then use any point on the line to define planes throughout the vessel. We can search for the center line of the aorta by noting that the points furthest away from the vessel walls corresponds to the vessel center. We use a signed distance function (SDF) to calculate the distance from any voxel in our image to the vessel wall

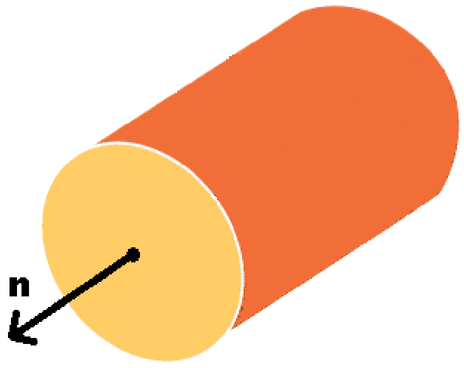


Figure 3.16: If we imagine a small length of the aorta as a cylinder, the analysis plane we want to define is just the cylindrical face. Finding $\vec{n}$ and the point at the center of the cylinder defines the plane. (Image modified from [42].)

defined by our segmentation. In general, the SDF is defined as

$$f(\mathbf{x}) = \begin{cases} d(\mathbf{x}, \mathbf{\Omega^c}), & \text{if } \mathbf{x} \in \mathbf{\Omega}, \\ -d(\mathbf{x}, \mathbf{\Omega}), & \text{if } \mathbf{x} \in \mathbf{\Omega^c}, \end{cases} \tag{3.16}$$

where Ω is the segmented region and Ω^c is its complement, $\mathbf{x}$ is a point in 3D space, and $(\mathbf{x}, d)$ is some metric space. Notice that any point inside the boundary will be positive, increasing in value the further away from the boundary. Therefore, the centerline will correspond to the highest values in the SDF. Ω will be defined by our segmentation and we use the Euclidean metric. The resulting SDF applied on the CFS segmentation can be seen in Figure 3.17.

It is not immediately obvious from Figure 3.17 where the centerline lies. Consider Equation (3.16) evaluated at two neighbouring points on the centerline, x_1 and x_2. If the radius of the aorta does not change dramatically in the region, then $f(x_1) \approx f(x_2)$ (Figure 3.18). Then taking the 3D gradient of Equation (3.16) gives $|\nabla f(x_1)| \approx |\nabla f(x_2)|$. Imaging the gradient allows us to discern the centerline of the vessel in Figure 3.19.

It is then a simple matter to threshold Figure 3.17 or Figure 3.19 and obtain the centerline in

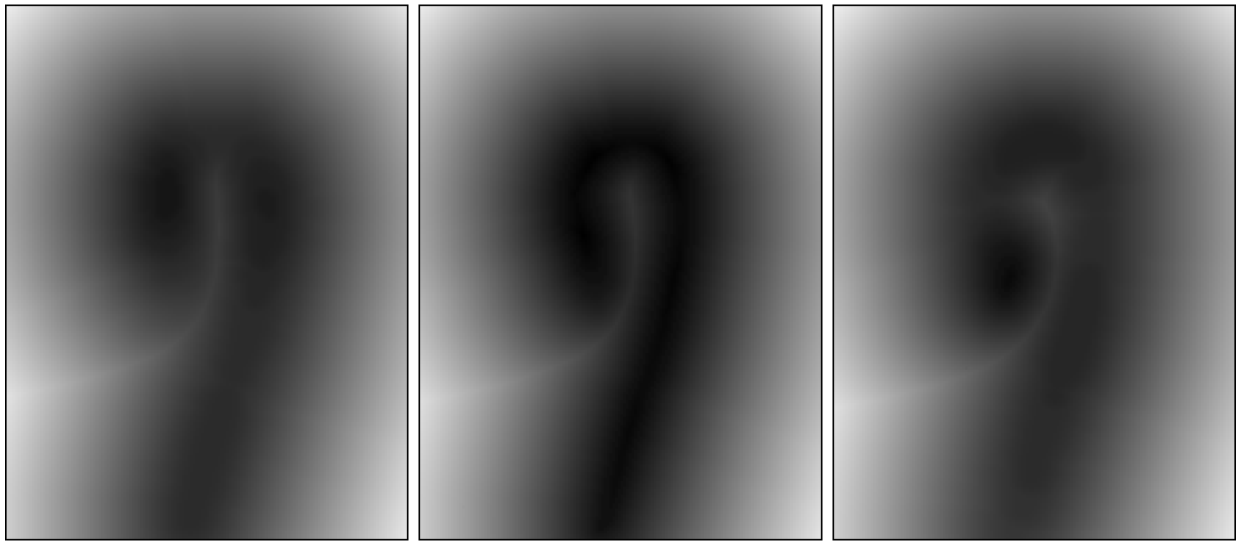


Figure 3.17: Three 2D sagittal slices that show the signed distance function (Equation (3.16)) of an aortic segmentation. Darker intensities correspond to voxels inside the aorta, but further away from the aortic edge. Voxels outside the aorta become brighter the farther away they are. The centerline of the vessel is darkest, though it is not discernible by eye here.

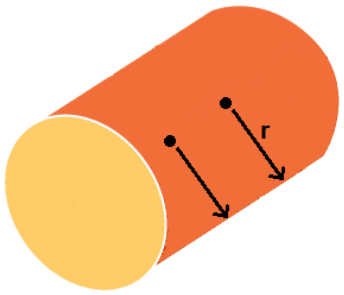


Figure 3.18: For a small section of the aorta, we can assume nearly constant radius and approximate the vessel shape as cylindrical. The distance from two neighbouring points on the centerline are approximately the same, and equal to the radius of the vessel.

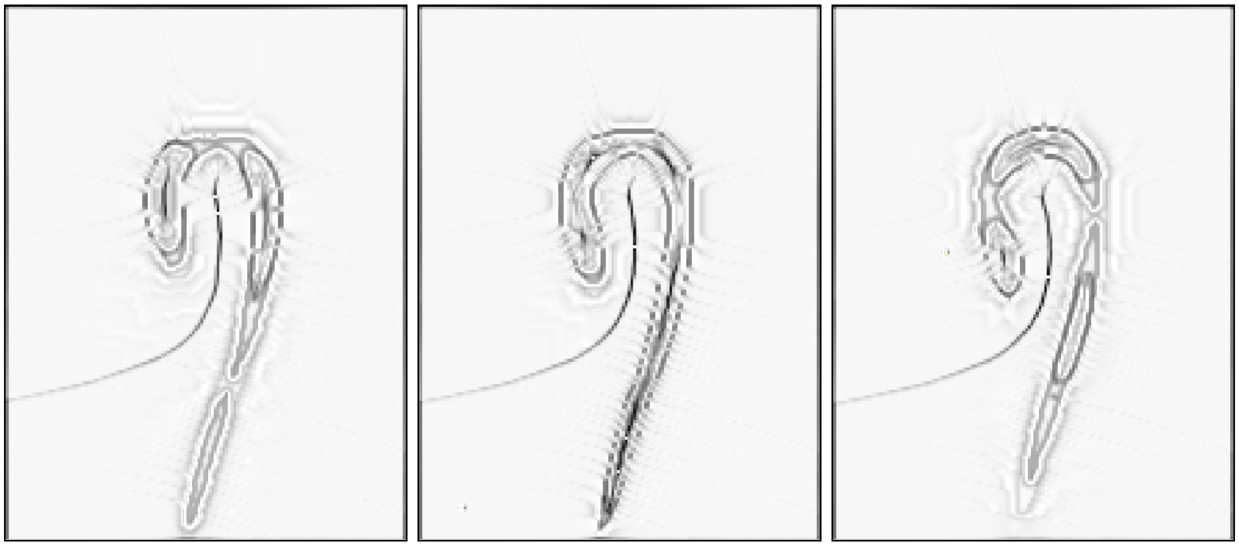


Figure 3.19: Gradient of Figure 3.17. The centerline is now discernible and lies mostly in the center image slice. We can easily filter out the rest of the image with a simple thresholding algorithm.

its entirety. Programs like ITK-SNAP allow users to interactively control the threshold and see results in real-time. A thresholded image is shown in Figure 3.20. Figure 3.21 shows the centerline overlayed on the magnitude image for more clarity. Given a point on this centerline, the normal vector of the analysis plane we wish to define can be calculated by principal component analysis (PCA) and singular value decomposition (SVD), as discussed in the next section.

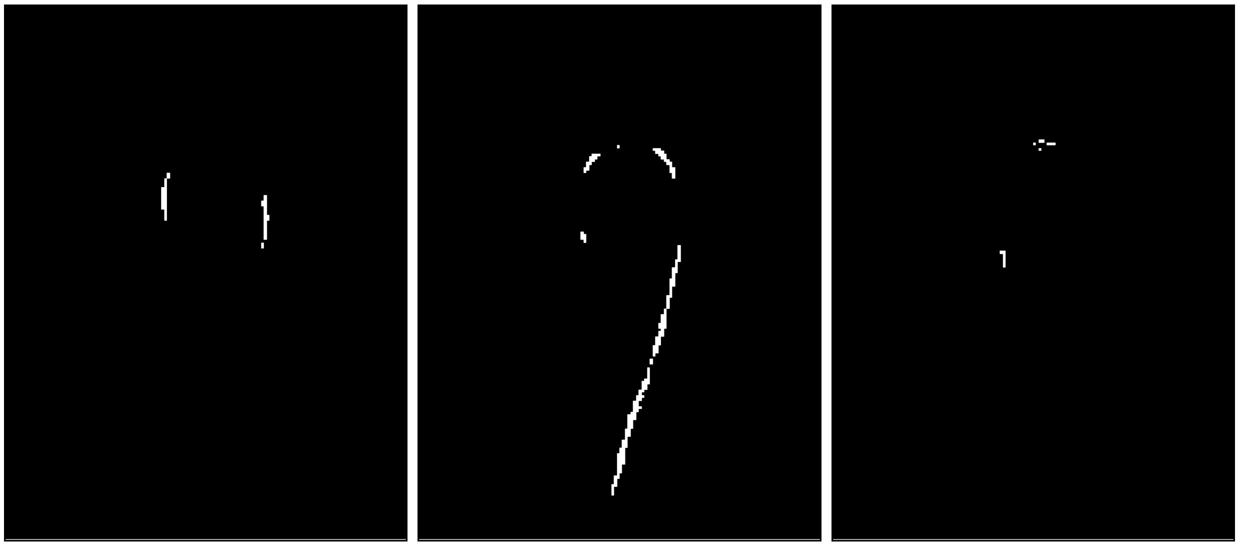


Figure 3.20: The centerline left over after thresholding the SDF gradient. The line is 2-3 voxels wide. These points can now be used to approximate the cross-sectional normal vector $\vec{n}$ at any point in the vessel.

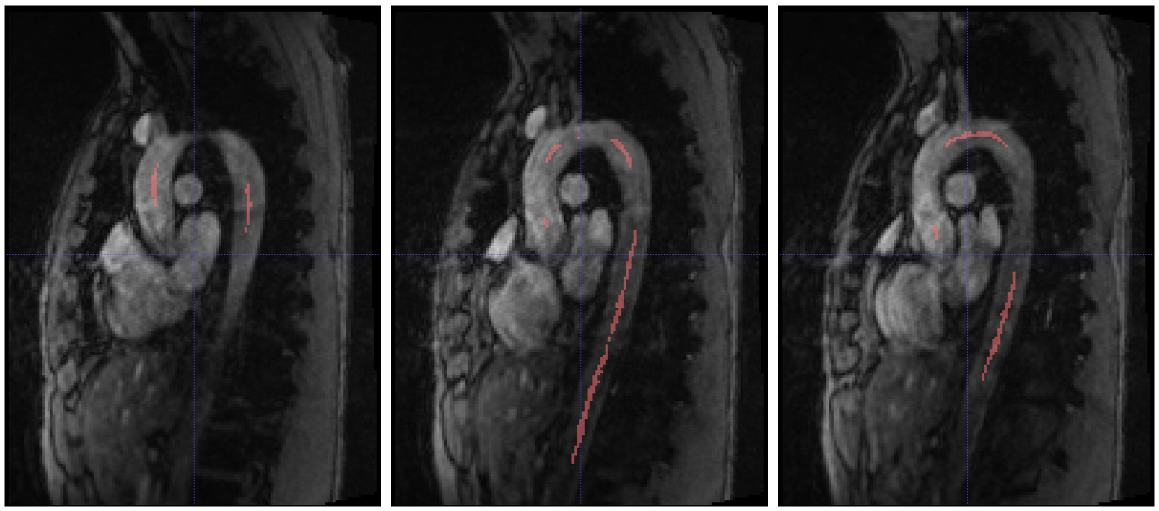


Figure 3.21: The centerline overlayed on the magnitude image.

3.6.3 Principal component analysis and singular value decomposition

In general, principal component analysis is a method used to transform a set of points or observations which are possibly correlated, into a set of linearly uncorrelated variables called principal components. The principal components are mutually orthogonal, and they may be regarded as basis vectors. The transformation is defined so that the first principal component has the largest variance with the data, with each successive component having the next highest variance. When this transformation is applied to a line in 3D space, it is easy to imagine that the condition of highest variance sets the first two principal components to vectors perpendicular to the line. The idea then, is that the 3rd principal component will be parallel to the line, since it must be orthogonal to the other two.

Figure 3.22 illustrates the principal components of points scattered in a Gaussian distribution. When applied to points scattered in a linear distribution, the third principal component corresponds to the least squares linear fit of the data. This linear fit is exactly the normal vector to our analysis plane we wish to find. The procedure will then be to take a small neighbourhood on our centerline, and using PCA, extract the third principal component which is the least-squares fit for the normal vector. The determination of the exact size of the neighbourhood will be discussed later in Chapter 5.6.

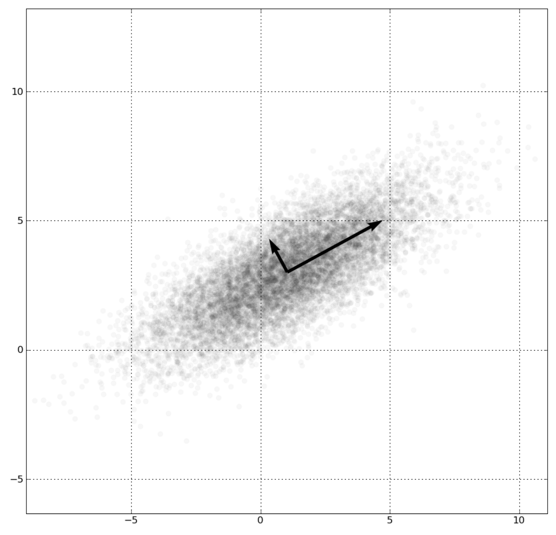


Figure 3.22: The two principal components for a scattered Gaussian distribution are shown in two dimensions. The first component (shown with smaller magnitude) has the highest variance. (The above is modified from a public domain image.)

Suppose we define a small neighbourhood in our image somewhere on the centerline so that s points are in the neighbourhood. The transformation $\mathbf{T}$ that transforms a set of s 3D points $\mathbf{M}$ in this way is defined as

$$\mathbf{T} = \mathbf{M}\mathbf{W}, \tag{3.17}$$

where $\mathbf{W}$ is an $s \times s$ matrix whose columns are the eigenvectors of $\mathbf{M^T M}$, and $\mathbf{M}$ is a $s \times 3$ matrix. Note that for Equation (3.17) to hold, $\mathbf{M}$'s mean value must equal zero. An elegant solution method presents itself if we consider the SVD Theorem [43]. The theorem states that for any $m \times n$ matrix $\mathbf{M}$ with real values, there exists a factorization of the form

$$\mathbf{M} = \mathbf{U\Sigma V^T}, \tag{3.18}$$

where U is an $m \times m$ unitary matrix, V is an $n \times n$ unitary matrix and Σ is a diagonal matrix. Since U and V are unitary, the columns of each matrix form orthonormal vectors which can be regarded as basis vectors. Then if $\mathbf{M}$ is comprised of the coordinates of s centerline points within some neighbourhood in our image,

$$\mathbf{M} = \begin{pmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_s & y_s & z_s \end{pmatrix} \tag{3.19}$$

We set the mean to zero by modifying $\mathbf{M}$ to

$$\mathbf{M} = \begin{pmatrix} (x_1 - \mu_x) & (y_1 - \mu_y) & (z_1 - \mu_z) \\ \vdots & \vdots & \vdots \\ (x_s - \mu_x) & (y_s - \mu_y) & (z_s - \mu_z) \end{pmatrix} \tag{3.20}$$

where $\vec{\mu} = (\mu_x, \mu_y, \mu_z)$ is the mean of the distribution. We may now apply PCA to $\mathbf{M}$. The matrix $\mathbf{M^T M}$ can be written as

$$\begin{aligned} \mathbf{M^T M} &= \mathbf{V \Sigma U^T U \Sigma V^T} \\ &= \mathbf{V \Sigma^2 V^T}, \end{aligned} \tag{3.21}$$

which is the eigenvector decomposition of $\mathbf{M^T M}$. This exactly matches the criteria defining the matrix $\mathbf{W}$ in Equation (3.17). This allows us to set $\mathbf{W} = \mathbf{V}$ in Equation (3.17), giving us

$$\begin{aligned} \mathbf{T} &= \mathbf{MV} \\ &= \mathbf{U \Sigma V^T V} \\ &= \mathbf{U \Sigma}. \end{aligned} \tag{3.22}$$

It is easy to see from Equations (3.17) and (3.22) that the columns of $\mathbf{V}$ are then the principal components of $\mathbf{M}$. Therefore, to obtain the third principal component of a set of data, one needs only to extract the third column of the matrix $\mathbf{V}$.

3.6.4 Summary of procedure for analysis plane definition

To define a plane in general, a point P on the plane and a normal vector $\vec{n}$ are needed. In an effort to define an analysis plane which cross-sects the aorta at any point, we first find the center line. A signed distance function transforms our image so that an image gradient maximizes the pixel intensity at the centerline, making it easily discernible. P can be defined to be any point on this line. To find $\vec{n}$, we notice that a small neighbourhood around P resembles some linear data with error. Applying singular value decomposition and using the theory of principal component analysis, we can extract the third principal component which corresponds to the least squares linear fit of our data points. This component vector is $\vec{n}$.

The reader may be curious as to why this particular technique of linear least squares fitting was used. By formulating the least-squares problem in matrix form, we allow ourselves many advantages in the MATLAB environment, which was designed to operate primarily on matrices and arrays. Additionally, many parallelized matrix operations are pre-defined in MATLAB which are easily integrated into the GPU architecture, which we will make use of later.

3.6.5 Visualization and particle traces

A method of velocity interpolation and the definition of analysis planes now allows us to visualize flow data within the segmented aorta. Many different visualizations of flow are possible, with each technique potentially offering different and complementary representations of the same flow field. Since it is not possible to clearly show without overlap all the velocities measured in a volume, the method of visualization is important.

Two visualization methods are streamlines and pathlines. A streamline is defined as a path instantaneously tangent to the velocity vector of the blood flow, whereas pathlines can be thought of as the path a virtual particle would take when released in the time-varying flow field. After placing an analysis plane (or emitter plane) within the aorta, virtual particles are seeded at points within the plane. Streamlines can then be calculated forwards and backwards in time to show flow velocity along an arbitrary length of the streamline. We can acquire different streamline representations at any point in the cardiac cycle. The easiest way to convey speed information for both streamlines and pathlines is to colour code each trace according to the instantaneous speed. Recall from Section 3.6.1 that velocities outside the segmentation are set to zero. This way particles that hit the segmentation boundary do not escape.

Pathlines are emitted at a chosen time-frame and propagate according to the time-varying flow field. By emitting pathlines at successive instants in time, we can visualize the dynamics of 3D blood flow over the cardiac cycle. Particles speed up during systole, and are directed away from the heart. During diastole, the particles slow down and diffuse. For this reason, particle traces are usually emitted for short time periods, and if further information is required, a new emitter plane will be defined.

Some characteristic flow patterns have been observed in healthy patients. Helical flow patterns and mild early diastolic retrograde flow have been identified in the ascending aorta, while abnormal flow patterns can also be distinguished in cases of aortic aneurysm. In Figure 3.23, a vortical flow pattern can be seen enveloping an aneursym. Visualizations like these could serve as good early warnings for aortic disease, or as a data check when used in combination with other velocity map acquisitions. In addition to visualization, a pathline representation of flow can be used as an internal check of data quality. A large number of pathlines inappropriately leaving the blood pool indicates insufficient data quality, which might be due to noise, background error, or insufficient spatial or temporal resolution [55].

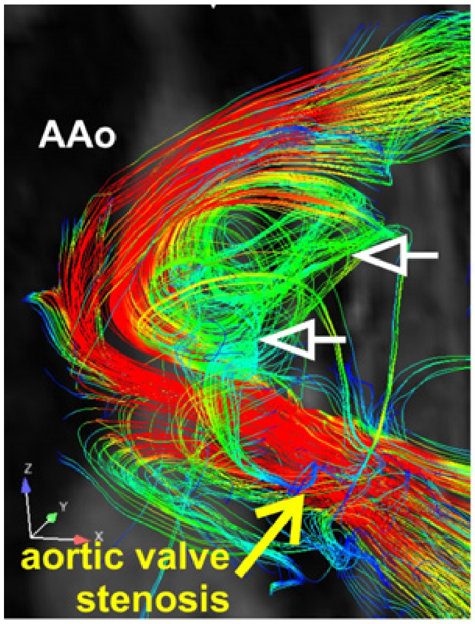


Figure 3.23: An example of pathline visualization, in a non-healthy patient. In the ascending aorta (AAo), a nine year old pediatric patient with aortic valve stenosis has a vortex flow pattern of pathlines which occupies the shape of the aneurysm. (Image from M. Markl et al [55].)

Chapter 4

Parallel computing

In this chapter, we discuss computing on the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU), and how either architecture can be used to compute in parallel, or to perform multiple instructions simultaneously. Different forms of parallel computing, such as multi-core processing, hyper-threading, and GPU computing are discussed, and we discuss why parallel computing is becoming so important in the field of medical imaging. We also describe in the architecture of the GPU conceptually, and how it is used to accelerate our segmentation algorithms.

4.1 Multi-core processing and hyper-threading

In the last few decades, single-core processors have undergone a trend described by Moore's Law [44], roughly doubling their performance every 18 months. Recently, however, the performance of single-core processors has slowed due to excessive power dissipation at GHz clock rates [17]. The advancement of parallel computing has encouraged many programmers to increasingly shift their algorithms to parallel computing architectures for more practical processing times. Already, there has been much enthusiasm for the advantages GPU computing offers in fields and disciplines such as linear algebra, differential equations, ray tracing, computational biophysics and fluid dynamics, to name a few [45, 46, 47, 48].

Normally, a CPU core can only handle one instruction at a time. When programs on a computer are run at the same time, the processor is asked to execute several commands simultaneously. The list of instructions would be handled sequentially, and doubling the amount of data to be processed by a CPU would also double the processing time involved. Within the CPU architecture there have been two advancements in technology that allow us to partially avoid the bottleneck of serial processing: multi-core CPUs, and hyper-threading.

A multi-core processor is simply a CPU with physically separate cores on the same chip. Multiple I/O (input/output) registers control the traffic of instructions by sending commands to idle cores so that multiple instructions can be executed in parallel. Some level of coordination is needed between cores. Imagine processors A and B are working on programs 1 and 2. If processor A finishes program 1, it might find that the next program on its to-do list, requires some initial data from program 2. It must wait for processor B to finish its calculations before proceeding. For this reason, an X-core processor does not equal the processing power of a single-core processor multiplied by X, but it is a close approximation if you avoid cross-communication bottlenecks. As always, the actual speedups depend on the application and implementation of parallel coding.

In contrast to the multi-core approach, hyper-threading maximizes the throughput of any single core on the CPU. If a core starts a calculation and finds that it requires additional data that has not yet arrived from memory or another core, it will continue work on the next instruction in its list until that information is available. In this way, the CPU keeps any delays to a minimum and from the software's viewpoint, appears to be executing two commands simultaneously. The speedups of hyper-threading again depend on the application, but have been found to be around the 30% mark [49]. Hyper-threading can then be implemented on multi-core processors, so that each individual core has maximized throughput.

4.2 Graphics Processing Units

The GPU was first introduced to process pixel and texture data, involving calculations that are inherently parallel and very time-consuming on a CPU. As the GPUs name suggests, this data is used to display graphics, in the form of complex 3-D scenes. Since 1997, the number of GPU cores have doubled every 1.4 years [17], and have become increasingly sophisticated with expanded instruction sets and support for double-precision floating-point arithmetic, built-in mathematical functions, and more [17].

An application programming interface (API) is useful to develop programs for the GPU. A common API, and the one used in this thesis, is NVIDIA CUDA (Compute Unified Device Architecture). CUDA extends the computer language *C* to allow the user to access GPU resources. For example, a developer may write an application in *C* that executes on the CPU, and only calls the GPU to perform a parallel task when a separate program called a kernel is executed within the application. The CUDA library allows this kernel access to the GPU and to a variety of built-in functions that are optimized for parallel computing, for example, the summation or multiplication of two or more large data arrays or matrices.

To visualize how a kernel is executed on a GPU, refer to Figure 4.1. A GPU consists of a grid which is divided into blocks and threads. Each block manages a group of threads, and issues instructions in parallel to them. Memory is shared between these threads in the form of registers, local memory and shared memory. Data is stored in these hierarchical memories in order of how frequently they need to be accessed. In addition to block-local memory, there is also global memory alotted for interaction between blocks. Although the GPU is designed to issue the same instruction to all threads within a block, threads can follow different branches of the same kernel. For example, developers can place an *if* statement within a kernel that will determine which threads follow which branch of code. This will, however, substantially reduce the performance of the GPU since these diverging threads are serialized [17].

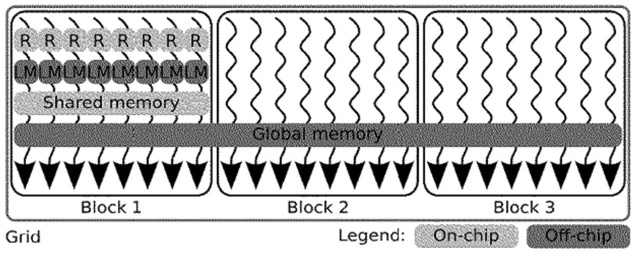


Figure 4.1: A grid representing the GPU. The grid is divided into blocks with are further subdivided into threads. Local memory (LM), registers (R) and shared memory for threads are shared within each block. Threads within the same block are all given the same instructions. (Image modified from B. Pratx and L. Xing [17].)

Optimal performance with a GPU depends on the careful allocation of computing and storage resources. A major bottleneck for any application is the transfer of data between the computer host and the GPU device. Transfer of data should be minimized at all times. A careful analysis of code efficiency could reveal that implementing small serial calculations on the GPU in between parallel tasks may be faster than switching back and forth between the CPU and GPU. In the end, the highest achievable speedup is determined by the sequential fraction of the program. This is known as Amdahl's law [50], where the speedup S is given by

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)} \qquad B \in [0, 1], \tag{4.1}$$

where B is the fraction of the algorithm that is serial and n is the number of threads.

4.3 Algorithm Optimizations

The segmentations introduced in Chapter 3 can all be parallelized to some extent. For example, each point on an evolving level set contour evolves according to Equation (3.3). The level set algorithm can be parallelized by assigning a GPU thread to each voxel on the contour to calculate its evolution independently of other voxels. Since there have already been numerous studies on the optimization of the level set for the GPU [16, 17], we have foregone any optimization of the level set in this thesis, and have focused on the phase-based segmentation algorithms.

The phase algorithms of Sections 3.3 and 3.4 both involve the examination of flow curves at each voxel of the image. An image with N voxels yields $3N$ curves which must be examined (one for each phase-encoded direction). The $3N$ curves are each assigned their own GPU threads so that any calculations required of the algorithm (mean flow speed, standard deviations of flow speed, or Gaussian fitting) are all done in parallel.

The visualization of flow through the aorta is also inherently parallel, with each particle of the simulation evolving under Equation (3.12). Although a parallel algorithm was developed for the particle trace calculations, other algorithms still need to be written for the GPU in order to display the wealth of information in the form of an animation or 3D image. To the best of our knowledge, MATLAB does not yet support graphical GPU functions that would allow the quick rendering of a particle trace animation. This is discussed further in Section 5.7 and Chapter 6.

4.4 Computer Specifications

GPU computing work in this thesis was done using the NVIDIA Tesla C2070 featuring 448 CUDA cores at 1.15 GHz. The C2070 has a total dedicated memory of 6GB GDDR5 with a memory speed of 1.5 GHz [51]. Double and single precision floating point performances are 515 Gflops and 1.03 Tflops, respectively. The workstation used a 3.2 GHz Intel Xeon E5-1650 six-core CPU [52] with a 12 MB cache and 16 GB DDR5 RAM. The processor uses 'Hyper-Threading Technology', so that two threads per core can run calculations simultaneously. Algorithms for the GPU were written on the CUDA platform, or in MATLAB (see Appendix A for source code).

Chapter 5

Results

This chapter presents the results of the segmentation algorithms first introduced in Chapter 3. We describe the strengths and weaknesses of each algorithm, and choose the most accurate segmentation for calculation and visualization of flow through the aorta. Using the definition of analysis planes also described in Chapter 3, we set up a plane for the emission of particles into the aorta, in an effort to simulate the flow of blood. The motion of these particles are tracked by the interpolation method in Section 3.6.1.

5.1 Magnitude Segmentation

As discussed in Chapter 3, the level set requires a feature image which will control the level set contours growth. If the feature image is an edge image, we can discourage growth of the contour past image edges, and encourage it elsewhere. Ideally, if part of the contour reaches an image edge before another, the slowing of growth will be enough to let other parts of the contour catch up until the whole aorta is wrapped by the level set. We shall see in the following section that this unfortunately is not the case for our images.

To create a feature image, choices for parameters need to be made for the various pre-processing filters. The MCDE smoothing parameter, k, the Gradient Magnitude kernel width, σ, and the sigmoid parameters are all listed in Table 5.1. The values were chosen by trial and error, with two sets of values for the upper and lower portions of the image. This was necessary because it was impossible to create a satisfactory feature image of the aorta due to its inhomogeneous intensity. Some parameters needed to be fine tuned in order to pick up the very faint edges of the abdominal aorta. Figures 5.1 and 5.2 show the preprocessing stages for the upper and lower image. These images were combined to create the feature image for the magnitude level set.

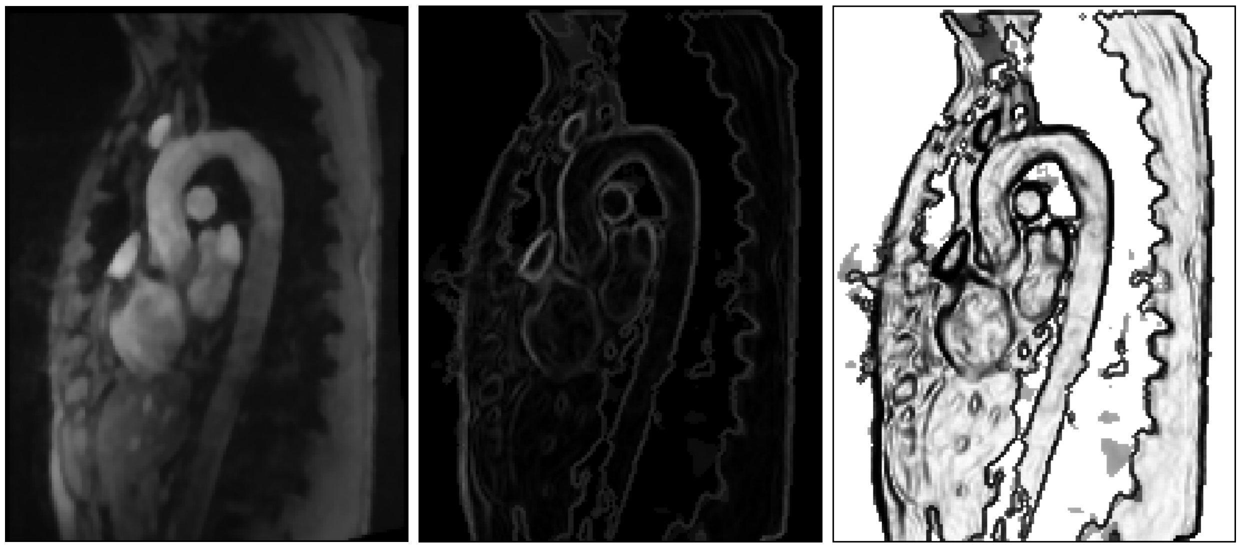


Figure 5.1: Pre-processing images for the upper magnitude segmentation. Notice that the edge detection in the lower aorta is very poor, creating a number of jagged edges at the bottom of the image. This necessitates another set of filter parameters for this portion of the image.

Small seed points (three-dimensional spheres of radius < 3 voxels) were manually inserted along the aortic center line for the initial level set contours. Values of parameters passed to the GAC algorithm are presented in Table 5.1. The final results of the magnitude segmentation are represented as an overlay on the original magnitude images in Figure 5.3.

As discussed in the caption of Figure 5.3, the magnitude segmentation performs very poorly, due to the combination of poor initialization of the level set contour and poor image contrast leading to poor edge detection (which controls when the level set stops expanding). It is not obvious how long the algorithm should iterate before completing - if too long, the level set expands beyond the desired boundary; if too early, the contour will not have yet reached the edge of the aorta. In Figure 5.3,

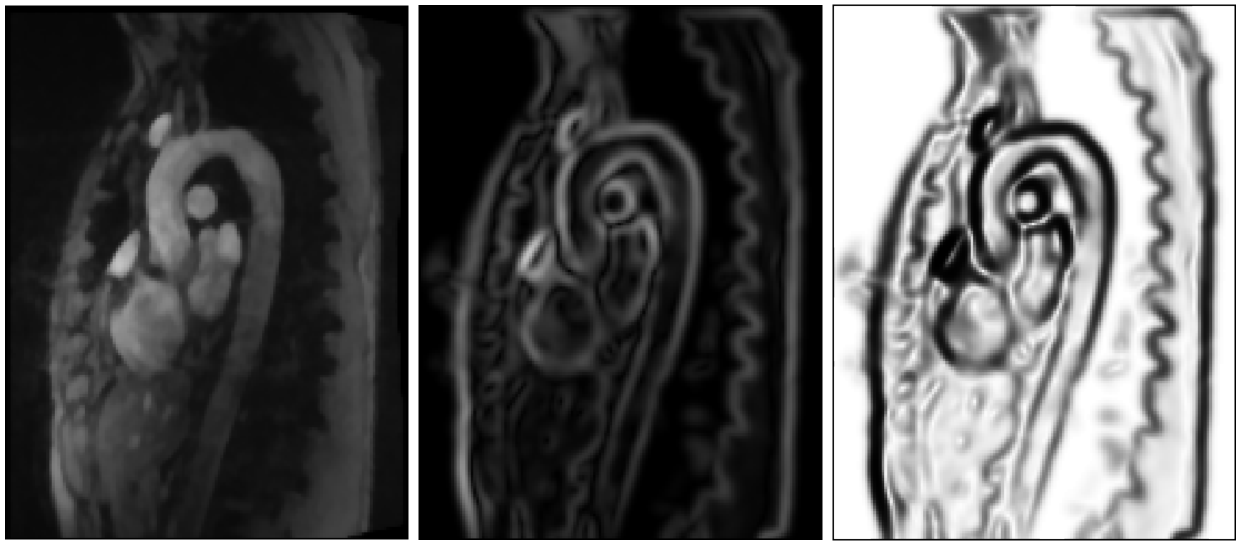


Figure 5.2: Pre-processing images for the lower magnitude segmentation. A large amount of smoothing is used in order to distinguish clearly the abdominal aortic wall with the background of the image. Unfortunately, the edge detection is still not ideal, but it is an improvement.

	Upper region	Lower Region
	Smoothing Filter	
k	9	13
	Gradient Magnitude Filter	
σ	0.01	3
	Sigmoid Filter	
α_s	-3.8	-2.4
β_s	5.2	4.2
	GAC	
α	5	5
β	0	0
γ	5	5

Table 5.1: Filter values for magnitude image segmentation. Larger smoothing parameters k and σ are used in the lower region in an effort to eliminate noise. Optimal α_s and β_s values are estimated based on the output intensites of the Gradient Magnitude Filter (refer to Equation (3.6)). Large propagation and advection parameters are passed to the GAC in order to wrap the level set boundary to image edges (refer to Equation (3.3)). The curvature term β is supressed because it discourages the growth of the initial small spherical contours, while the α and γ terms were chosen through a trial and error process.

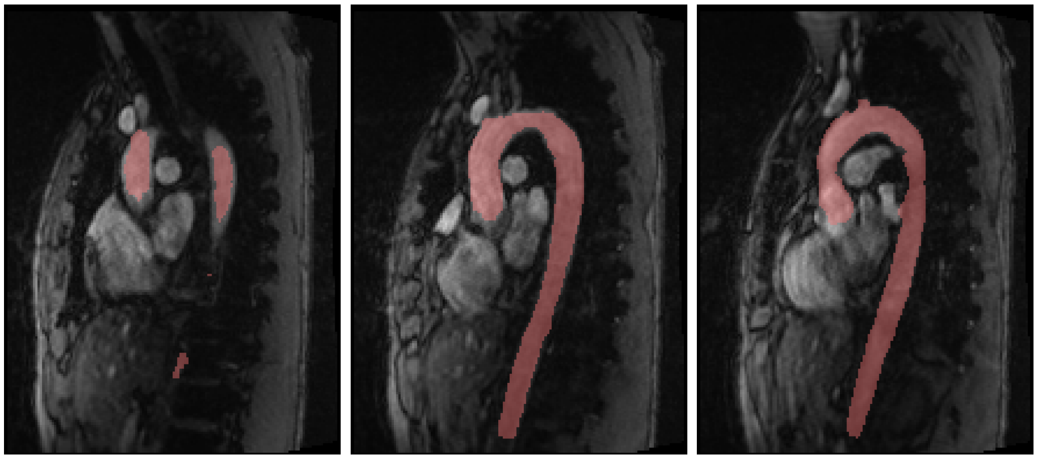


Figure 5.3: Results of the level set segmentation (red) overlayed on the magnitude image. The level set conforms well to image edges in the center image where edge detection was good. The left and right images show different image slices of the same segmentation. The left image shows the level set has not yet converged on image edges, while in the right image the contour has expanded beyond the image edges.

we see that the contour has not expanded uniformly, and some portions of the segmentation have expanded outside of the aorta, while others have not.

5.2 Phase Segmentation

Recall from Section 3.3, the goal of the phase segmentation was to examine each voxel in the image, and find some criteria for the inclusion or exclusion of the voxel from the segmentation based on the flow data obtained there. We calculate mean and standard deviation values of flowspeed for different cycles of the heart (early systole, systole, diastole, and the entire heart cycle), and narrow the ranges of mean and deviation values we will accept into the segmentation until an optimal range is found.

Table 5.2 shows the acceptable ranges of mean and standard deviation values for different sections of a flow curve as determined by trial and error. Three regions of the aorta are segmented separately, each corresponding to a different predominant direction of flow. Region 1 is the ascending aorta, where the predominant direction of flow is the $+z$-direction. Region 2 is the aortic arch, where the predominant direction of flow is in the x-direction. Region 3 is the descending and abdominal aorta, where the predominant direction of flow is the $-z$-direction.

In addition to the mean and standard deviation values, it was found to be useful to include additional criteria for Regions 1 and 2. In Region 1, the aorta is very bright on the magnitude image and we may use a minimum voxel intensity i as a check on our data. In Region 2, we only include voxels with a max velocity of 34.5 cm/s in the x-direction. In practice we found that the segmentations in Regions 1 and 3 capture most of the aortic arch, so only the high velocities in the x-direction need to be captured.

Region 1 (+ z-direction)							
m_{es}	-	m_s	(3.125, 150)	m_d	(0, 12.5)	$\|m_c\|$	(0, 6.25)
σ_{es}	-	σ_s	(0, 37.5)	σ_d	(0, 12.5)	σ_c	(0, 6.25)
Additional dependencies: $i_{min} = 70$							
Region 2 (x-direction)							
m_{es}	(15.6, 53)	m_s	(9.5, 50)	$\|m_d\|$	(0, 12.5)	$\|m_c\|$	(0, 12.5)
σ_{es}	(0, 18.75)	σ_s	(5, 43.75)	σ_d	(0, 15)	σ_c	(6.8, 31.25)
Additional dependencies: $v_{max} > 34.5$							
Region 3 (− z-direction)							
m_{es}	(-45, -18.75)	m_s	-	$\|m_d\|$	(0, 7.5)	$\|m_c\|$	(0, 9.5)
σ_{es}	(0, 18.75)	σ_s	(0, 37)	σ_d	(0, 7.5)	σ_c	(0, 9.5)
Additional dependencies: None							

Table 5.2: Table of acceptable ranges for mean (m) and standard deviation (σ) values for flow curves (in cm/s). Refer to Figure 3.10, which define the periods over which each m and σ value are calculated. A range of acceptable parameter values are listed for each predominant direction of flow in the aorta. Lower and upper bound values for each parameter are listed in brackets. Some parameters did not improve the segmentation results and are listed with no range. A minimum intensity, i_{min}, and a lower bound for maximum velocity, v_{max}, were also found to improve segmentations for regions 1 and 2.

The results of one of these segmentations is shown in Figure 5.4. The noise reduction techniques of Section 3.3 are next applied to these segmentations to eliminate the random voxels which have erroneously been segmented.

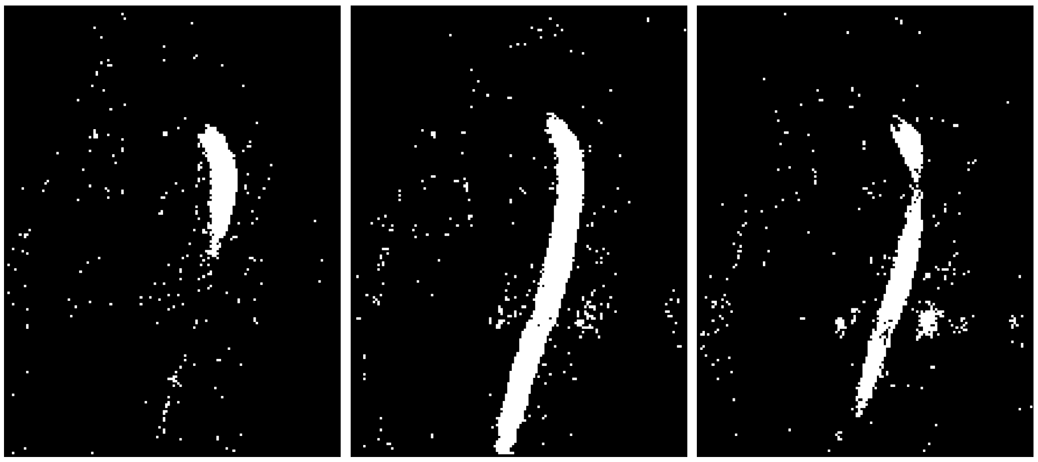


Figure 5.4: Three different 2D slices lying on the y-axis of phase segmentations for flow in the $-z$ direction (before noise correction).

Two different noise reduction techniques were used whose results can be seen below. Figure 5.5 is obtained by applying the nearest-neighbour filter. The main advantage of this filter is that it will keep intact any small branch arteries that only reach widths of two or three voxels. This can best be seen in the figures referred to in Section 5.5.

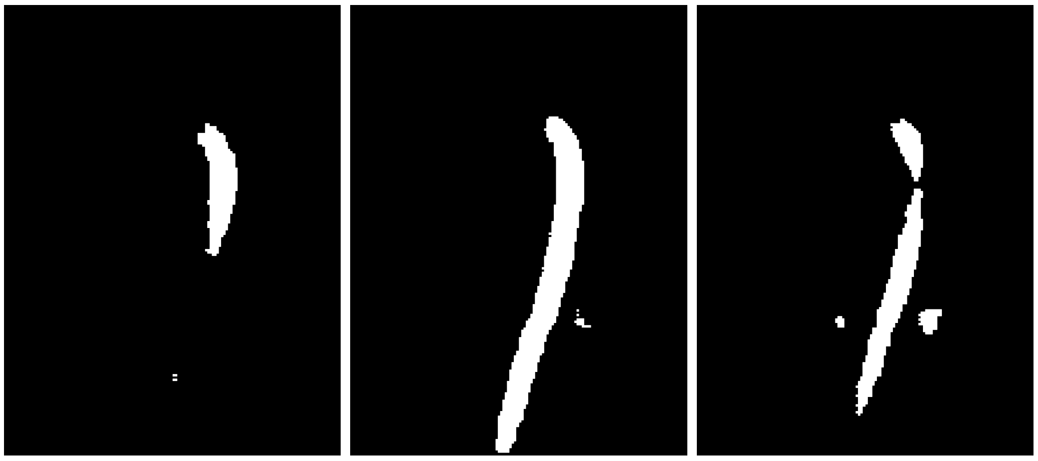


Figure 5.5: Three different 2D slices lying on the y-axis of phase segmentations for flow in the $-z$ direction after nearest-neighbor based noise reduction.

Figure 5.6 uses a 3D Gaussian filter with a width of 3 voxels to eliminate noise. The output of a Gaussian filter on a binary image gives values ranging between 0 and 1, so we threshold the result at 0.5 to obtain a noise-corrected binary segmentation. Although the results of the Gaussian filter are smoother, it has the effect of eliminating smaller arteries, which may be undesirable depending on the intended application of the segmentation.

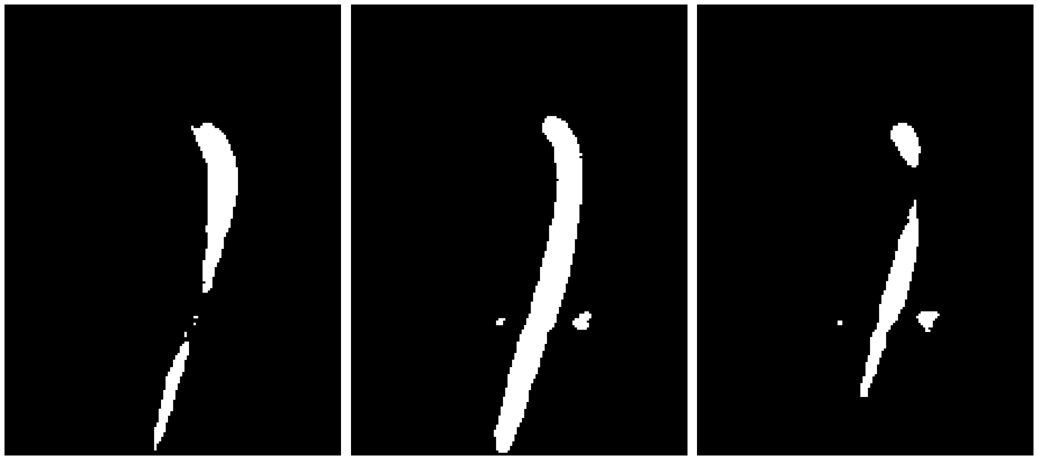


Figure 5.6: 2D slices of phase segmentations for flow in the –z direction after Gaussian filter noise reduction.

Unfortunately, neither method filters out nearby structures that have similar flow characteristics to the aorta. In particular, our algorithm was unable to differentiate sections of the pulmonary arteries from the aorta (Figure 5.7). A connectivity based algorithm could potentially extract the aorta from other structures, but in our case the resolution was too poor and the pulmonary artery appeared to be connected to the aorta in several areas. For this reason we use the GAC level set to extract the aorta. Scalar constant values used for the GAC (Equation 3.3)) were $\alpha = 3, \beta = 3.3, \gamma = 0$. Figure 5.7 shows mesh representations of the segmentation before and after GAC extraction.

Figure 5.8 shows an overlay of the segmentation after noise reduction and GAC extraction on the magnitude image. Notice in the right image, there is a large portion of the aorta clearly visible on the magnitude image that has not been segmented. It is clear there is still room for improvement upon this result, though it is a good starting point.

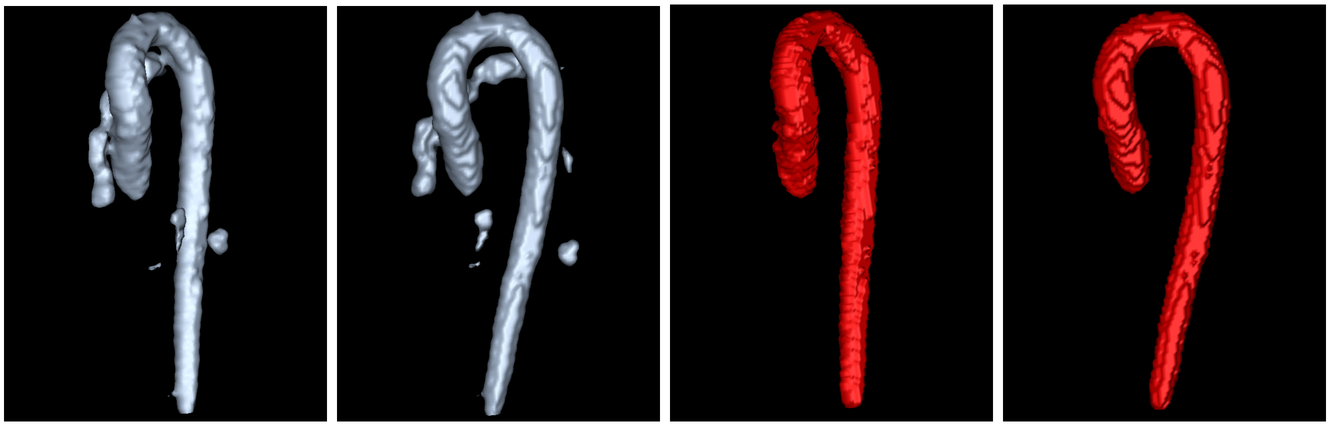


Figure 5.7: 3D mesh representations of phase segmentation with Gaussian filter noise reduction. The silver mesh represents the initial segmentation before GAC extraction. The two sets of images here are different views of the same segmentation. Notice that the pulmonary artery is the main source of error here. The red meshes show the corresponding output after GAC extraction.

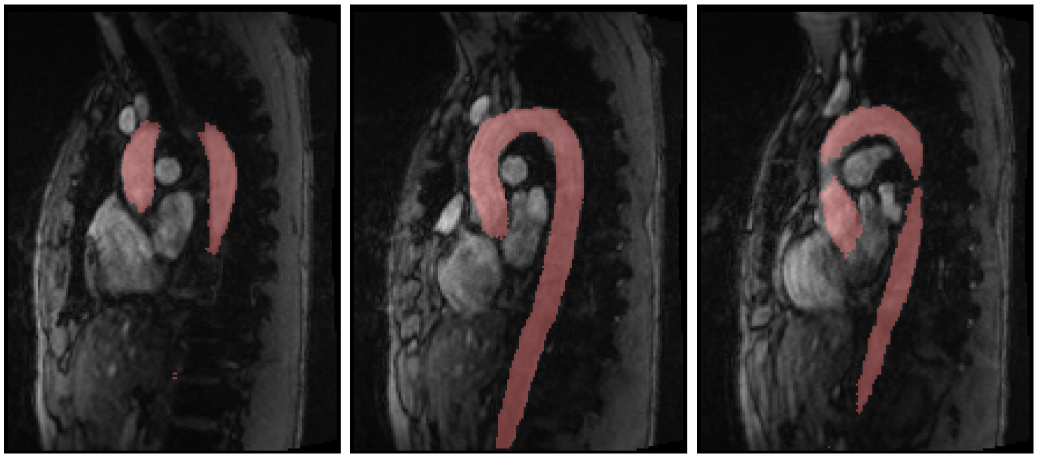


Figure 5.8: Final phase segmentation with nearest-neighbor noise reduction and GAC aorta extraction overlayed on magnitude image. The ascending aorta in the right image has not been fully segmented.

5.3 Curve-fitting pulsatile flow

Refining the ideas of the previous section, we would like to reduce the number of variables we need to calculate and keep track of, in order to produce a segmentation. Instead of calculating mean and standard deviations of flow speed at various intervals on the cardiac cycle, we fit the entire flow curves to Gaussian waveforms. The fitted Gaussians will have parameters σ and μ associated with them, with which we can define ranges of σ and μ that will be accepted into the segmentation.

Figure 5.9 shows the output of the initial CFS algorithm for different ranges of μ and σ. The CFS correctly segments the region of interest, but also incorrectly segments many voxels. Tightening restrictions on μ and σ eliminates some noise but the remaining segmentation is poor, particularly in the abdominal aorta. Similar to the previous phase segmentation, user defined seed points may be placed within the aorta as initializations of the GAC, which can be evolved using a high curvature force to prevent leakage into noisy voxels. Although at first this method seems less useful than the previous phase segmentation (producing initial output with less accuracy, and requiring a minimum of three seed points in the ascending and descending aorta, and the aortic arch), using the CFS as an initialization of the phase-magnitude segmentation method produces comparable results to the other phase-based segmentations. Figure 5.10 shows the results of the CFS after GAC extraction. Although some noise is still present, the segmentation conforms very well to the image edges.

The GDS segmentation algorithm, rather than fitting each flow curve to a Gaussian, took one Gaussian wave (whose μ and σ were chosen by trial and error) and compared each flow curve to this standard. The deviation from this standard can then be plotted. This produces a continuous image with a more uniform intensity throughout the aorta, and better contrast between aorta and background than the original phase and magnitude images. This suggests that the GDS may be implemented so as to enhance contrast of the original datasets, though this possibility is not further explored in this thesis. Thresholding the output of the GDS (which is shown in Figure 5.11) allows us to apply the noise correction and GAC extraction techniques of the previous section, which re-

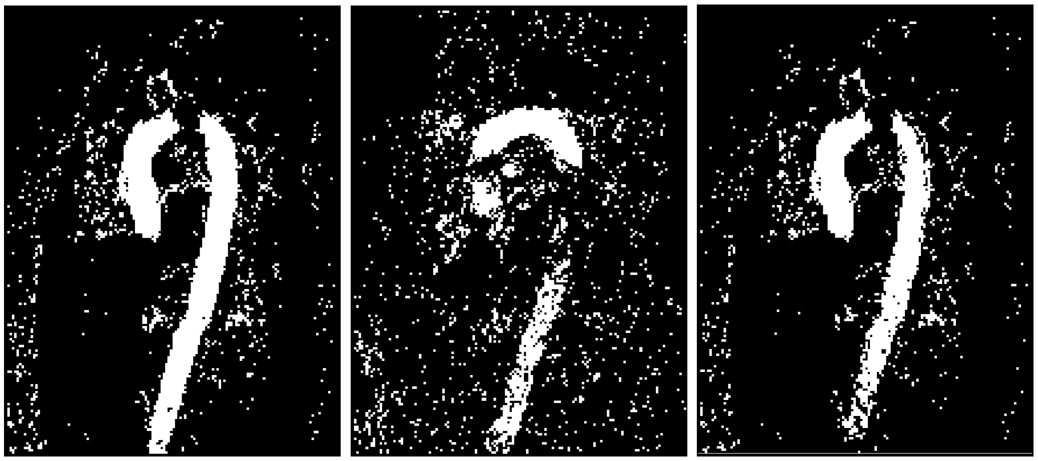


Figure 5.9: CFS output. Flow curves are fitted to Gaussian curves, and then included in the segmentation based on fitted values of μ and σ. Left and center images show segmentations in the z- and x- direction, respectively, with $7 \leq \mu, \sigma \leq 63$ ms being the criteria for segmentation. Right image shows a restriction of the criteria to $10 \leq \mu, \sigma \leq 49$ ms. Notice how the noise in the segmentation has not been altered much, but the segmentation begins to fail in the lower aorta.

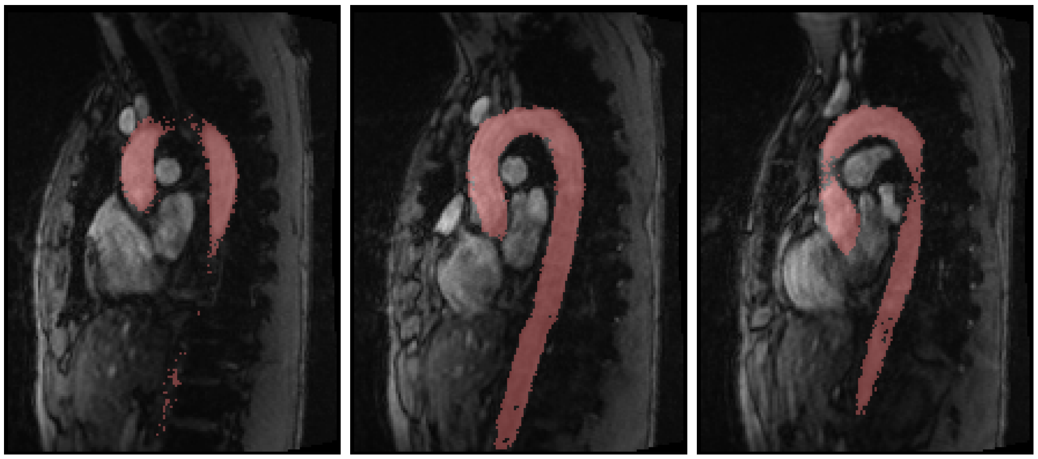


Figure 5.10: Final CFS segmentation after NN noise reduction and GAC aorta extraction overlayed on magnitude image. Through a visual inspection, the right image appears to show some improved accuracy in the ascending aorta compared to the previous phase-segmentation.

sults in the segmentation shown in Figure 5.12.

We see some small improvement in both the CFS and GDS segmentations over the phase-contrast segmentation in the segmentation of the ascending aorta. This is based on a visual inspection of both the 2D overlays as well as the 3D meshes of the segmentation results. Some holes appear in the lower aorta of these segmentations, but as we shall see in the following section, these are easily smoothed over by level set techniques. The important improvement upon the previous segmentation techniques is that we have increased the percentage of aortic volume that our segmentation covers without expanding the contour out too far. This will make refining the segmentation much easier.

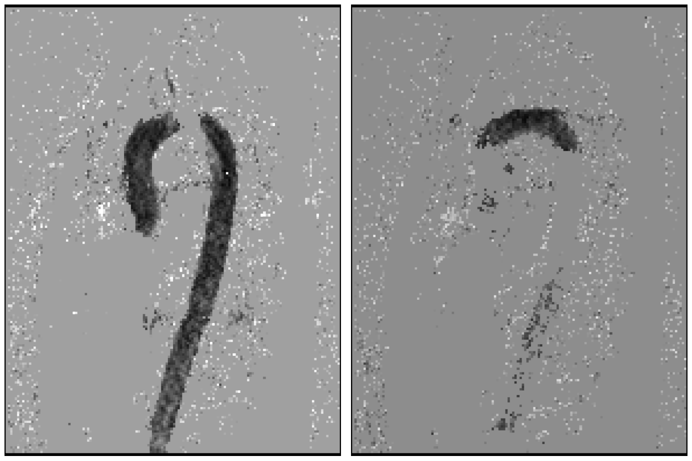


Figure 5.11: Output of the GDS algorithm before thresholding. A 2D slice of the GDS is shown for flow curves encoded in the z- and x-directions, respectively. Darker pixels represent a low deviation from the Gaussian waveform with $\mu = 28$ ms, $\sigma = 21$ ms.

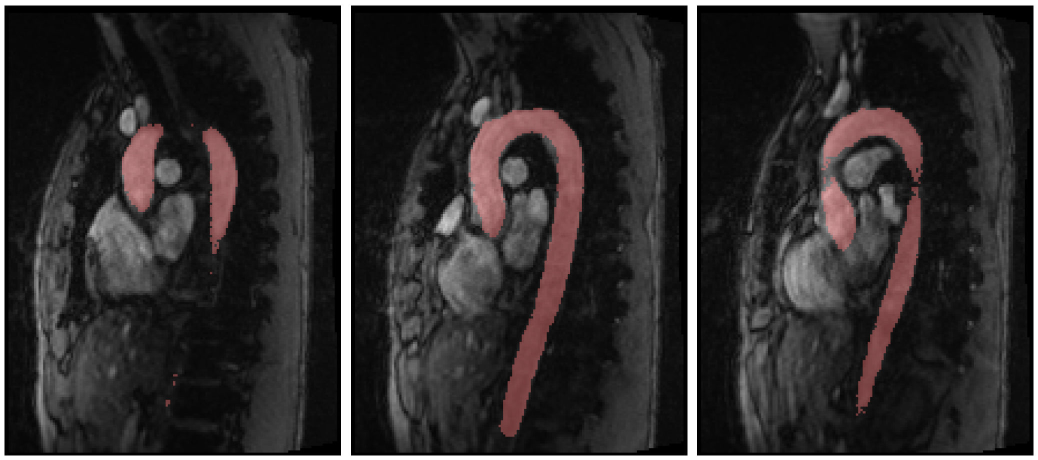


Figure 5.12: Final GDS segmentation after nearest-neighbor based noise reduction and GAC aorta extraction overlayed on the magnitude image. The segmentation produces a smoother output than the CFS, but it slightly underestimates the size of the abdominal aorta (based on a visual comparison with the magnitude image).

5.4 Phase-Magnitude Segmentation

Although the phase segmentations of the previous sections showed promising results, accuracy is generally poor in the ascending aortic arch. Conveniently, this is where the magnitude image has the best contrast, so we can take advantage of this by using the phase-based segmentations as initializations for another GAC level set on the magnitude image. Accuracy will be greatly improved and computation time reduced compared to the segmentations of section 5.1 since the contour is already near the desired region.

	α	β	γ	iterations
Phase	0.6	0.8	0	27
CFS	0.5	0.8	0	27
GDS	0.4	0.56	0	27

Table 5.3: GAC values used for PM Segmentations. The advection parameter γ, from Equation (3.3), is suppressed, since our contour is already close to the image edges. All algorithms required a maximum of 27 iterations to complete. The inherent noise of the phase and CFS techniques required a larger smoothing parameter than the GDS counterpart.

Table 5.3 shows the GAC parameter values and number of iterations for each initial level set used. An overlay of the PM segmentations on the magnitude image shows improved accuracy. A small propagation factor, α, prevents the contour from shrinking in areas where the magnitude image has poor contrast while expanding it in regions where the initial segmentation was poor. The curvature weighting eliminates any noise near boundaries. The level set reintroduces sub-voxel resolution for the segmentation and by eliminating the crutches of poor edge resolution in the magnitude image and potentially large processing times for the GAC, the final segmentation possesses all the strengths of both the phase-based and magnitude-based methods while also eliminating their largest weaknesses.

In practice, we found that the initial boundary was close enough to the desired location so that a maximum of 27 iterations of the GAC were needed. As a comparison, the GAC used in the magnitude segmentation, which was initialized from seed points, generally required many hundreds of iterations whose exact number varied with seed placement. The processing time for the additional 27 iterations was under 1.5 seconds on the CPU. As with the magnitude segmentations in section 3.2, it is important that the level set not undergo too many iterations, since convergence is not guaranteed and the contour may propagate outside the aorta. The results of the GAC initialized with phase, CFS and GDS segmentations are presented in Figures 5.13, 5.14, and 5.15, respectively, overlaid on the magnitude image.

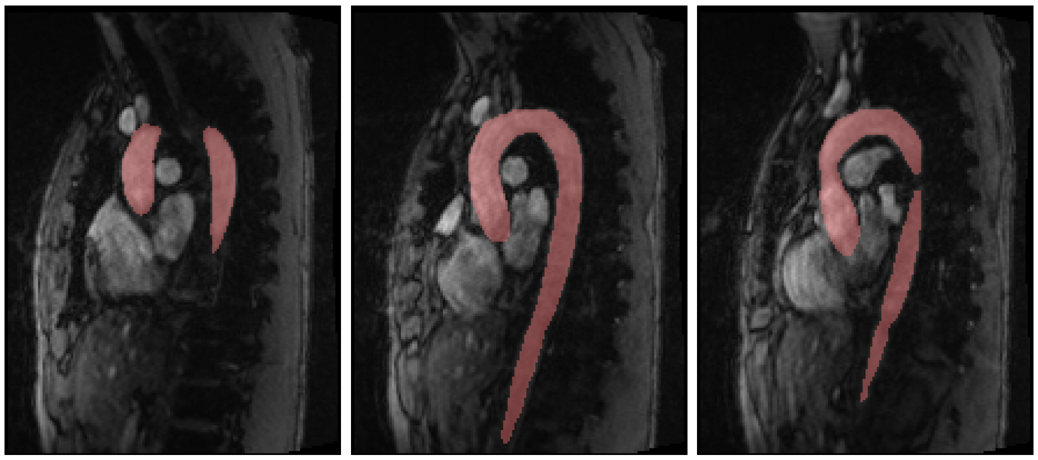


Figure 5.13: Overlay of PM segmentation with phase-based initialization on magnitude image.

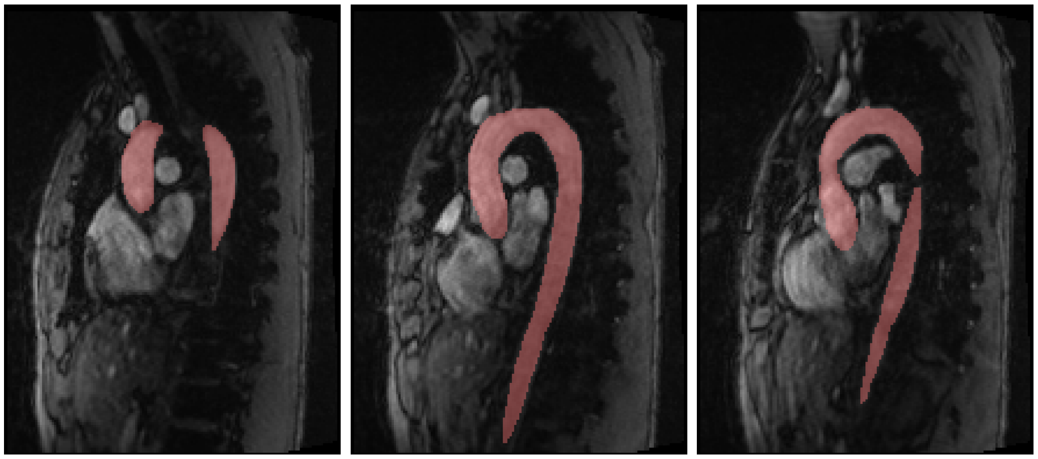


Figure 5.14: Overlay of PM segmentation with CFS-based initialization on magnitude image.

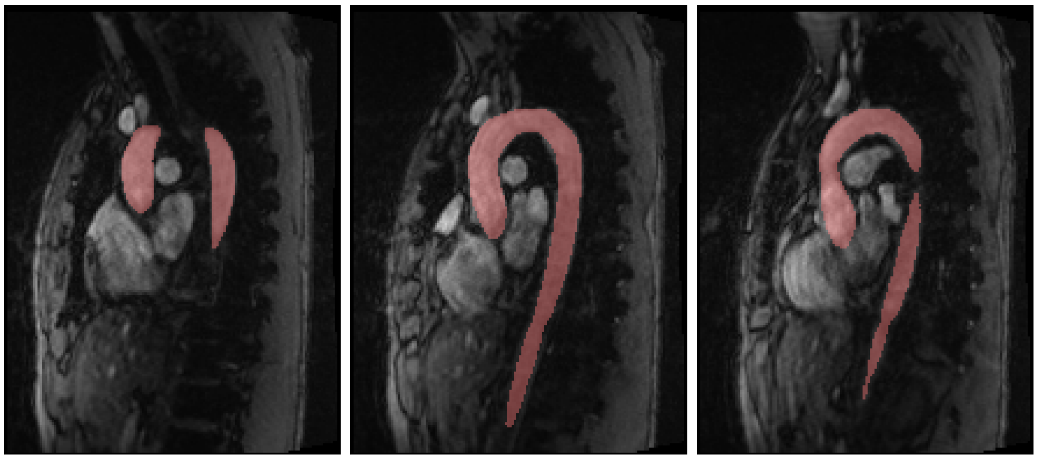


Figure 5.15: Overlay of PM segmentation with GDS-based initialization on magnitude image.

5.5 Summary of Segmentation Results

Accuracy of final segmentations can be judged by inspecting the results overlayed on the original magnitude images, and 3D mesh representations. We first present a side by side comparison of each segmentation overlayed on the magnitude image to show the strengths and weaknesses of each segmentation. Figures 5.16, 5.17, and 5.18 each show one sagittal magnitude slice of the aorta, with each segmentation overlayed on top. Since this only represent a small sample of possible 2D cross sections that make up the whole 3D volume, it is also of great interest that we produce and compare 3D meshes of each segmentation.

Below are some 3D mesh representations of each of the segmentations we have seen. The mesh representation of the magnitude segmentation of section 5.1 can be seen in Figure 5.19. Final PM meshes initialized by the phase, CFS and GDS methods are seen in Figures 5.20, 5.21, and 5.22, respectively. It is immediately clear that the magnitude segmentation initialized from seed points is of poor quality, and must be supplemented by additional filters or information. Poor image contrast leads to poor edge detection, which allows the level set to escape and 'bubble out' of the vessel. The phase segmentation is particularly well suited to the detection of branch arteries in the abdominal aorta when using the nearest neighbour-based noise filter. The CFS and GDS results are similar, although the GDS produces output with a narrower abdominal aorta. Unfortunately, it is impossible to exactly quantify the accuracy of each segmentation, since we have no standard segmentation to compare with. In the future, a comparison should be made with an expert manual tracing, or the segmentation routines should be performed on a test segmentation

All segmentations incorrectly segment a small portion of the superior vena cava (shown in Figure 5.23), which is difficult to differentiate in both magnitude and phase images. Through examination of meshes and the overlays of each result, it appears that the CFS-initialized PM segmentation produces the most accurate results, though this is difficult to quantify without some ideal segmentation to compare with.

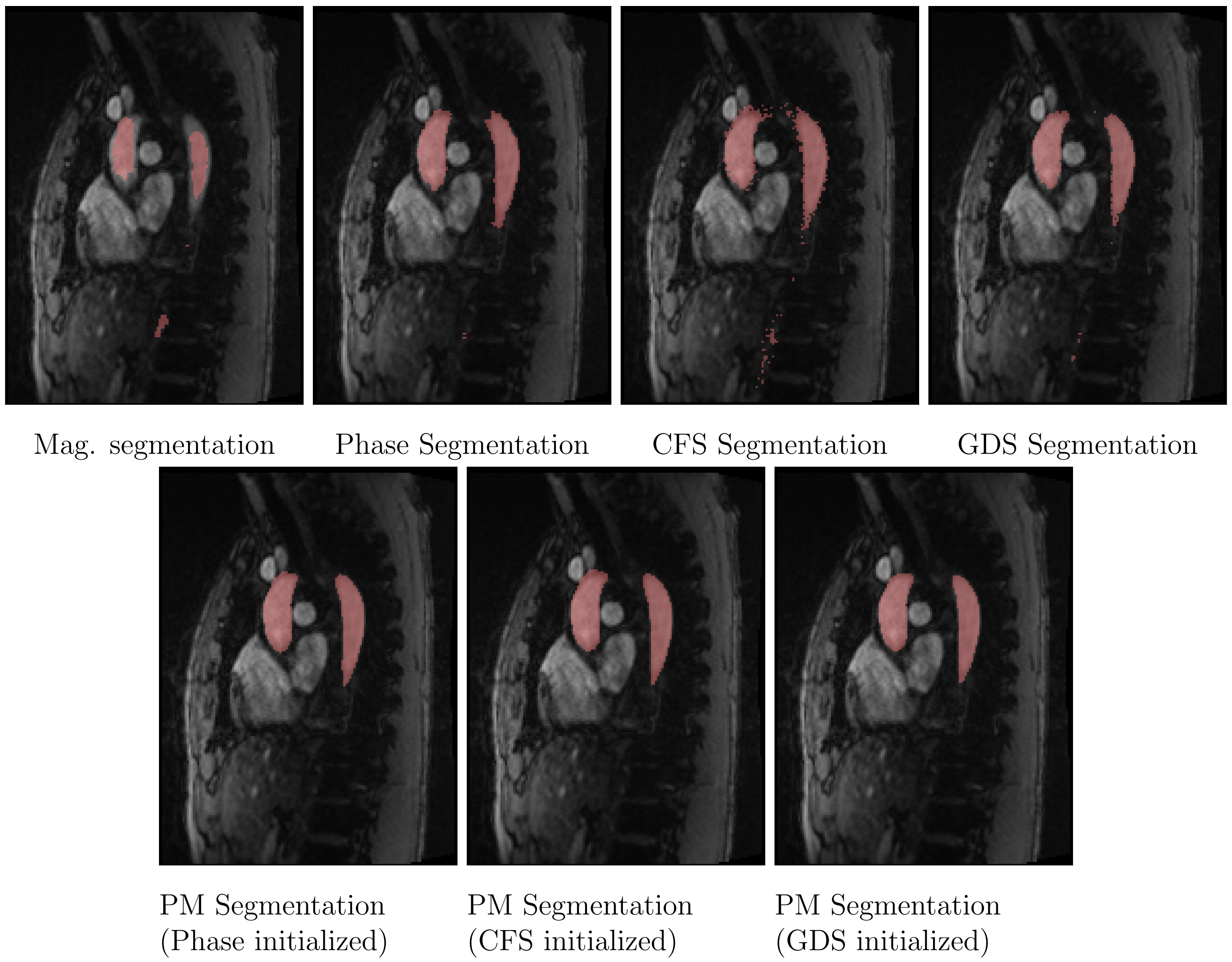

Mag. segmentation
Phase Segmentation
CFS Segmentation
GDS Segmentation
PM Segmentation (Phase initialized)
PM Segmentation (CFS initialized)
PM Segmentation (GDS initialized)

Figure 5.16: Comparison of segmentations overlayed on magnitude images. In this slice, there is not a huge difference in accuracy between all PM segmentation. Notice that even though the intial CFS segmentation is quite noisy on the edges of the aortic walls, the level set smooths the segmentation quite nicely in the CFS-intialized PM segmentation.

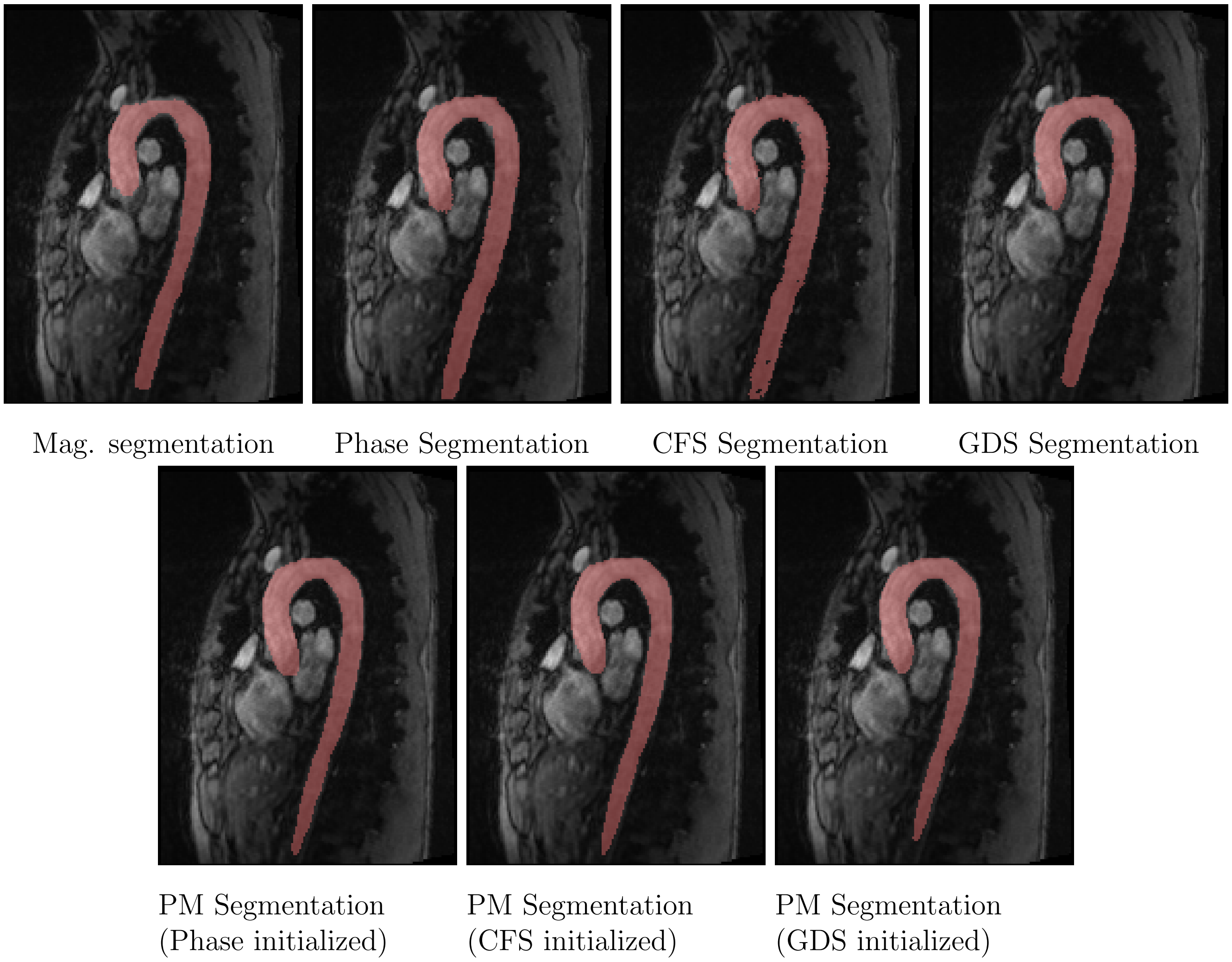

Mag. segmentation
Phase Segmentation
CFS Segmentation
GDS Segmentation
PM Segmentation (Phase initialized)
PM Segmentation (CFS initialized)
PM Segmentation (GDS initialized)

Figure 5.17: Comparison of segmentations overlayed on magnitude images. In this slice, all the segmentations are fairly similar. The magnitude segmentation still underestimates the aorta, however. The initial CFS segmentation has a few holes around the abdominal aorta, but the level set again removes this noise. The initial GDS segmentation underestimates the size of the abdominal aorta, and this carries over to the GDS-initialized PM segmentation. The phase-based initialization also slightly underestimates the abdominal aorta.

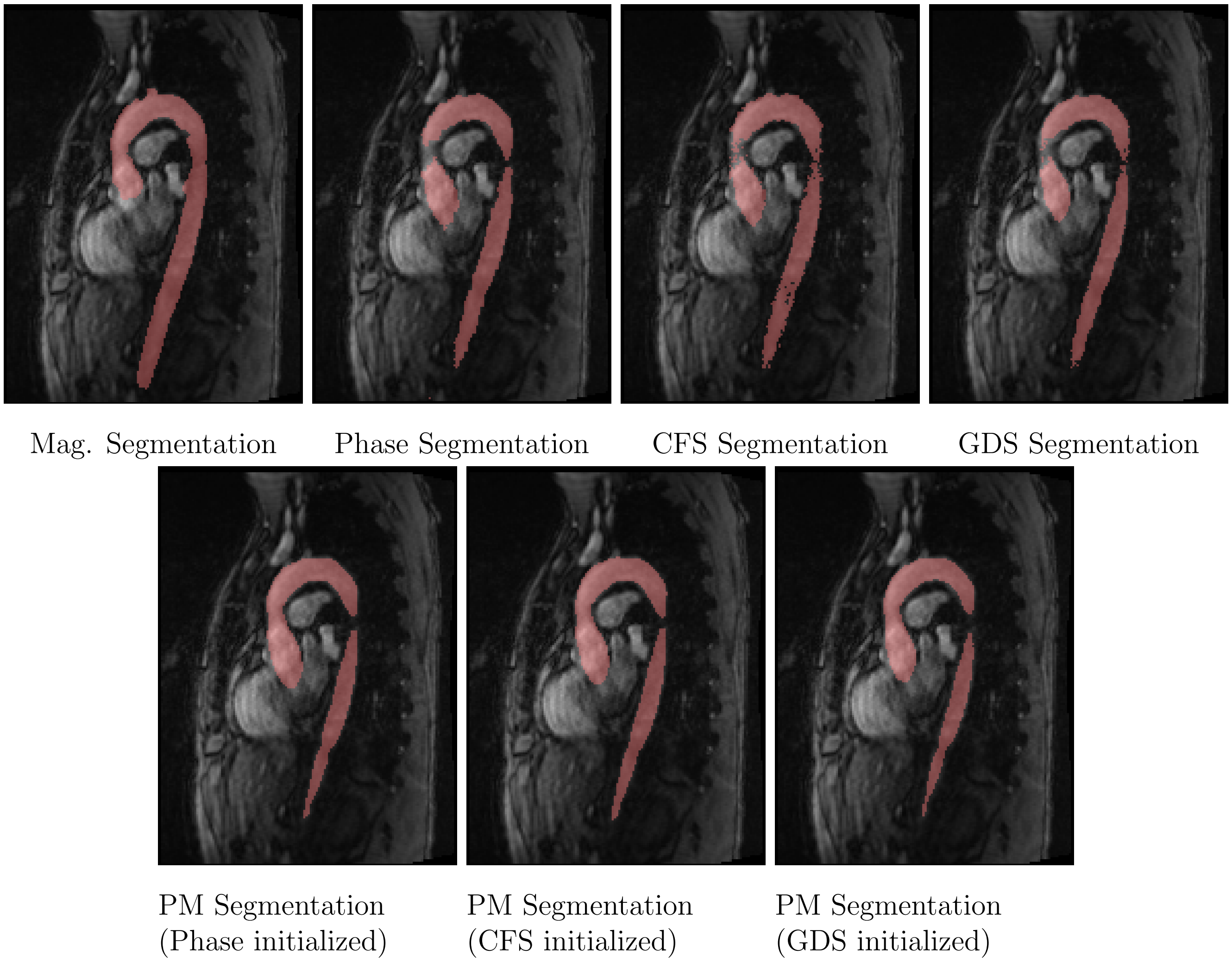

Mag. Segmentation
Phase Segmentation
CFS Segmentation
GDS Segmentation
PM Segmentation (Phase initialized)
PM Segmentation (CFS initialized)
PM Segmentation (GDS initialized)

Figure 5.18: Comparison of segmentations overlayed on magnitude images. In this slice, it is perhaps most clear how all the segmentations differ in accuracy. The contour bleeds out in several areas in the magnitude segmentation. Each initial phase, CFS and GDS segmentation does not segment a small portion of the ascending aorta, though the CFS does the best job. Again, some noise is present in the CFS segmentation. After applying the PM segmentations, we see that the phase and GDS initializations both start to pinch off in the lower aorta. The CFS segmentation is a bit smoother in the same area.

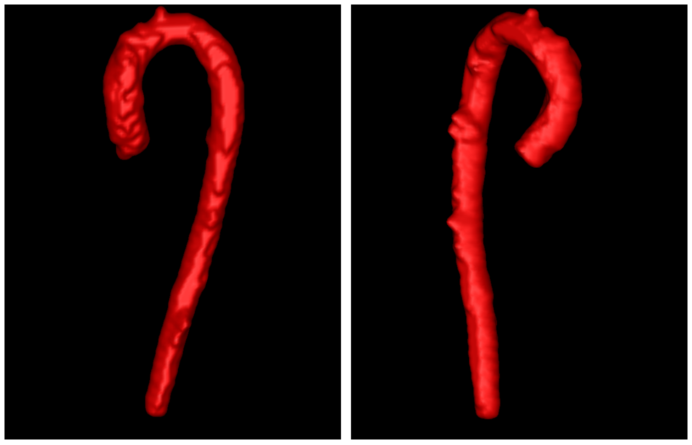


Figure 5.19: Two different views of a mesh representation of the segmentation produced by the GAC on the magnitude image with user-defined seed point initialization. Notice how the contour has expanded beyond image edges along the descending aortic arch.

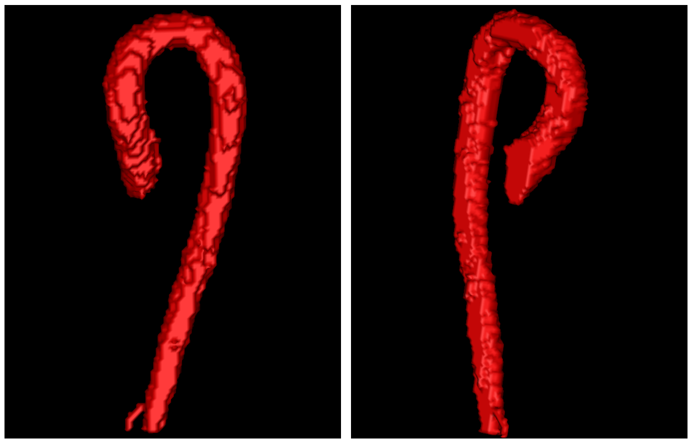


Figure 5.20: Two different views of a mesh representation of the segmentation produced by the phase algorithm with nearest neighbour based noise reduction and GAC extraction. An advantage of this technique is its ability to capture small branch arteries such as the one pictured at the bottom of the figure here. This is due to nearest-neighbour noise filter effectively having a radius of 1 voxel.

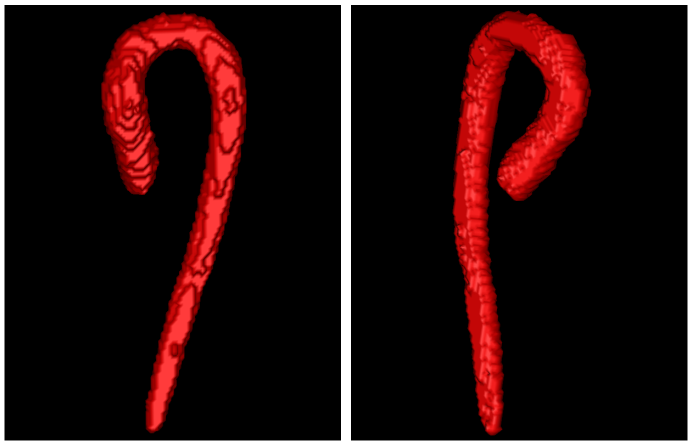


Figure 5.21: Two different views of the mesh representation of the segmentation produced by the PM algorithm with CFS-based initialization.

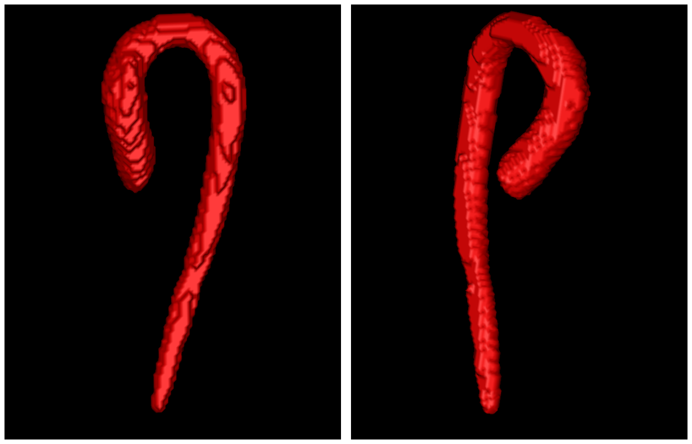


Figure 5.22: Two different views of the mesh representation of the segmentation produced by the PM algorithm with GDS-based initialization.

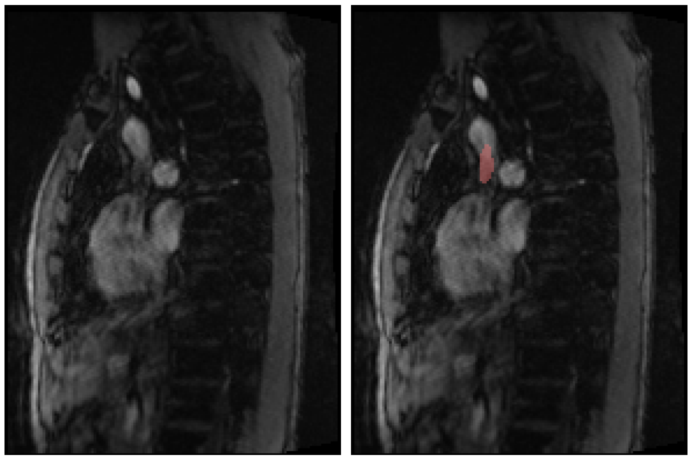


Figure 5.23: Left: A 2D slice of the magnitude image. The superior vena cava can easily be seen in the top left section of the chest cavity. Right: The CFS algorithm wrongly segments a small portion of the superior vena cava.

5.6 Applications

Now that we have some aortic segmentations to work with, we can apply the visualization techniques discussed in Section 3.6 to the segmented region of data. We choose to use the CFS segmentation as our data region, as it appears to be the most accurate. From here, we can set velocity data outside of the segmented region to zero, so it does not interfere with any interpolation calculations. We are now able to define analysis planes on the segmented aorta so that we can emit particles at points of our choosing for particle tracking simulations and flow visualization.

5.6.1 Analysis planes

We are able to test our algorithm for the automated definition of analysis planes along the aorta. Recall that a point and a vector are all that are necessary to define the analysis plane. Since we want to define our analysis planes so that they are perpendicular to the longitudinal axis of the vessel, the normal vectors we want are just those vectors parallel to the centerline. Therefore, finding and fitting the points of the centerline gives us all the information we need to define an analysis plane anywhere on the aorta. Figures 5.24 and 5.25 show $\vec{n}$ along the entire center line from different viewing angles. Three different centerline fits are shown, where the neighbourhood size that determines how many points are inside the fit changes. This corresponds directly to changing the size of $\mathbf{M}$ in Equation (3.19). Since the centerline can only be locally linearly fit, we must determine the number of points that yields the truest centerline fit. A neighbourhood of $6\times3\times6$ voxels appeared to produce the most accurate results.

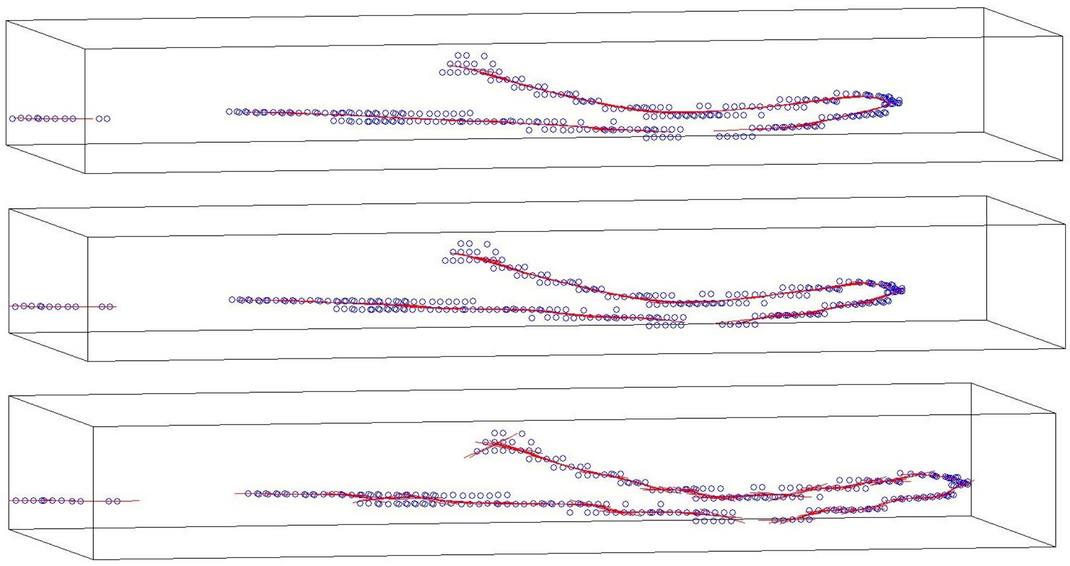


Figure 5.24: In this image, circles represent voxels that lie on the centerline, as determined by the SDF algorithm in Section 3.6.2. The red lines represent the normal vector $\vec{n}$ calculated at each point on the center line. On top, a neighbourhood size of 10×3×10 voxels was chosen for principal component analysis. The middle and bottom images have neighbourhood sizes of 6×3×6 and 3×3×3, respectively. With a smaller neighbourhood, the centerline fit becomes bumpy and disjointed, while a larger neighbourhood moves the centerline from its true position on the aortic arch.

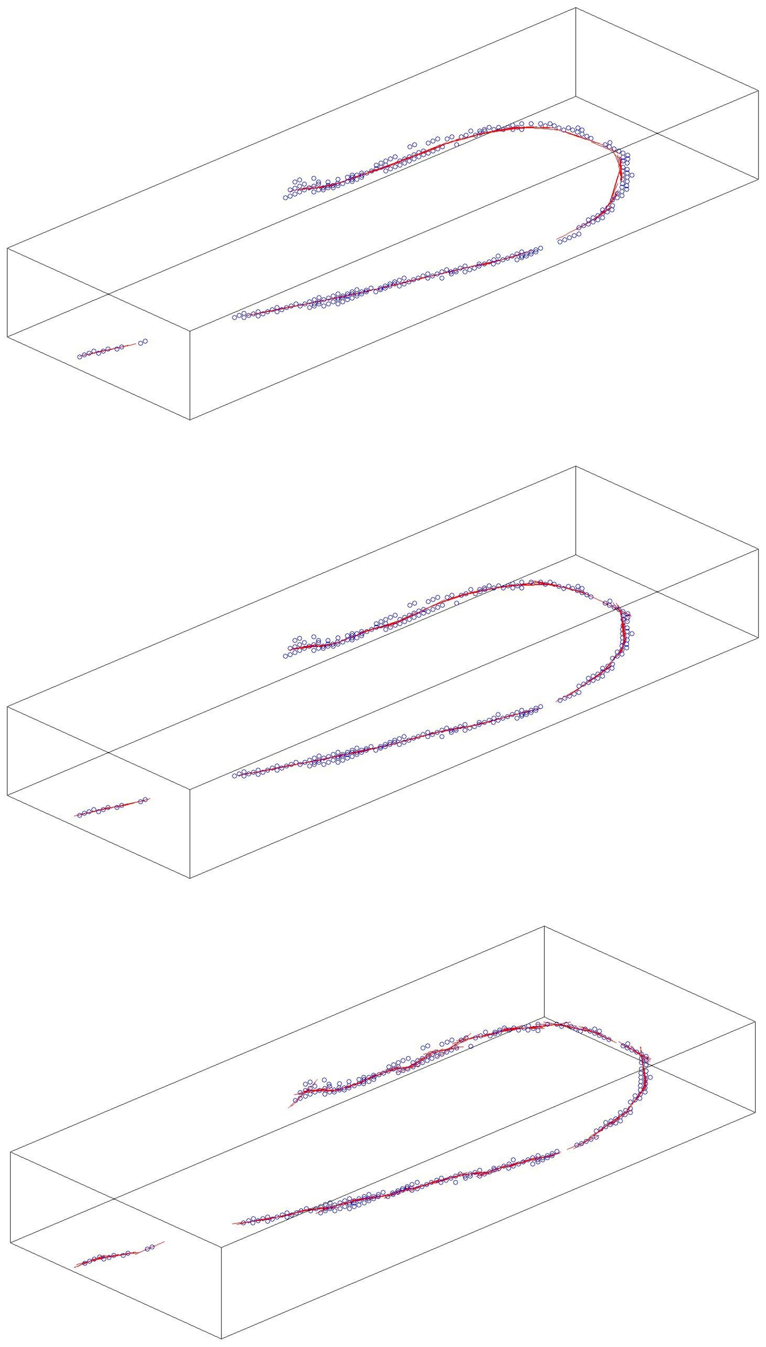


Figure 5.25: Another view of the centerline fit. Notice how the calculated centerline lies slightly below the voxels in the top image. This is the result of using too large of a neighbourhood in the PCA calculation, so that the local mean in Equation (3.20) falls outside of the true centerline and skews the results of the fit.

5.6.2 Visualization and particle-tracing

Figure 5.26 shows a semi-transparent mesh of the CFS segmentation of Section 5.4. Into this mesh, we can emit our particles and track them over time, colour-coding each trace by its speed. We first present some streamline representations of flow. Recall that these representations show the instantaneous velocity field at a point in time in the heart cycle. Figure 5.27 shows the streamline representation for the aorta at peak systole. This can be compared to the streamline representation during diastole (Figure 5.28), where the flow velocity is small. Although the streamlines can be traced for an arbitrary amount of time, it is inevitable that some traces will leave the segmentation region after a period of time. This is not necessarily the result of a poor segmentation, and could be attributable to poor temporal/spatial resolution of the data. Figure 5.27 and Figure 5.28 are traced for the same amount of time. More streamlines leave the segmentation during diastole since the flow is less directed and particles diffuse throughout the volume.

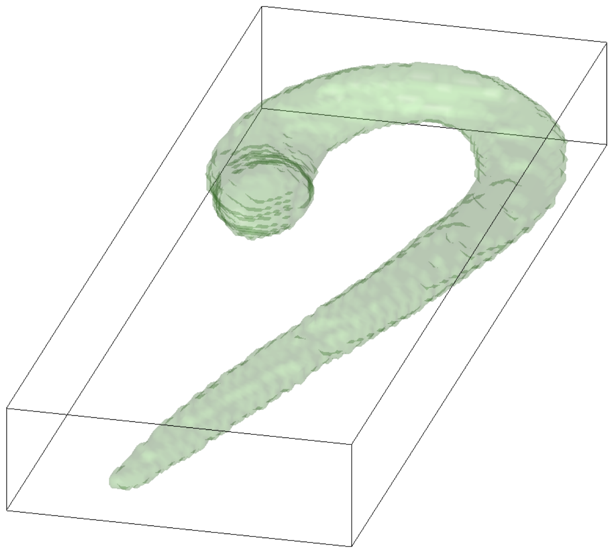


Figure 5.26: A semi-transparent mesh of the CFS segmentation.

The pathline representation of flow has a velocity field that varies with time, unlike the stream line respresentation. While streamlines are useful for an overall picture of the flow field at cer-

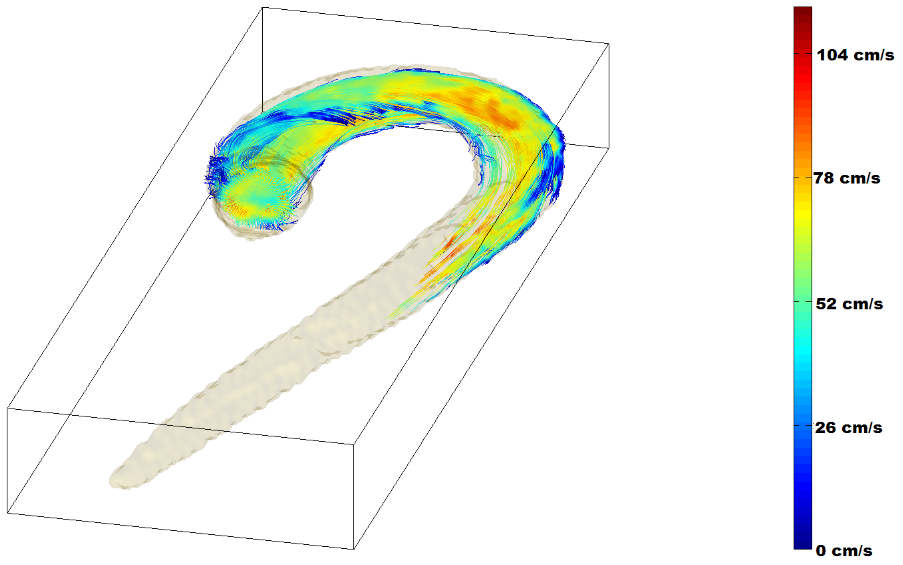


Figure 5.27: During systole, blood flows more quickly through the aorta. Here is a representation of the velocity field at an instant during systole. Streamlines are followed from the ascending to descending aorta. As streamlines leave the segmentation they are stopped. Additional emitter planes can be defined at any point to map a more complete velocity field if necessary.

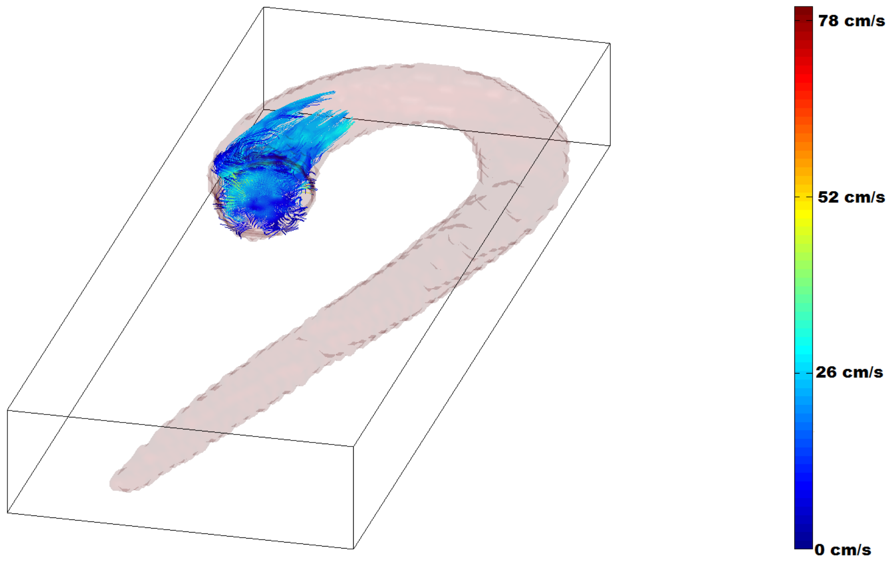


Figure 5.28: Streamline representation at early diastole. Streamlines were followed for the same amount of time as in Figure 5.27. Many more streamlines leave the aorta sooner due to the diffusion of virtual particles in the low-velocity stream.

tain points in time, pathlines are used for simulations of flow. It is also possible to animate the pathlines, to get a sense of the flow characteristics in a way that a static image cannot convey. Figures 5.29 and 5.30 show the path that virtual particles would take over a heart cycle, emitted at two different planes in the aortic vessel. Notice in Figure 5.31, when particles are traced after the first heart cycle, the diffusion of particles makes it difficult to make out the pattern of flow. This representation becomes more interesting when rendered into an animation where pathlines are continuously emitted from the plane over time, to simulate the real-time flow of blood through the aorta.

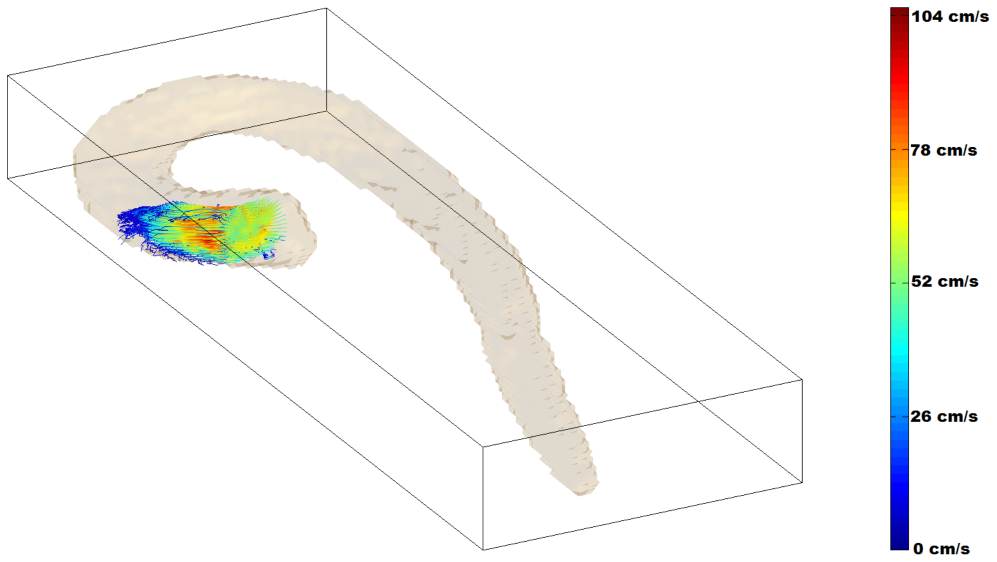


Figure 5.29: Particles are emitted in the ascending aorta and traced for one heart cycle.

Some evidence of the characteristic helical flow patterns identified in the ascending aorta in healthy patients can also be found in our data in Figure 5.32 [53]. This is a good sign that our segmentation, and the steps we have taken to represent the flow information are effective. It is also encouraging that not many stream or pathlines leave the segmentation region.

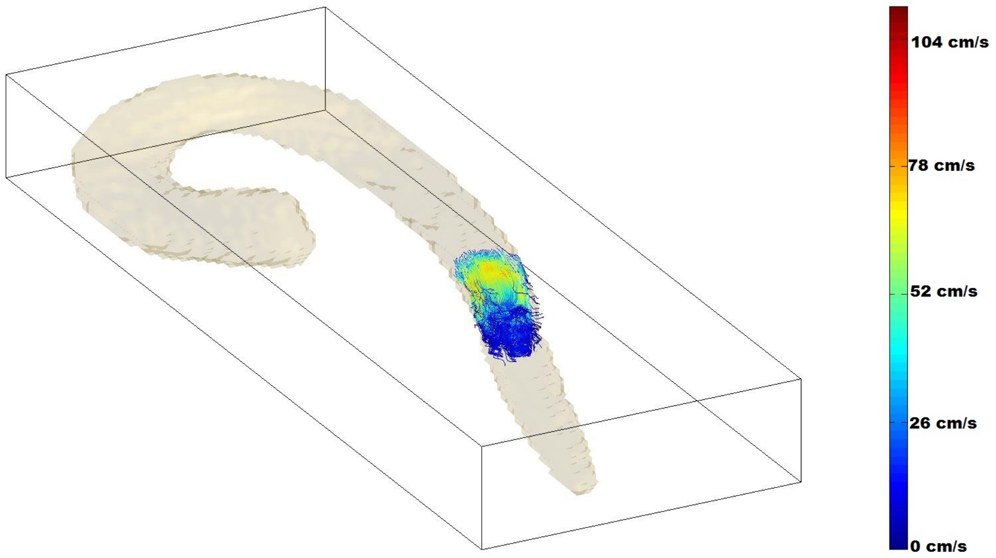


Figure 5.30: Particles are emitted in the abdominal aorta and traced for one heart cycle.

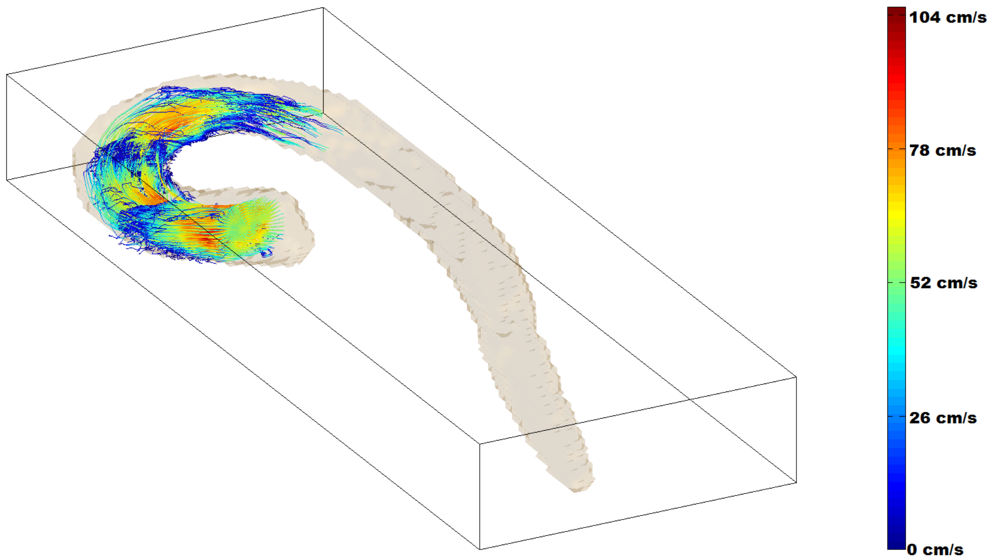


Figure 5.31: Particles are emitted as in Figure 5.29, but this time they are tracked for three heart cycles. Notice that during diastole, when flow velocity is low, the particles motion is more random and they diffuse. More pathlines exit the segmentation the longer the simulation runs.

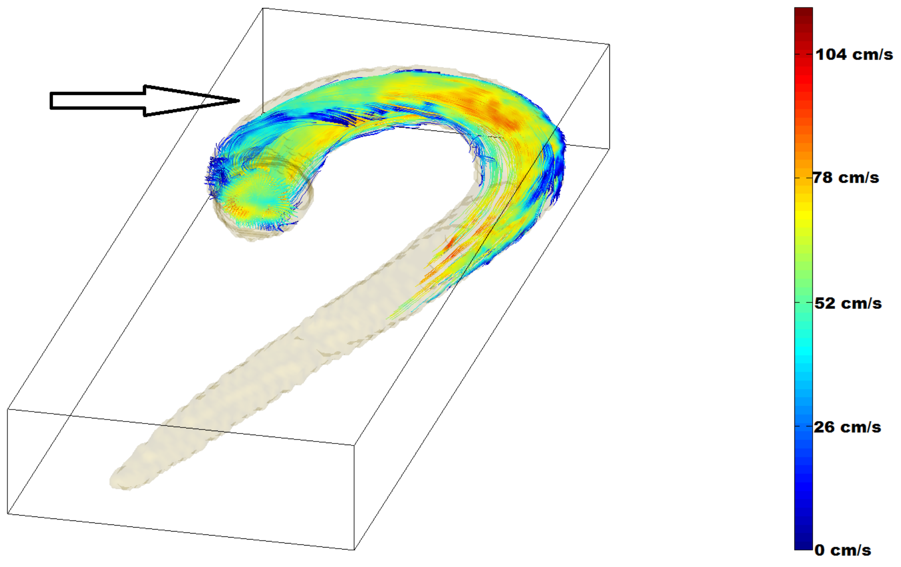


Figure 5.32: In the streamline respresentation during peak systole, some evidence of helical flow can be seen in the ascending aorta.

5.7 GPU optimization

Processing time is always an important consideration with medical imaging. Less processing time can directly translate into less waiting times for patients and faster diagnoses. A GPU can perform many calculations in parallel which can vastly decrease computation time compared to the CPU. All of the segmentation methods above rely on performing the same computations many times on multiple flow curves, so it is of interest to see if and to what extent the GPU can decrease computation times. Table 5.4 shows computation times of all initial segmentation algorithms and noise reduction, where applicable. Since the GAC extraction computation time is highly dependent on the location and number of seed points defined by the user, its computation time is not listed, though generally this task could be accomplished in under one minute. Calculations for the CPU were performed in MATLAB, which makes use of CPU multithreading [54]. Algorithms were specifically coded with this in mind, so that the fastest possible CPU computational times could be acheived. Out of interest, we also show the computational time for the phase-segmentation method where no form of multithreading or parallel computing is used.

	CPU	GPU	Noise Reduction
Phase (no multi-threading)	430	-	-
Phase	9.8	5.2	Nearest Neighbour
Phase	2.5	1.3	Gaussian Filter
CFS	1380	0.50	-
GDS	0.23	0.10	-

Table 5.4: Computation times for initial segmentations (seconds). Noise reduction times are included in listed values where applicable. No noise reduction techniques were used before GAC extraction for the CFS and GDS methods. Extraction times are not listed because of the high dependence of computation time on user input. It was found that a GAC extraction could generally be accomplished in under one minute on the CPU.

The method that most benefits from the GPU is by far the CFS. Each GPU thread is assigned one voxel whose flow data undergoes the Gaussian fit. These calculations are rapidly performed in parallel in under one second. The amount of information processed using this method is not too large to be suitable for calculation on a CPU. Though the accuracy of the initial segmentation appears poor, noise correction and GAC extraction produced results that seem to outperform the other segmentation algorithms presented here. The other algorithms perform well enough with CPU multi-threading that GPU calculations are not particularly beneficial. This is because of the simplicity of the calculations involved, which the CPU can speedily perform. However, it is important to remember that in the future as imaging resolution increases and data sets become larger, GPUs will greatly outperform any CPU, as the sheer number of computations must necessarily increase. Over 90% of total computation times were attributable solely to noise reduction in phase, CFS and GDS segmentations and/or the GAC extraction.

Unfortunately, the GAC extraction technique used to separate the aorta from other vessels that were incorrectly segmented is a bottleneck for speed in each algorithm. The program must stop and wait for the user to initialize the GAC extraction by entering seed points into the aorta. It would be desirable to introduce some way of automatically reducing noise in a way that would require no user initialization. Our GPU-based segmentation has shown that once a solution to this problem is found, it will become increasingly possible that an application could be developed that performs complex computations that automatically produce accurate 3D results in real-time.

Chapter 6

Discussion and Conclusions

In this thesis, we have explored several methods for the segmentation of the aorta. First we illustrated that the level set, which has been previously shown by others to be a useful segmentation tool, is by itself insufficient for an accurate segmentation on magnitude image data. We proposed that for the level set to be useful, some initial 3D contour that is already close to the desired segmentation region must be found.

We devised new algorithms which use phase-contrast data to determine whether voxels belong inside the segmented region. These algorithms calculated mean flow speed and standard deviations of flow speed during different time frames of the heart cycle, or compared flow curves to Gaussian waveforms. Based on the results of these calculations, voxels were included or excluded from the segmentation, depending on whether the calculated parameters fell inside an expected range. The segmentations that used phase-contrast data were then refined by noise correction techniques, and the aorta was extracted from any other structures the algorithms erroneously segmented, through a level set initialized by seed points defined by the user. Afterwards, these segmentations served as initializations to another level set on the magnitude image.

Our best segmentation appeared to be the CFS initialized algorithm, which makes use of a GPU to rapidly perform curve fits on all voxels in parallel. Other segmentation algorithms are shown to have similar performance on the CPU in terms of run-time, but only when some form of parallel computing across multiple CPU cores is used, and the algorithms are not computationally expensive. However, these come at the cost of somewhat less accurate results. The efficiency of the GPU versus the inefficiency of the CPU for the CFS algorithm (see Table 5.4) stems from the relative complexity of the algorithm relative to the other phase-based segmentation algorithms presented in this thesis. (Fitting curves to pulsatile flow is an iterative process where one must minimize the difference between the data and fit, which is more computationally complex than simply taking the mean or standard deviation of some data, for example.) This suggests that any segmentation algorithm (on large 4D data sets) that is sufficiently complex will only be practical on the GPU. In fact, it can be argued that as demands for visualization and analysis of large 4D data sets increase with imaging technology and capabilities, GPU computing will become increasingly practical, and perhaps necessary for image processing.

Our proposed automated technique for which analysis planes are defined on the aortic volume appears to function well, and its definition constructed in such a way as to make use of the GPU architecture needed for massively parallel, rapid calculation. Our visualization of the aortic flow appears to validate the segmentation and the method of interpolation we have used, as some evidence of characteristic flow patterns normally found in healthy patients are also found in our data. Unfortunately, the work in this thesis was done entirely with the MR images of one healthy patient. In the future, the segmentation algorithms should be tested across multiple data sets on healthy and non-healthy patients, to confirm that the algorithms are robust.

4D flow visualization has yet to become commonly accepted in routine clinical cardiovascualar MR practice, mainly because of the time and experience currently needed for appropriate acquisition and analysis of the large 4D data sets. Additionally, irregular heart beat or breathing during data acquisition tend to result in suboptimal data [55]. More automated methods for flow visualization and retrospective quantification of data would therefore be extremely beneficial towards introducing 4D flow applications within a clinical setting. New software tools and algorithms should also be developed. For example, there is no standardized way of defining the analysis planes of section 3.6.2 [55]. A standard definition of these planes in routinely acquired 4D velocity data would be useful for easy post-analysis of flow.

The parallelization of a particle trace presents some challenges that this work has not been able to explore fully. It is true that all the particles of a trace simulation evolve under the same equation (Equation (3.12)), making at least the calculation of instantaneous particle velocities and positions a trivial task to parallelize across the threads of a GPU. In a time-varying field however, velocity information must always be updated at each successive iteration, slowing the potential speed up of the application. Some solutions to this problem involve pre-partitioning of the 4D domain into regions that approximate the flow directions. The preprocessing of such a method is quite expensive, and Yu et. al [56] have reported that less than one second of rendering required approximately 15 minutes of preprocessing time.

Another approach to the problem is the use of partitioning and load balancing. This involves the division of the velocity information into subdomains and the assignment of these subdomains to the GPU cores in such a way as to reduce overall computation and communication between cores. A common approach is geometry-based partitioning, where a core will monitor and update all the particles within a volume of the simulation, only transmitting new information to other cores when the particles leave its assigned subdomain [57]. These methods should be further studied to optimize the visualization of 4D flow data sets. Additionally, some algorithms that calculate flow parameters like wall shear stress may be devised in the future, which would also benefit from GPU computing in that the pressure or stress at each point on the aortic wall could be calculated in parallel. However, the usefulness of such an algorithm may be questionable, as some studies have found that calculations based on MR flow data generally underestimate true wall shear stress, and sometimes have very large errors associated with them [58].

Bibliography

[1] J. Earls, V. Ho, T. Foo, E. Castillo, S. Flamm. Cardiac MRI: Recent Progress and Continued Challenges. Journal of Magnetic Resonance Imaging 2002; 16: 111-127.

[2] P. Chai, R. Mohiaddin. How we perform cardiovascular magnetic resonance flow assessment using phase-contrast velocity mapping. Journal of Cardiovascular Magnetic Resonance 2005; 7(4):705-716.

[3] P. Beerbaum, H. Korperich, P. Barth, H. Esdorn, J. Gieseke, H. Meyer. Noninvasive quantification of left-to-right shunt in pediatric patients: phase-contrast cine magnetic resonance imaging compared with invasive oximetry. Circulation 2001; 103(20):2476-2482.

[4] D. Didier. Assessment of valve disease: qualitative and quantitative. Magn Reson Imaging Clin N Am 2003; 11(1):115-134.

[5] P.D. Gatehouse, J. Keegan, L.A. Crowe, S. Masood, R.H. Mohiaddin,K.F. Kreitner, D.N. Firmin. Applications of phase-contrast flow and velocity imaging in cardiovascular MRI. Eur Radiol 2005; 15(10):2172-2184.

[6] S.R. Underwood, D.N. Firmin, R.S. Rees, D.B. Longmore. Magnetic resonance velocity mapping. Clin Phys Physiol Meas 1990; 11(Suppl A):37-43.

[7] A.F. Stalder, M.F. Russe, A. Frydrychowicz, J. Bock, J. Hennig, M. Markl. Quantitative 2D and 3D phase contrast MRI: optimized analysis of blood flow and vessel wall parameters. Magn Reson Med 2008; 60(5):1218-1231.

[8] T. Ebbers, L. Wigström, A.F. Bolger, B. Wranne, M. Karlsson. Noninvasive measurement of time-varying three-dimensional relative pressure fields within the human heart. J Biomech Eng 2002; 124(3):288-293.

[9] G.Z. Yang, P.J. Kilner, N.B. Wood, S.R. Underwood, D.N. Firmin. Computation of flow pressure fields from magnetic resonance velocity mapping. Magn Reson Med 1996; 36(4):520-526.

[10] J.N. Oshinski, D.N. Ku, S. Mukundan Jr, F. Loth, R.I. Pettigrew. Determination of wall shear stress in the aorta with the use of MR phase velocity mapping. J Magn Reson Imaging 1995; 5(6):640-647.

[11] S. Oyre, W.P. Paaske, S. Ringgaard, S. Kozerke, M. Erlandsen, P. Boesiger, E.M. Pedersen. Automatic accurate non-invasive quantitation of blood flow, cross-sectional vessel area, and wall shear stress by modelling of magnetic resonance velocity data. Eur J Vasc Endovasc Surg 1998; 16(6):517-524.

[12] V. Caselles, J.-M. Morel, G. Sapiro, A. Tannenbaum, editors. 1998. Special Issue on Partial Differential Equations and Geometry-Driven Diffusion in Image Processing and Analysis.

IEEE Transactions on Image Processing 1998; 7:269.

[13] M. Nielsen, P. Johansen, O. F. Olsen, J.Weickert, editors. Scale-Space Theories in Computer Vision. Lecture Notes in Compuer Science 1999; 1682.

[14] M. Lynch, O. Ghita, and P. F. Whelan. Segmentation of the left ventricle of the heart in 3-D+t MRI data using an optimized nonrigid temporal model. IEEE Trans. Med. Imag. 2008; 27(2):195-203.

[15] L. Wigström, L. Sjöqvist, B. Wranne. Temporally Resolved 3D Phase-Contrast Imaging. Magnetic Resonance in Medicine 1996; 36(5):802.

[16] J.E. Cates, A.E. Lefohn, R.T. Whitaker. GIST: an interactive, GPU-based level set segmentation tool for 3D medical images. Medical Image Analysis 2004; 8:217-231.

[17] G. Pratx, L, Xing. GPU Computing in medical physics: A review. Medical Physics - New York - Institute of Physics 2011; 38(5):2685 -2697.

[18] Z.-P. Lhiang, P. C. Lauterbur. Principles of Magnetic Resonance Imaging. New Jersey. Wiley-IEEE Press 2000.

[19] E.M. Purcell, H.C. Torrey, R. V. Pound. Resonance Absorption by Nuclear Magnetic Moments in a Solid. Physical Review 1946; 69(1,2): 37-38

[20] F. Bloch, W.W. Hansen, M. Packard. The Nuclear Induction Experiment. Physical Review 1946; 70:474-485

[21] Puddephat, M. Principles of magnetic resonance imaging; c2014 [cited 2014 April 27]. Available from: `http://www.mikepuddephat.com/page/1603/Principles-of-magnetic-resonance-imaging`

[22] Nuclear magnetic Spectroscopy. University of California, Davis; [cited 2014 April 27]. Available from: `http://chemwiki.ucdavis.edu/Physical_Chemistry/Spectroscopy/Magnetic_Resonance_Spectroscopies/Nuclear_Magnetic_Resonance/Nuclear_Magnetic_Resonance_II`

[23] Hyperpolarized Noble Gas MRI Laboratory. Harvard Medical School; c2006 [cited 2014 April 27]. Available from: `http://www.spl.harvard.edu/archive/HypX/theory1.html`

[24] S. Day. Basic principles of NMR. [cited 2014 April 27]. Available from `http://www2.warwick.ac.uk/fac/sci/physics/research/condensedmatt/imr_cdt/students/stephen_day/relaxation/`.

[25] M. Markl. Velocity Encoding and Flow Imaging. 2006 [cited 2013 November 28]. Available from: `http://ee-classes.usc.edu/ee591/library/Markl-FlowImaging.pdf`.

[26] Paul A. Yushkevich, Joseph Piven, Heather Cody Hazlett, Rachel Gimpel Smith, Sean Ho, James C. Gee, and Guido Gerig. User-guided 3D active contour segmentation of anatomical structures: Significantly improved efficiency and reliability. Neuroimage 2006; 31(3):1116-1128.

[27] S. Osher, J. Sethian. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. Journal of Computational Physics 1988; 79:12-48.

[28] M. Roberts, J. Packer, M. Costa Sousa, J. Ross Mitchell. A Work-Efficient GPU Algorithm for Level Set Segmentation. High Performance Graphics 2010.

[29] V. Caselles, R. Kimmel, and G. Sapiro. Geodesic active contours. International Journal on Computer Vision 1997; 22(1):61-97.

[30] L. Ibáñez, W. Schroeder, L. Ng, J. Cates and the Insight Software Consortium. The ITK Software Guide. Kitware Inc., 2005.

[31] L.M. de Heer, R.P. Budde, W.P. Mali, A.M. de Vos, L.A. van Herwerden, J. Kluin. Aortic root dimension changes during systole and diastole: evaluation with ECG-gated multidetector row computed tomography. Int J Cardiovasc Imaging 2011; 27(8):1195-1204.

[32] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. IEEE Transactions on Pattern Analysis Machine Intelligence 1990; 12:629-639.

[33] R. Deriche. Fast algorithms for low level vision. IEEE Transactions on Pattern Analysis and Machine Intelligence 1990; 12(1):78-87.

[34] R. Deriche. Recursively implementing the gaussian and its derivatives. Technical Report, INRIA, 1993.

[35] K.H. Fraser, S. Meagher, J.R. Blake, W.J. Easson, P.R. Hoskins. Characterization of an Abdominal Aortic Velocity Waveform in Patients with Abdominal Aortic Aneurysm. Ultrasound in Medicine & Biology 2008; 34(1):73-80.

[36] M.H. Buonocore. Visualizing blood flow patterns using streamlines, arrows, and particle paths. Magn. Reson. Med. 1998; 40(2):210-226.

[37] H.G. Bogren, M. H. Buonocore. 4D magnetic resonance velocity mapping of blood flow patterns in the aorta in young vs. elderly normal subjecs. J Magn. Reson. Imaging 1999; 10(5):861-869.

[38] S. Kozerke, J.M. Hasenkam, E.M. Pedersen, P. Boesiger. Visualization of flow patterns distal to aortic valve prostheses in humans using a fast approach for cine 3D velocity mapping. J Magn. Reson. Imaging 2001; 13(5):690-698.

[39] L. Wigström, T. Ebbers, A. Fyrenius, M. Karlsson, J. Engvall, B. Wranne, A.F. Bolger. Particle trace visualization of intracardiac flow using time-resolved 3D phase contrast MRI. Magn. Reson. Med. 1999; 41(4):793-799.

[40] J.C. Butcher, 1987. The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods. Wiley-Interscience, New York, NY.

[41] C.P. Cheng, D. Parker, C.A. Taylor. Quantification of Wall Shear Stress in Large Blood Vessels Using Lagrangian Interpolation Functions with Cine Phase-Contrast Magnetic Resonance Imaging. Annals of Biomedical Engineering 2002; 30:1020-1032.

[42] Tutorvista; [cited 2014 Feb 27]. Available from `http://image.tutorvista.com/content/feed/tvcs/cross4.PNG`.

[43] L.N. Trefethen, D. Bau III. Numerical linear algebra. Philadelphia: Society for Industrial and Applied Mathematics, 1997.

[44] G. E. Moore. Cramming More Components onto Integrated Circuits. Electronics 1965; 114-117.

[45] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T.J. Purcell. A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum 2008; 26(1):21-51.

[46] J. Owens, M. Houston, D. Leubke, S. Green, J. Stone, J. Phillips. GPU Computing. Proc. IEEE 2008; 96(5): 879-899.

[47] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov. Parallel computing experiences with CUDA. IEEE MICRO 2008; 28(4):13-27.

[48] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. 2008; 68(10):1370-1380.

[49] W. Magro, P. Peterson, and S. Shah. Hyper-Threading Technology: Impact on Compute-Intensive Workloads. Intel Technology Journal 2002.

[50] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings 1967. 30:483485.

[51] Tesla C2050/C2070 GPU Computing Processor [Internet]. NVIDIA Corporation; c2010 [cited 2013 October 29]. Available from: `http://www.nvidia.ca/docs/IO/43395/NV\_DS\_Tesla\_C2050\_C2070\_jul10\_lores.pdf`.

[52] Ark Intel Xeon Processor E5-1560 (12M Cache, 3.2GHz, 0.0 GT/s Intel QPI [Internet]. Intel; [cited 2013 October 29]. Available from: `http://ark.intel.com/products/64601`

[53] P.J. Kilner, G.Z. Yang, R.H. Mohiaddin, D.N. Firmin, D.B. Longmore. Helical retrograde secondary flow patterns in the aortic arch studied by three-directional magnetic resonance velocity mapping. Journal of the American Heart Association 1993; 88:2235-2247.

[54] MATLAB multicore. Mathworks; c1994-2013 [cited 2013 October 29]. Available from: `http://www.mathworks.com/discovery/matlab-multicore.html`

[55] M. Markl, P. J. Kilner, T. Ebbers. Comprehensive 4D velocity mapping of the heart and great vessels by cardiovascular magnetic resonance. Journal of Cardiovascular Magnetic Resonance 2011; 13:7.

[56] H. Yu, C. Wang, K.-L. Ma. Parallel hierarchical visualization of large time-varying 3d vector fields. SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing. New York, NY: ACM, 2007, pp. 1-12.

[57] M. J. Berger, S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. IEEE Trans. Comput. 1987; 36(5):570-580.

[58] S. Petersson, P. Dyverfeldt, T. Ebbers. Assessment of the accuracy of MRI wall shear stress estimation using numerical simulations. Journal of Magnetic Resonance Imaging 2012; 36(1):128-138.

[59] C. Humphries. University of California, 2000 [cited 2014 Feb 27]. Available from `http://www.neuro.mcw.edu/~chumphri/matlab/writeanalyze.m`.

Appendix A

Computer code for GPU-accelerated segmentations and applications

A.1 Phase-contrast segmentation

The following code is the segmentation and noise reduction of one direction-encoded phase image. The code follows the same structure as the other direction-encoded phase-images, with the parameters restricted according to Table 5.2.

This script uses a function 'writeanalyze', which written by Colin Humphries at the University of California [59]. This function allows MATLAB to write .hdr format images.

```
% Read magnitude and phase-contrast image, then rescale
A = analyze75read('MM');
P1 = analyze75read('1P');
sA=size(A);
input1 = zeros(sA(1),sA(2),sA(3)/15, 15);
for i = 1:sA(3)
    if mod(i,38) == 0
        v = 38;
        input1(:,:,v, floor(i/38)) = P1(:,:,i);
    else
        input1(:,:,mod(i,38), floor(i/38)+1) = P1(:,:,i);
    end
end



% Write arrays to GPU


Gvec1a = gpuArray(input1);
Gvec1 = gpuArray(input1(:,:,:,1:5));
G1 = gpuArray(input1(:,:,:,1:3));
Gvecend1 = gpuArray(input1(:,:,:,7:15));

% Calculations of mean and std
minGvec1 = min(Gvec1,[],4);
stdGvec1 = std(Gvec1,0,4);
meanG1 = mean(G1,4);
stdG1 = std(G1,0,4);
meanGvec1a = mean(Gvec1a,4);
stdGvec1a = std(Gvec1a,0,4);
meanGvecend1 = mean(Gvecend1,4);
stdGvecend1 = std(Gvecend1,0,4);



minGvec1 = minGvec1(:,:,:) < 1600; %All elements meeting this requirement are set to 1
dummy = stdGvec1(:,:,:) < 600; %All elements less than 600 are 1
dummy2 = stdGvec1(:,:,:) > 80; %All elements more than 80 are 1
stdGvec1 = dummy .* dummy2; %All elements less than 600 and more than 80 are 1

dummy = Gvec1a(:,:,:,2) < 2075;


dummy2 = Gvec1a(:,:,:,1:4) < 2021;
dummy3 = Gvec1a(:,:,:,2:5) < 2021;
```

```
dummy2 = sum(dummy2,4);
dummy3 = sum(dummy3,4);

dummy2 = dummy2 >= 4;
dummy3 = dummy3 >= 4;

%This statement checks all conditions above
Goutput = minGvec1.*stdGvec1.*dummy.*dummy2 + minGvec1.*stdGvec1.*dummy.*dummy3;

dummy = meanG1 < 1750;
dummy2 = meanG1 > 1330;
stdG1 = stdG1 < 300;
dummy3 = G1(:,:,:,3) > G1(:,:,:,2) -50;

%This statement checks new conditions
Goutput = Goutput + minGvec1.*dummy.*dummy2.*dummy3.*stdG1;
Goutput = Goutput >= 1;

dummy = stdGvec1a >= 500;
Goutput = Goutput - dummy;
Goutput = Goutput >=1;


dummy = meanGvec1a < 2200;
dummy2 = meanGvec1a > 1900;
dummy3 = stdGvec1a < 150;

Goutput = Goutput - dummy.*dummy2.*dummy3;
Goutput = Goutput >=1;

dummy = abs(meanGvecend1 -2048) > 150;
dummy2 = stdGvecend1 > 200;
dummy3 = abs(meanGvecend1 -1848) > 80;
```

```
Goutput = Goutput - Goutput.*dummy.*dummy3 - Goutput.*dummy2.*dummy3;
Goutput = Goutput >=1; % Last if

Goutput = Goutput.*255;

% Write images
outimg = gather(Goutput);

% Need to switch x and y axes for image to write properly with writeanalyze
for i = 1:192
outimg2(:,i,:) = outimg(i,:,:);
end

writeanalyze(outimg2, [144 192 38],'test', [1.771 1.771 3.3]);

% Apply gaussian noise reduction
h = fspecial3('average', [4 4 2]);

outimg3 = imfilter(outimg2,h);

%Thresholding the filter
outimg3 = (outimg3>=110).*255;

writeanalyze(outimg3, [144 192 38],'test-gauss', [1.771 1.771 3.3]);

% Apply NN noise-reduction
outimg2 = GPUnoise(outimg, 5);
B = zeros(144,192,38);
for i = 1:192
B(:,i,:) = outimg2(i,:,:);
end
writeanalyze(B, [144 192 38],'test-NN', [1.771 1.771 3.3]);
```

A.2 GDS Segmentation with GPU in MATLAB

```
%Read in segmentation and the phase-contrast images
A = analyze75read('finalcfs');
P1 = analyze75read('1P');
P2 = analyze75read('2P');
P3 = analyze75read('3P');
s=size(P1);

%Give GPU the data
A = gpuArray(A);
P1 = gpuArray(P1);
P2 = gpuArray(P2);
P3 = gpuArray(P3);

input1 = double(reshape(P1, s(1), s(2), s(3)/15, 15));
input2 = double(reshape(P2, s(1), s(2), s(3)/15, 15));
input3 = double(reshape(P3, s(1), s(2), s(3)/15, 15));

%Define error function
input1 = input1 - 2048;
input2 = input2 - 2048;
input3 = input3 - 2048;

%Use absolute values of flow curves
input1 = abs(input1);
input2 = abs(input2);
input3 = abs(input3);

%Define the gaussian parameters
mu = 2;
sigma = 1.5;
```

```
%Normalize flow curves

norm1 = max(input1,[],4);

norm2 = max(input2,[],4);

norm3 = max(input3,[],4);


input1 = input1./norm1;

input2 = input2./norm2;

input3 = input3./norm3;


%Define Gaussian we wish to compare flow curves to

x = 1:15;

gauss = (1/(sqrt(2*pi))*sigma)*exp(-(0.5*((mu-x)/sigma)^2));


gauss = repmat(gauss, [144 192 38]);

gauss = reshape(gauss, 144, 192, 38, 15);


Chi1 = (input1 - gauss)^2;

Chi2 = (input2 - gauss)^2;

Chi3 = (input3 - gauss)^2;


%Chi can be imaged in each direction

%Thresholding to be done interactively in ITK-SNAP
```

A.3 GPU kernel for CFS segmentation

```
#include <iostream>

#include <cuda.h>

#include <stdlib.h>

#include <ctime> //need only for random number generator

#include <fstream>

#include <vector>

#include <time.h>
```

```
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

//Start definition of GPU kernel
__global__ void CUDAGaussFit(float * a, float * b, int count)
{
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if(idx < count/15)
{
//Initial guesses for mu, sigma is 2 and 0.8
double mu = 2;
double sigma = 0.8;
double param = 2*M_PI; //define 2*pi
double result = sqrt(param); //define sqrt(2*pi)
double sdsigma;
double sdmu;
double sdsigma2;
double sdmu2;
double sdsigmamu;
double elem1;
double elem2;

for (int it = 0; it<10; it++)
{


sdsigma = 0;
sdmu = 0;
sdsigma2 = 0;
sdmu2 = 0;
sdsigmamu = 0;

//Assume data is already normalized (gaussian lies above +y axis, goes to y = 0
// at infinite)

for (int i = 0; i < 15; i++)   //Calculate fit for 15 point curve
{

double param2 = -(i+1-mu)*(i+1-mu)/(2*sigma*sigma);
double result2 = exp(param2); //define exp(-((x-mu)^2)/2*sigma^2)

//sum of dsigmas
sdsigma += -1/(result*sigma*sigma) * result2 + 1/(result*sigma) * result2 * (i+1-mu)
* (i+1-mu) / (sigma*sigma*sigma);

//sum of dmus
sdmu += 1/(result*sigma) * result2 * (i+1-mu) / (sigma * sigma);

//sum of squares of dsigma
sdsigma2 += (-1/(result*sigma*sigma)* result2 + 1/(result*sigma) * result2 * (i+1-mu)
* (i+1-mu) / (sigma*sigma*sigma)) * (-1/(result*sigma*sigma)* result2 + 1/(result*sigma)
* result2 * (i+1-mu) * (i+1-mu) / (sigma*sigma*sigma));

//sum of squares of dmu
sdmu2 += (1/(result*sigma) * result2 * (i+1-mu) / (sigma * sigma))*(1/(result*sigma)
* result2 * (i+1-mu) / (sigma * sigma));

//sum of mu*sigma
sdsigmamu += (-1/(result*sigma*sigma)* result2 + 1/(result*sigma) * result2 * (i+1-mu)
* (i+1-mu)/(sigma*sigma*sigma))*(1/(result*sigma) * result2 * (i+1-mu) / (sigma * sigma));

}

elem1 = 0;
elem2 = 0;
// Calculate fit for 15 point flow curve (should be generalized in the future)
for (int i = 0; i < 15; i++)
```

```
{
double param2 = -(i+1-mu)*(i+1-mu)/(2*sigma*sigma);
double result2 = exp(param2); //define exp(-((x-mu)^2)/2*sigma^2)
elem1 += ((-1/(result*sigma*sigma)* result2 + 1/(result*sigma) * result2 * (i+1-mu)
* (i+1-mu) / (sigma*sigma*sigma))*sdmu2 - (1/(result*sigma) * result2 * (i+1-mu)
/ (sigma * sigma))*sdsigmamu)*(a[idx+i*count/15] - 1/(result*sigma)*result2);

elem2 += (-(-1/(result*sigma*sigma)* result2 + 1/(result*sigma) * result2 * (i+1-mu)
 * (i+1-mu) / (sigma*sigma*sigma))*sdsigmamu + (1/(result*sigma) * result2 * (i+1-mu)
/ (sigma * sigma))*sdsigma2)*(a[idx+i*count/15] - 1/(result*sigma)*result2);
}

elem1 = elem1/(sdsigma2*sdmu2 - sdsigmamu*sdsigmamu) + sigma;
elem2 = elem2/(sdsigma2*sdmu2 - sdsigmamu*sdsigmamu) + mu;

mu = elem2;
sigma = elem1;

b[2*idx] = elem1; //store sigmas in b matrix
b[2*idx+1] = elem2; //store mus in b matrix

//one loop complete!
}
}
}

\\end of kernel

\\main program to read in data, call kernel

int main()
{

srand(time(NULL));
```

```
//number of data points in 4D image - should be generalized in future
int count = 15*38*192*144;

//b will store the sigmas and mus, so the number of stored data points is divided
//by the time resolution, multiplied by two
float *b = new float[((count/15)*2)];

float *d_a;
float *d_b;



clock_t begin, end;
double time_spent;
begin = clock();

////////////////////////////////////////////////////////////////////////////// READ FILE



//Create a dynamic array to hold the values
vector<float> numbers;

//Create an input file stream
ifstream in("input1.txt",ios::in);

/*
            As long as we haven't reached the end of the file, keep reading entries.
*/

float number;  //Variable to hold each number as it is read

        //Read number using the extraction (>>) operator
        while (in >> number) {
//Add the number to the end of the array
numbers.push_back(number);
}

//Close the file stream
in.close();

/*
    Now, the vector<float> object "numbers" contains both the array of numbers,
            and its length (the number count from the file).
*/

float *a = new float[count];
for(int i = 0; i<count; i++)
a[i] = numbers[i];

///////////////////////////////////////////////////////////////////////////////////////

end = clock();
time_spent = (double)(end-begin) / CLOCKS_PER_SEC;

//Print out time elapsed while reading in data to CPU
cout << "Elapsed time is " << time_spent << " milliseconds." << endl;

for(int i = 0; i<((count/15)*2); i++)
{
b[i] = 0;
}

cudaMalloc(&d_a, sizeof(float)*count);            // Allocate memory on the GPU
cudaMalloc(&d_b, sizeof(float)*(count/15)*2);

cudaMemcpy(d_a, a, sizeof(float) * count, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * (count/15)*2, cudaMemcpyHostToDevice);
```

```
float memsettime;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);

CUDAGaussFit<<<count/256 + 1,256>>>(d_a, d_b, count); //Run CUDA Kernel

cudaEventRecord(stop,0);

cudaThreadSynchronize();

cudaEventElapsedTime(&memsettime,start,stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);

//Print out elapsed time while running fitting kernel
cout << "Elapsed time is " << memsettime << " milliseconds." << endl;

cudaMemcpy(a, d_a, sizeof(float) * count, cudaMemcpyDeviceToHost);
cudaMemcpy(b, d_b, sizeof(float) * (count/15) * 2, cudaMemcpyDeviceToHost);

//Write array to text file for import of data into MATLAB///////////////////

ofstream fl("output1short.txt");

//Check data for infinite or NAN values (which MATLAB will not be able to read)
for (int k=0;k<2*(count/15);k++)
{
if (isfinite(b[k])==1)
{
fl<<b[k]<<endl;
}
if (isfinite(b[k])==0) //check for NAN
{
fl<<0<<endl;
}
}
fl.close();
/////////////////////////////////////////////


cudaFree(d_a); //Free memory
cudaFree(d_b);
free(a);
free(b);


return 0;
}
```

A.4 MATLAB code for the definition of emitter planes

```
%Returns a set of points within a plane centered near user-defined 'point1'
function [X,Y,Z] = planeemitter(point1)

%Read centerline and segmentation images
A = analyze75read('cfscenterline');
s=size(A);
B = analyze75read('finalcfs');

%Search for nearest center line points
ind = 1:(size(A,1)*size(A,2)*size(A,3));
[I,J,K] = ind2sub(size(A),ind);
dist = (I-point1(1)).^2 + (J-point1(2)).^2 + (K-point1(3)).^2;
```

```
distmat = zeros(size(A));
distmat(ind(:)) = dist(:);
W = A.*distmat;
W(~W) = nan; %Set points outside of segmentation to NAN

[~,IND] = min(W(:));
[aIND,bIND,cIND] = ind2sub(size(W),IND);
point1 = [aIND bIND cIND];

%Calculate local linear fit

mat1 = (abs(I-point1(1))<10).*(abs(J-point1(2))<10).*(abs(K-point1(3))<3);
mat2= zeros(size(A));
mat2(ind(:)) = mat1(:);

linfitpts = A.*mat2;
linfitptindex = find(linfitpts);
[x,y,z] = ind2sub(size(A),linfitptindex);

%Points are weighted exponentially by their SDF values
w = exp(-(A(linfitptindex))/25);

x = x';
y = y';
z = z';
w = w';

s2 = size(x);
n = s2(2);
xyz = [x',y',z'];

%Weighted mean to find center of plane

P01 = ((w./sum(w))*xyz) ;
```

```
%Subtract P0 from all xyz2 elements
xyz = xyz-repmat(P01,n,1);

%Weighted linear fit
[~,~,V]=svd(xyz.*repmat(w'/mean(w),1,3),0);

%First component is linear fit, other components comprise perpendicular plane
P12 = V(:,2)';
P13 = V(:,3)';

%Emit particles in plane in a radius r around point P01
r = 0:0.4:6;
theta = (0:(2*pi)/60:2*pi);
costheta = cos(theta);
sintheta = sin(theta);

X = [];
Y = [];
Z = [];
for i = 0:0.4:6
rctheta = i'*costheta;
rstheta = i'*sintheta;

X = [X (P01(1)+P12(1).*rctheta + P13(1).*rstheta)];
Y = [Y (P01(2)+P12(2).*rctheta + P13(2).*rstheta)];
Z = [Z (P01(3)+P12(3).*rctheta + P13(3).*rstheta)];
end

s = size(r,2)*size(theta,2);

X1 = reshape(X, 1, s);
Y1 = reshape(Y, 1, s);
Z1 = reshape(Z, 1, s);
```

```
floorX = floor(X1);
ceilX = ceil(X1);
floorY = floor(Y1);
ceilY = ceil(Y1);
floorZ = floor(Z1);
ceilZ = ceil(Z1);

floorB = sub2ind(size(B), floorX, floorY, floorZ);
ceilB = sub2ind(size(B), ceilX, ceilY, ceilZ);

floorB = B(floorB);
ceilB = B(ceilB);

X = X1(~~(floorB.*ceilB));
Y = Y1(~~(floorB.*ceilB));
Z = Z1(~~(floorB.*ceilB));

%If using the GPU, need to gather data for CPU
% X = gather(X);
% Y = gather(Y);
% Z = gather(Z);
```

A.5 Particle trace with GPU in MATLAB

In the following script, we have used a function, 'cline', which was written by Daniel Ennis, and taken from the MATLAB file exchange. The code is available for download at `http://www.mathworks.com/matlabcentral/fileexchange/3747-cline-m`, and is used to colour code the particle traces according to their velocities.

```
%Read in segmentation and the phase-contrast images
A = analyze75read('finalcfs');
```

```
P1 = analyze75read('1P');

P2 = analyze75read('2P');

P3 = analyze75read('3P');

s=size(P1);


%Give GPU the data

A = gpuArray(A);

P1 = gpuArray(P1);

P2 = gpuArray(P2);

P3 = gpuArray(P3);


input1 = double(reshape(P1, s(1), s(2), s(3)/15, 15));

input2 = double(reshape(P2, s(1), s(2), s(3)/15, 15));

input3 = double(reshape(P3, s(1), s(2), s(3)/15, 15));



%Set all points outside the segmentation to zero

%An intensity of 2048 corresponds to a speed of 0

repA = (repmat(A,[1 1 1 15]));


input1 = input1.*((repA) > 0);

input1 = input1 + ~input1.*2048;


input2 = input2.*((repA) > 0);

input2 = input2 + ~input2.*2048;


input3 = input3.*((repA) > 0);

input3 = input3 + ~input3.*2048;



%Calculate speed at each point in the image

speedimg = sqrt((input1-2048).^2 + (input2-2048).^2 + (input3-2048).^2).*1./2048;


%Open figure window
```

```
fullscreen = get(0,'ScreenSize');
hf = figure('Position',[0 -50 fullscreen(3) fullscreen(4)]);

%Create trasnparent mesh of segmentation
isosurface(A)
alpha(0.1);

%Define variables

[X, Y, Z] = planeemitter([80 85 20]);
numparticles = size(X,2); % Number of particles being emitted
T = ones(numparticles,1); %Time
it = 195; %iterations
t = 1.0; %Starting time
n = 14; %Number of time steps between frames
h = 1/n; %Step size

%Define GPU storage arrays

GPUppos = gpuArray([X(:) Y(:) Z(:) T(:)]);
GPUnewppos = GPUppos;
GPUpvel = gpuArray.zeros(numparticles,4);
GPUhist = gpuArray.zeros(numparticles,it+1,4);

GPUhist(:,1,:) = GPUppos;
GPUspeed = gpuArray.zeros(numparticles,it+1);

%Begin particle trace
for i = 1:it

GPUppos = GPUnewppos;
GPUhist(:,i+1,:) = GPUppos;

%Points used for velocity interpolation
```

```
intpos1 = floor(GPUppos);

intpos2 = intpos1 + 1;


%Offsets

alpha = GPUppos(:,1)-intpos1(:,1);

beta = GPUppos(:,2)-intpos1(:,2);

gamma = GPUppos(:,3)-intpos1(:,3);

omega = GPUppos(:,4) - intpos1(:,4);


%Use the code below if interpolating streamlines (time constant)


% xspeed = interp3(single(gather(input1(:,:,:,t))),single(gather(GPUppos(:,2))),

% single(gather(GPUppos(:,1))),single(gather(GPUppos(:,3))));

% yspeed = interp3(single(gather(input3(:,:,:,t))),single(gather(GPUppos(:,2))),

% single(gather(GPUppos(:,1))),single(gather(GPUppos(:,3))));

% zspeed = interp3(single(gather(input2(:,:,:,t))),single(gather(GPUppos(:,2))),

% single(gather(GPUppos(:,1))),single(gather(GPUppos(:,3))));

% xspeed = interp3_gpu(gpuArray(1:144), gpuArray(1:192), gpuArray(1:38),

% input1(:,:,:,t),GPUppos(:,2),GPUppos(:,1),GPUppos(:,3));

% yspeed = interp3_gpu(gpuArray(1:144), gpuArray(1:192), gpuArray(1:38),

% input3(:,:,:,t),GPUppos(:,2),GPUppos(:,1),GPUppos(:,3));

% zspeed = interp3_gpu(gpuArray(1:144), gpuArray(1:192), gpuArray(1:38),

% input2(:,:,:,t),GPUppos(:,2),GPUppos(:,1),GPUppos(:,3));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%Use code below for interpolation of pathlines


%X values

x1 = input1(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos1(:,3),intpos1(:,4)))

.*(1-alpha(:)).*(1-beta(:)).*(1-gamma(:)).*(1-omega(:));

x2 = input1(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos2(:,3),intpos1(:,4)))

.*(1-alpha(:)).*(1-beta(:)).*(gamma(:)).*(1-omega(:));

x3 = input1(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos1(:,3),intpos1(:,4)))
```

```
.*(1-alpha(:)).*(beta(:)).*(1-gamma(:)).*(1-omega(:));

x4 = input1(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos1(:,3),intpos1(:,4)))

.*(alpha(:)).*(1-beta(:)).*(1-gamma(:)).*(1-omega(:));

x5 = input1(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos2(:,3),intpos1(:,4)))

.*(1-alpha(:)).*(beta(:)).*(gamma(:)).*(1-omega(:));

x6 = input1(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos2(:,3),intpos1(:,4)))

.*(alpha(:)).*(1-beta(:)).*(gamma(:)).*(1-omega(:));

x7 = input1(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos1(:,3),intpos1(:,4)))

.*(alpha(:)).*(beta(:)).*(1-gamma(:)).*(1-omega(:));

x8 = input1(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos2(:,3),intpos1(:,4)))

.*(alpha(:)).*(beta(:)).*(gamma(:)).*(1-omega(:));

x9 = input1(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos1(:,3),intpos2(:,4)))

.*(1-alpha(:)).*(1-beta(:)).*(1-gamma(:)).*(omega(:));

x10 = input1(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos2(:,3),intpos2(:,4)))

.*(1-alpha(:)).*(1-beta(:)).*(gamma(:)).*(omega(:));

x11 = input1(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos1(:,3),intpos2(:,4))

).*(1-alpha(:)).*(beta(:)).*(1-gamma(:)).*(omega(:));

x12 = input1(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos1(:,3),intpos2(:,4)))

.*(alpha(:)).*(1-beta(:)).*(1-gamma(:)).*(omega(:));

x13 = input1(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos2(:,3),intpos2(:,4)))

.*(1-alpha(:)).*(beta(:)).*(gamma(:)).*(omega(:));

x14 = input1(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos2(:,3),intpos2(:,4)))

.*(alpha(:)).*(1-beta(:)).*(gamma(:)).*(omega(:));

x15 = input1(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos1(:,3),intpos2(:,4)))

.*(alpha(:)).*(beta(:)).*(1-gamma(:)).*(omega(:));

x16 = input1(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos2(:,3),intpos2(:,4)))

.*(alpha(:)).*(beta(:)).*(gamma(:)).*(omega(:));


xspeed =  x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 + x12 + x13 + x14 + x15 + x16;


%Y values


x1 = input3(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos1(:,3),intpos1(:,4)))

.*(1-alpha(:)).*(1-beta(:)).*(1-gamma(:)).*(1-omega(:));
```

```
x2 = input3(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos2(:,3),intpos1(:,4))
).*(1-alpha(:)).*(1-beta(:)).*(gamma(:)).*(1-omega(:));
x3 = input3(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos1(:,3),intpos1(:,4)))
.*(1-alpha(:)).*(beta(:)).*(1-gamma(:)).*(1-omega(:));
x4 = input3(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos1(:,3),intpos1(:,4)))
.*(alpha(:)).*(1-beta(:)).*(1-gamma(:)).*(1-omega(:));
x5 = input3(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos2(:,3),intpos1(:,4)))
.*(1-alpha(:)).*(beta(:)).*(gamma(:)).*(1-omega(:));
x6 = input3(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos2(:,3),intpos1(:,4)))
.*(alpha(:)).*(1-beta(:)).*(gamma(:)).*(1-omega(:));
x7 = input3(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos1(:,3),intpos1(:,4)))
.*(alpha(:)).*(beta(:)).*(1-gamma(:)).*(1-omega(:));
x8 = input3(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos2(:,3),intpos1(:,4)))
.*(alpha(:)).*(beta(:)).*(gamma(:)).*(1-omega(:));
x9 = input3(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos1(:,3),intpos2(:,4)))
.*(1-alpha(:)).*(1-beta(:)).*(1-gamma(:)).*(omega(:));
x10 = input3(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos2(:,3),intpos2(:,4)))
.*(1-alpha(:)).*(1-beta(:)).*(gamma(:)).*(omega(:));
x11 = input3(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos1(:,3),intpos2(:,4)))
.*(1-alpha(:)).*(beta(:)).*(1-gamma(:)).*(omega(:));
x12 = input3(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos1(:,3),intpos2(:,4)))
.*(alpha(:)).*(1-beta(:)).*(1-gamma(:)).*(omega(:));
x13 = input3(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos2(:,3),intpos2(:,4)))
.*(1-alpha(:)).*(beta(:)).*(gamma(:)).*(omega(:));
x14 = input3(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos2(:,3),intpos2(:,4)))
.*(alpha(:)).*(1-beta(:)).*(gamma(:)).*(omega(:));
x15 = input3(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos1(:,3),intpos2(:,4)))
.*(alpha(:)).*(beta(:)).*(1-gamma(:)).*(omega(:));
x16 = input3(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos2(:,3),intpos2(:,4)))
.*(alpha(:)).*(beta(:)).*(gamma(:)).*(omega(:));

yspeed =  x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 + x12 + x13 + x14 + x15 + x16;

%Z values
```

```
x1 = input2(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos1(:,3),intpos1(:,4)))
.*(1-alpha(:)).*(1-beta).*(1-gamma).*(1-omega);
x2 = input2(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos2(:,3),intpos1(:,4)))
.*(1-alpha(:)).*(1-beta).*(gamma(:)).*(1-omega);
x3 = input2(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos1(:,3),intpos1(:,4)))
.*(1-alpha(:)).*(beta(:)).*(1-gamma).*(1-omega);
x4 = input2(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos1(:,3),intpos1(:,4)))
.*(alpha(:)).*(1-beta).*(1-gamma).*(1-omega);
x5 = input2(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos2(:,3),intpos1(:,4)))
.*(1-alpha(:)).*(beta(:)).*(gamma(:)).*(1-omega);
x6 = input2(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos2(:,3),intpos1(:,4)))
.*(alpha(:)).*(1-beta).*(gamma(:)).*(1-omega);
x7 = input2(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos1(:,3),intpos1(:,4)))
.*(alpha(:)).*(beta(:)).*(1-gamma).*(1-omega);
x8 = input2(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos2(:,3),intpos1(:,4)))
.*(alpha(:)).*(beta(:)).*(gamma(:)).*(1-omega);
x9 = input2(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos1(:,3),intpos2(:,4)))
.*(1-alpha(:)).*(1-beta).*(1-gamma).*(omega(:));
x10 = input2(sub2ind(size(input1),intpos1(:,1),intpos1(:,2),intpos2(:,3),intpos2(:,4)))
.*(1-alpha(:)).*(1-beta).*(gamma(:)).*(omega(:));
x11 = input2(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos1(:,3),intpos2(:,4)))
.*(1-alpha(:)).*(beta(:)).*(1-gamma).*(omega(:));
x12 = input2(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos1(:,3),intpos2(:,4)))
.*(alpha(:)).*(1-beta).*(1-gamma).*(omega(:));
x13 = input2(sub2ind(size(input1),intpos1(:,1),intpos2(:,2),intpos2(:,3),intpos2(:,4)))
.*(1-alpha(:)).*(beta(:)).*(gamma(:)).*(omega(:));
x14 = input2(sub2ind(size(input1),intpos2(:,1),intpos1(:,2),intpos2(:,3),intpos2(:,4)))
.*(alpha(:)).*(1-beta).*(gamma(:)).*(omega(:));
x15 = input2(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos1(:,3),intpos2(:,4)))
.*(alpha(:)).*(beta(:)).*(1-gamma).*(omega(:));
x16 = input2(sub2ind(size(input1),intpos2(:,1),intpos2(:,2),intpos2(:,3),intpos2(:,4)))
.*(alpha(:)).*(beta(:)).*(gamma(:)).*(omega(:));
```

```
zspeed =  x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 + x12 + x13 + x14 + x15 + x16;


%Speed normalized to range of 0-1 (for colouring purposes)

 GPUspeed(:,i+1) = (1/2048).*sqrt((xspeed-2048).^2 + (yspeed-2048).^2 + (zspeed-2048).^2);


%Advance time by 1ms

t = t + h;

GPUpvel = [(xspeed-2048) (yspeed-2048) (zspeed-2048)].*repmat([1 1 0.5],numparticles,1)./2048;

GPUnewppos = GPUppos + [GPUpvel repmat(h,numparticles,1)];


%Use this update scheme if keeping time constant

%GPUnewppos = GPUppos + [GPUpvel zeros(numparticles,1)];


%Fix precision error

if t > 15

    t = 1;

    GPUnewppos = GPUnewppos - repmat([0 0 0 14+h],numparticles,1);

    GPUnewppos(:,4) = ceil(GPUnewppos(:,4));

end


end


%Use cline to colour line

hist = gather(GPUhist);

sp = gather(GPUspeed);


for s = 1:numparticles

  h = cline(hist(s,:,2), hist(s,:,1), hist(s,:,3), sp(s,:));

  set(h, 'LineWidth', 1)

end


hcb = colorbar;

set(hcb, 'YTick', [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]*sqrt(3))

set(hcb, 'YTickLabel', {'0 cm/s','26 cm/s','52 cm/s','78 cm/s','104 cm/s','130 cm/s',
'156 cm/s','182 cm/s','208 cm/s','234 cm/s','260 cm/s'})
 axis equal
 axis tight

%Create movie
 set(gca, 'NextPlot', 'replaceChildren');
 F(it+1) = struct('cdata', [], 'colormap', []);

  for frame = 1:it+1
      for s = 1:numparticles
          h = cline(hist(s,1:frame,2), hist(s,1:frame,1), hist(s,1:frame,3), sp(s,1:frame));
          set(h, 'LineWidth', 3.5)
          set(hcb, 'YTick', [0, 0.2, 0.4, 0.6, 0.8, 1])
          set(hcb, 'YTickLabel', {'0 cm/s','52 cm/s','104 cm/s','156 cm/s','208 cm/s','260 cm/s'})
          axis equal
          axis tight
          hold on;
      end
      F(frame) = getframe;
  end
hcb = colorbar;
set(hcb, 'YTick', [0, 0.2, 0.4, 0.6, 0.8, 1])
set(hcb, 'YTickLabel', {'0 cm/s','52 cm/s','104 cm/s','156 cm/s','208 cm/s','260 cm/s'})
 movie2avi(F, 'Flow.avi');
```