TRABAJO DE FIN DE MÁSTER

Máster en Interdisciplinary and Innovative Engineering

IMAGE PROCESSING PLATFORM FOR THE ANALYSIS OF BRAIN VASCULAR PATTERNS

Memoria

Autor:	Nicolás Arrieta
Director:	Raúl Benítez
Convocatoria:	Octubre 2021

ABSTRACT

This project consists in the development of a web application for the support of medical professionals in the analysis of cerebrovascular image data. The objective is to build an open and modular prototype that can serve as an example or template for the development of other projects. The purpose is to have an open alternative to the commercial options currently available for data analysis tools in the health industry market. The application is developed using Python. The application allows the user to load medical images contained in DICOM files, those images are processed for noise removal and segmentation in order to build the result graphs. The results are three graphs: an image graph called "isochronal map" reflecting the temporal evolution of the blood flow, an image graph showing the skeleton of the vascular system structure, a box-plot graph representing the numerical branch data extracted from the skeleton. The Dash framework is used to construct the user interface and to implement the user interaction functionalities. The subject can load two different samples at the same time and execute the analysis to compare the results for both samples in the same screen. Finally the application is containerized using Docker to package it and make it multi-platform. The app is tested and the results are satisfactory as the resulting application works properly and so do the image processing algorithms for the input data provided by the Hospital Sant Joan de Déu. Despite its obvious limitations, the work done serves as a starting point for future developments.

ACKNOWLEDGEMENTS

This project was made in collaboration with the Hospital Sant Joan de Déu of Barcelona. It is based on a previous work done by Carlos Rebollo in his Masters' thesis *Dynamic Characterization of Blood Flow Through The Cerebral Artery Circle By Magnetic Resonance Imaging*[1].

I would like to show my gratitude to my thesis supervisor, Dr. Raúl Benítez Iglesias, for guiding me in the development of the project. Also to Dr. Josep Munuera from the Hospital Sant Joan de Déu, who provided his medical counseling and the medical data used for analysis.

INDEX

SYMBOLS AND ABBREVIATIONS

AVM	Arteriovenous Malformation
CVD	Cerebrovascular Disease
MRA	Magnetic Resonance Angiography
NORD	National Organization for Rare Disorders
TRANCE	Time Resolved Angiography Non Contrast Enhanced

1. PREFACE

1.1 Overview

In the last few years, there has been an important evolution in the health industry due to the integration of emerging technologies. Information and communication technologies have played a key role in assisting health professionals in their activities, such as for the management and analysis of patients' data for diagnosis. Different initiatives have been launched to address these matters; one of special interest for this project is the Digital Imaging and Communications in Medicine (DICOM) standard for the communication and management of medical imaging information and related data. DICOM enables the integration of medical imaging devices (such as scanners, servers, printers, *etc*) and communication systems from multiple manufacturers.

In this scenario, different types of applications have been developed for the support of health professionals. The visualization, annotation and processing of medical images is an important part of the process of medical diagnosis. The objective of this project is to provide neurologists with a tool to help them in identifying vascular anomalies and malformations in the brain, by designing a web-based application for the automatic analysis of Magnetic Resonance Angiography (MRA) images and the presentation of the results.

1.2 Motivation

This project was conceived to support the Sant Joan de Déu hospital of Barcelona in exploring the idea of building a custom application to manage and process the medical imaging data that they store. This could help medical professionals in the diagnosis, treatment and research. Specifically, in this project, the idea is to process medical images that correspond to cerebral MRAs, in order to analyze the structure of the vascular vessels of the brain and the dynamics of the blood flow through them.

Such an application could be very useful for detecting structural malformations and functional anomalies in the brain; and to supervise the efficacy of the treatments applied. Currently, there are computer applications to achieve these goals available, but they are generally owned by medical imaging equipment manufacturers and often they are not cost-effective. One of the incentives of the project is to explore the possibility of an open and custom alternative to those existing applications.

There is also the interest that the application design of this project could serve as a base or template for the development of other related medical apps with different purposes and functionality. There are already public projects exploring this idea, such as the Open Health Imaging Foundation (OHIF), but there are not many open platforms available.

Another reason justifying the need for such a tool is to bring the possibility of customization and the use of custom algorithms or functions.

1.3 Scope And Objectives

The objective of this project is to develop an open and custom application for the analysis of MRA medical images to serve as a diagnosis support tool for neurologists. The application is intended to be simple, with the ease of use as a priority. It is aimed to be multiplatform, easy to install and portable, so the users have a simple setup and direct access.

The goal is to deliver a functional prototype app that provides graphical information of the results of the automatic analysis of the image data, those graphs are:

- A coloured isochronal map showing the evolution of the blood flow through the brain vessels.
- A binarized skeleton of all the vascular structure, identifying the nodes and branches formed by the vessels.
- A graphical visualization representing the information of different attributes that result from the skeleton analysis.
- A table containing the retrieved structured data of the skeleton.

The application will allow the user to load, analyze and compare the data for two different subjects at once, so a control subject and a case subject can be contrasted. The information available, provided by the hospital, are MRAs of different patients. The medical images analyzed for each subject consist of a set of frames, sequentially acquired, that correspond to a time slice of around 1.6 seconds. Those frames should be pre-processed in order to remove noise and to be able to properly identify the objects of interest in the image, the vessels. This is one of the main challenges of the project, as implementing an automatic filtering that works for most of the possible inputs can be an arduous task.

The image processing steps are fully detailed in the methodology section, here is a list of them:

1. Superposition/sum of the frames to retrieve a picture of all the vessels
2. Denoising of the frame sum and of each single frame
3. Segmentation
4. Plot of the isochronal map, built from the binarized frames
5. Skeletonization of the binarized total frame sum
6. Analysis of the skeleton

The technology stack chosen to build the application is based on a platform called Dash, which is an open source library that allows the development of modern cross-platform data apps in a simple and direct manner. Dash was chosen because of its simplicity, web based architecture and rich predefined components catalogue.

The application is almost entirely written in Python except for the CSS used for modifying the web layout style. Different Python libraries are used for the implementation of the image processing functions.

For the deployment of the application, a Docker container is configured. Docker is an OS-level virtualization platform used to deliver software in packages called containers. By using a Docker container, the app runs in an isolated environment having just the required dependencies installed. By making this setup, the user does not need to install anything but Docker, and can run the application in any host operating system.

The requirements and costs of the project are minimum, the software used for the development is open source and free to use. To check the performance of the final application, it is tested with some subjects' data, of patients presenting different conditions.

2. BACKGROUND

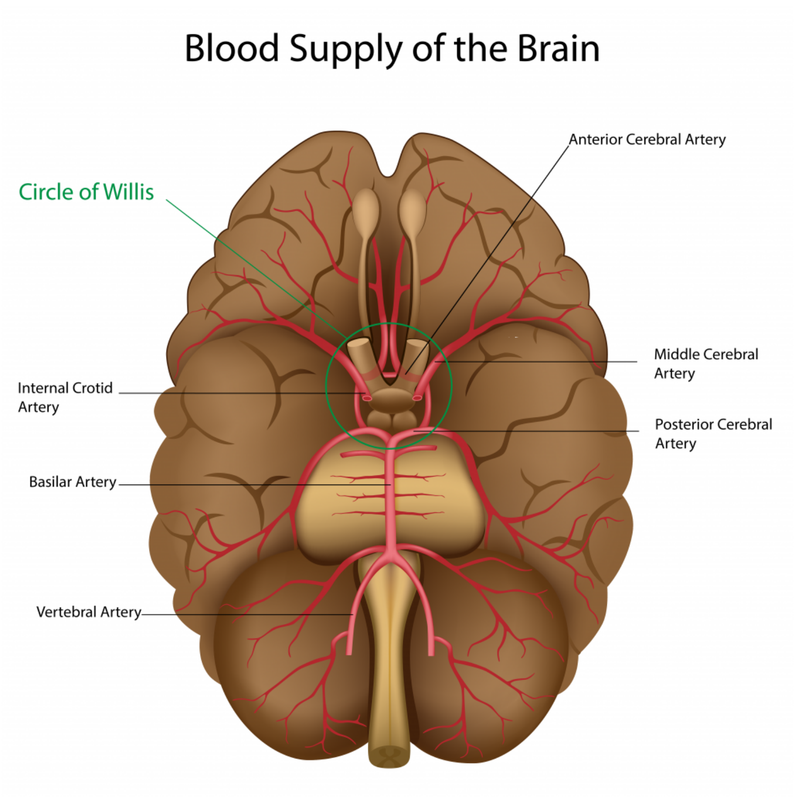
2.1 Cerebral Circulation, Pathologies And Diagnosis

2.1.1 The Brain Vascular System

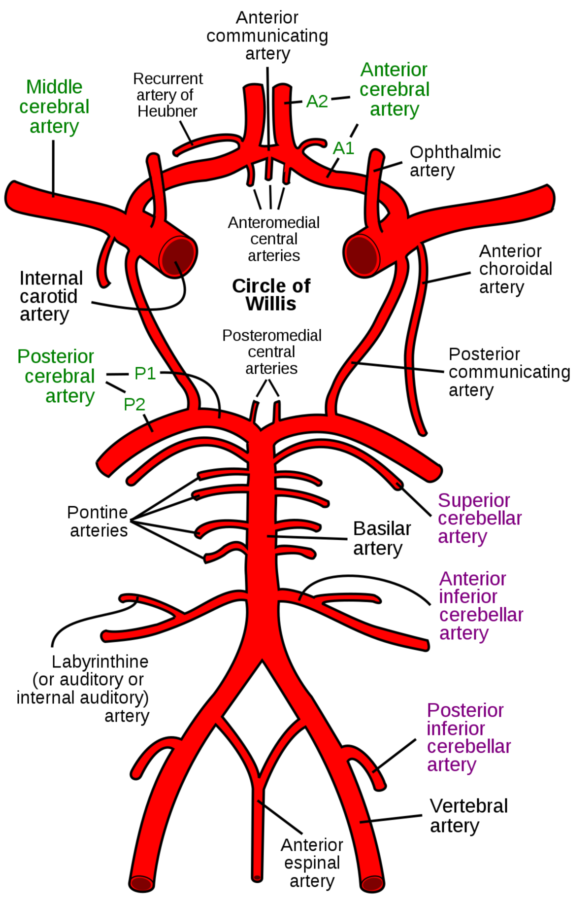
The vascular system of the brain is composed of a network of blood vessels that carry out the function of supplying the brain tissues with blood. As in other parts of the body, arteries deliver oxygen, glucose and nutrients; and veins remove different metabolic waste such as carbon dioxide or lactic acid. The blood supply of the brain is a critical process: "The brain only accounts for 2% of adult body mass (approximately 1400 g) but receives approximately 15% of the resting cardiac output" (S Shah and S Jeyaretna[2]). Any shortage of blood supply to the brain can lead to stroke or other life-threatening conditions. There are a variety of important factors that play a role in the cerebral circulation such as the mean arterial pressure, intracranial pressure, arterial carbon dioxide tension, *etc*. The system autorregulates in order to keep stable blood flow rates.

The heart pumps blood up to the brain through two sets of arteries: the carotid arteries, which supply blood to the front two-thirds of the brain; and the vertebral arteries, that supply the back third of the brain. The jugular and other veins carry blood out of the brain.

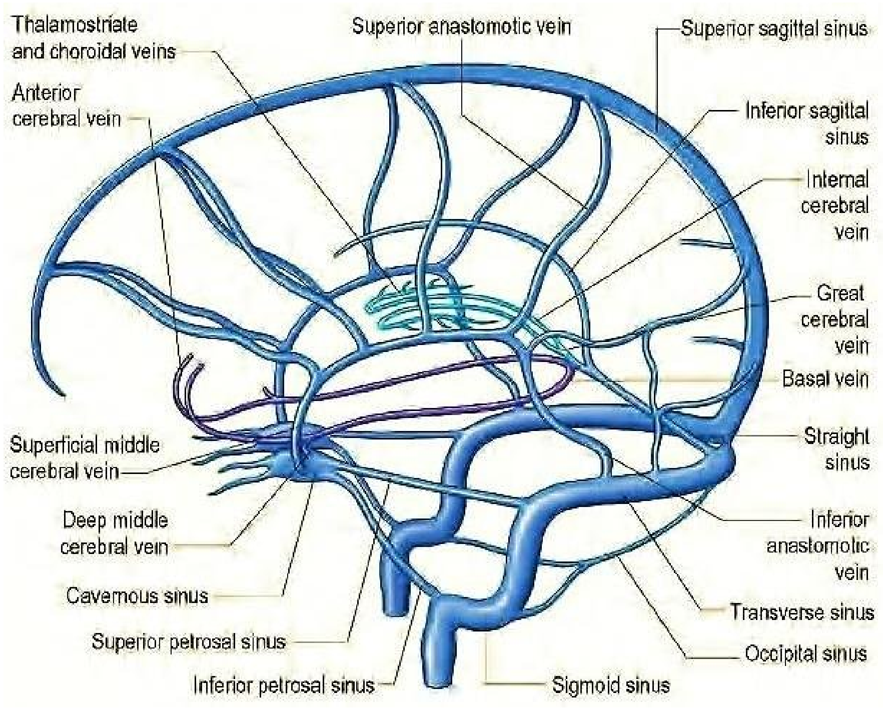
There are different structures involved in the arterial supply of the brain: the aortic arch, the anterior and posterior circulation and the circle of Willis, which is a vascular loop connecting both brain hemispheres and the anterior and posterior circulation.



Arterial supply (source: *Ansys.com*)


Anterior communicating artery
Middle cerebral artery
Recurrent artery of Heubner
A2
Anterior cerebral artery
A1
Ophthalmic artery
Anteromedial central arteries
Internal carotid artery
Circle of Willis
Anterior choroidal artery
Posteromedial central arteries
Posterior cerebral artery
P1
P2
Posterior communicating artery
Pontine arteries
Basilar artery
Superior cerebellar artery
Anterior inferior cerebellar artery
Labyrinthine (or auditory or internal auditory) artery
Posterior inferior cerebellar artery
Vertebral artery
Anterior espinal artery

The circle of Willis (source: *Wikipedia.org*)



Cerebral veins (source: Gray's *Anatomy,* 2008)

It is also important to mention the blood-brain barrier, which is a semi-permeable barrier that blocks the diffusion of many compounds from the blood to the brain. The venous system can be divided in three main parts: the venous sinuses, the superficial cerebral veins and the deep cerebral veins.

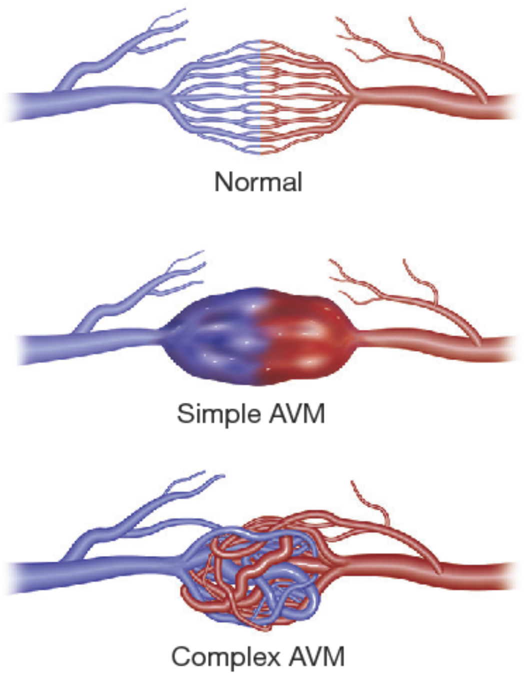
2.1.2 Cerebrovascular Disease (CVD)

As noted by the American Association of Neurological Surgeons[3] (AANS): "The term cerebrovascular disease includes all disorders in which an area of the brain is temporarily or permanently affected by ischemia (lack of blood flow) or bleeding and one or more of the cerebral blood vessels are involved in the pathological process. Cerebrovascular disease includes stroke, carotid stenosis, vertebral stenosis and intracranial stenosis, aneurysms, and vascular malformations."

The blood flow restrictions may be a cause of:

- Stenosis (vessel narrowing)
- Thrombosis (clot formation)
- Embolism (blockage)
- Hemorrhage (blood vessel rupture)

Vascular malformations are abnormal connections between arteries and veins, and are developed during pregnancy. These include the arteriovenous malformations (AVMs) which is a tangle of abnormal and poorly formed blood vessels, as shown in the next figure:



Arteriovenous Malformations (source: Chamarthy *et. al.*[4])

There are other pathologies affecting the brain vessels, for example the Moyamoya disease, which is a progressive condition characterized by the narrowing or closing of the carotid artery to the brain.

The National Organization for Rare Disorders[5] states that vascular malformations of the brain may cause headaches, seizures, strokes or hemorrhages. Other symptoms include weakness or numbness on one side of the body, speech trouble, balance and coordination difficulties, vomiting, loss of vision, *etc*.

Cerebrovascular diseases are related to both congenital (genetic) and acquired causes. Some other conditions such as the Moyamoya disease appear spontaneously without a known cause. Regarding epidemiology, CVDs are one of the most common causes of death and disability in the world population and AVMs affect about 1 percent of the general population[3].

These diseases are more common in advanced age, but can occur at any stage of life. Some of the main risk factors for CVDs include smoking, hypertension, diabetes, cholesterol, physical inactivity, obesity, stress, *etc*. Controlling or treating these factors can help in the prevention of these diseases.

The diagnosis of cerebrovascular problems is mainly based on imaging tests that allow neurologists to view the vessels around and inside the brain. Some of these medical imaging techniques, ones more invasive than others, include: cerebral angiography, carotid ultrasound, computed tomography, doppler ultrasound, magnetic resonance imaging (MRI) and magnetic resonance angiogram (MRA), *etc*. Some of these techniques are discussed later in the section 2.2.1 Medical Imaging Techniques. The medical history and physical and neurological exams also help in the detection of CVMs.

For the treatment of CVMs, there are several advanced surgery methods such as craniotomy, neuroendovascular therapy (i.e. embolization), microsurgery, stereotaxic radiosurgery, irradiation, *etc*. The treatment generally includes lifestyle changes and medication prescription, with antihypertensives to reduce the blood pressure, anticoagulants to prevent blood clotting or medications to reduce cholesterol.

It is crucial to detect and treat the development of CVMs as soon as possible in order to prevent damage and avoid risks leading to more serious conditions. As the brain is directly affected by these diseases, they are life-threatening, and speed is crucial in emergency cases. A late intervention can result in long-term disability or death of the patient; other complications include loss of cognitive functions and memory, partial paralysis and speech difficulties. The prognosis of a cerebrovascular disease depends on its severity and stage at the point of diagnosis.

2.2 Medical Imaging

2.2.1 Medical Imaging Techniques

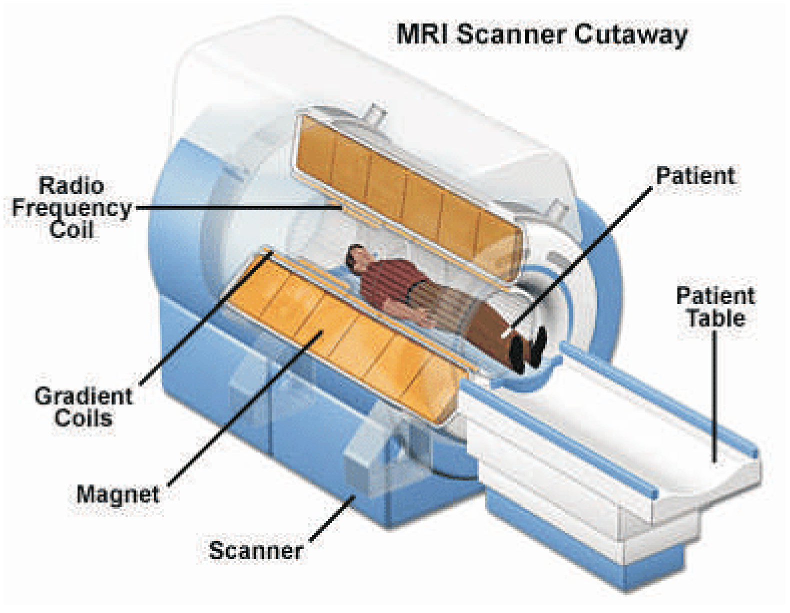
The morphological characteristics of the intracranial vessels can be studied using imaging methods. From a global classification, for the study of cerebrovascular diseases some of the techniques most commonly used include: Cerebral angiography, Magnetic Resonance Imaging (MRI), Computed Tomography (CT) and ultrasounds. (Imperial College London[6])

In most of the cases, a contrast (a special dye) is added to the bloodstream in order to facilitate the scanning. These imaging tests help detect vascular malformations and other pathologies.

Medical images are important for medical diagnosis and treatment, but also for investigation and improvement of techniques such as drug delivery.

2.2.2 Magnetic Resonance Angiography (MRA)

The MRA is a type of MRI technique that looks specifically at the body's blood vessels. It uses strong magnetic fields and radio waves to generate the images. It is a non-invasive procedure in which the patient lies inside a tunnel-like tube scanner. Some of the conditions studied using MRA are aneurysm, heart disease, narrowing or blocking of vessels, *etc*. The image acquisitions can be both 2D or 3D. (Johns Hopkins Medicine[7])



Component of a MRI scanner (Moore and Zouridakis, 2004)

The clinical applications for MRA are expanding due to the improvements in hardware and imaging techniques. New intravascular contrast agents and the use of advanced magnets (such as the 3.0 T magnets) have allowed the reduction in exogenous contrast dose, and therefore lower risks for the patient. One of the most used contrast agents is the gadolinium-based, and its use is a concern for high-risk patient groups who present renal insufficiency and vascular or metabolic disorders (Hartung et al.[8]). Nevertheless, there are alternative methods called non contrast enhanced MRA (NCE-MRA), which do not make use of contrast agents, however they require longer scan times than contrast enhanced (CE) methods. These techniques are widely used for intracranial imaging and there are different types of them:

- Time of Flight (TOF)
- Steady State Free Precession (SSFP)
- Phase Contrast (PC)

There are also specific proprietary methods used by MRA scanner manufacturers. One of these methods is used for the acquisition of the images that are analyzed in this project. The method is called Time Resolved Angiography Non Contrast Enhanced (TRANCE) and it is provided by Philips. It is an NCE-MRA method with high temporal resolution (down to 160 milliseconds) (Philips[9]).

2.2.3 The DICOM Standard

Medical data needs to be stored and transmitted in a standard manner in order to allow hospitals to exchange patient data and to allow the integration of medical devices from different manufacturers. With that aim the Digital Imaging and Communications in Medicine[10] (DICOM) standard was first published in 1993, and has become the global standard for medical imaging.

DICOM includes protocols for image exchange, compression, visualization, *etc*. It defines the structure of the DICOM files, which aggregate information into data sets, composed of different items such as name, ID, pixel data, *etc*. A single DICOM object can have only one attribute containing pixel data, but this attribute can contain multi-dimensional multi-frame images. Pixel data can be compressed using different standards such as JPEG, RLE, zip, *etc*. DICOM uses specific data encoding schemes that are defined in the standard.

DICOM also provides a function, the grayscale standard display function (GSDF), to allow identical grayscale image display on different monitors and printers that have been calibrated to the GSDF curve.

Finally, it is important to remark that DICOM provides services that involve transmission of data over the network.

Here is a useful explorer for the DICOM standard, provided by Innolitics:
https://dicom.innolitics.com/ciods

There are a wide variety of DICOM viewers available such as OsiriX or Postdicom. There are also open DICOM libraries with anonymized data, see dicomlibrary.com

And here is an open online DICOM viewer offered by the Open Health Imaging Foundation (OHIF):
https://v3-demo.ohif.org/

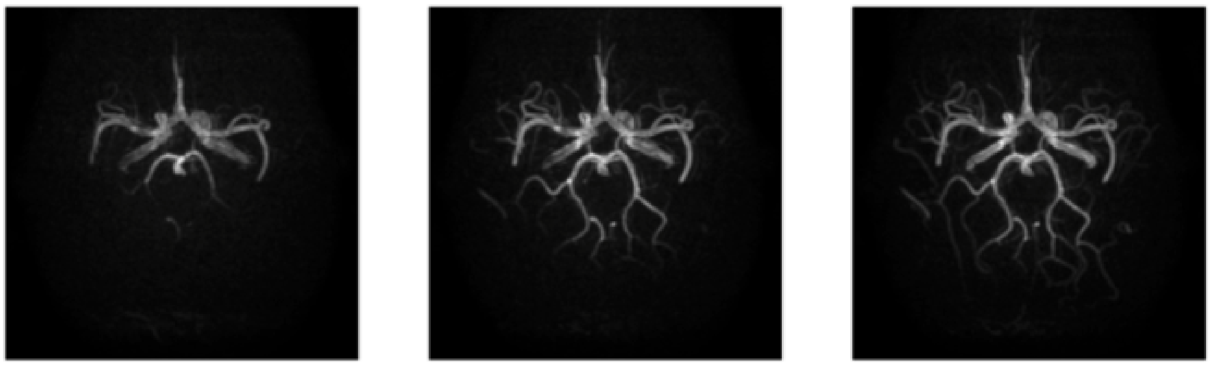
2.2.4 Medical Data Regulation

The protection of personal health data is a major growing concern, as nowadays healthcare is becoming more and more data-driven and sensitive medical data is a valuable target for cybercriminals. Security must be a priority, the legislation for it is specific to each country but there are general principles or guidelines proposed by important institutions such as the World Health Organization[12]. The risks associated with the vulnerability of medical data systems include personal safety, privacy, computer system attacks, *etc*.

2.3 Materials And Methods

2.3.1 Available Data

The images analyzed in the project consist of a set of frames for each case, containing a sequence of images of the brain vessels. Frames are separated by a temporal gap of 200 ms. The images are acquired using a Philips scanner with the TRANCE MRA technique.



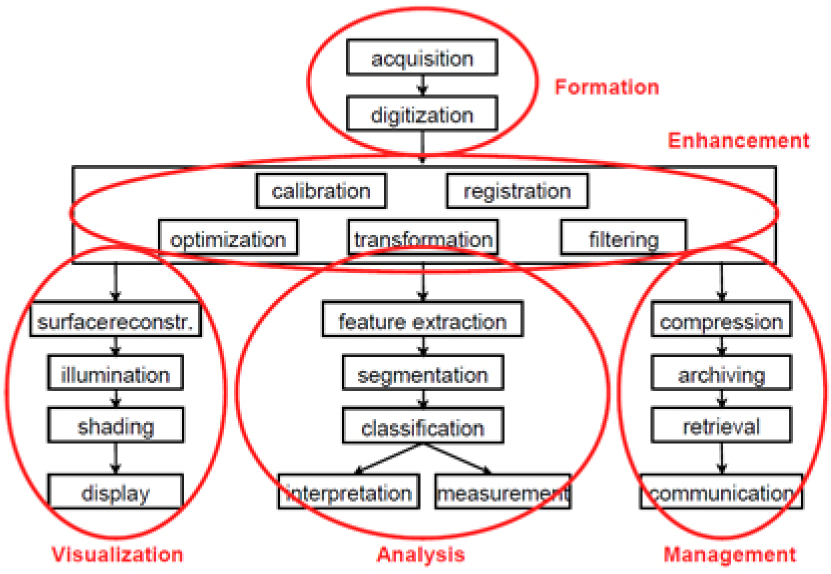
Samples of MRA images

A DICOM file is generated for each frame, the DICOMs are read in the application and only the pixel data attribute that contains the frame is used. The data provided by the Hospital Sant Joan de Déu includes both images from healthy subjects and cases presenting different conditions such as arteriovenous malformations.

2.3.2 Biomedical Image Processing

Digital signal processing techniques are applied to medical images in order to enhance them, remove noise and unwanted artifacts, and analyze them. The general process that is followed has these fundamental steps:

1. Filtering for noise and artifact removal.
2. Segmentation for identifying different anatomical regions.
3. Measurement and statistics to quantify different parts of the image data.



Scheme of image processing (The International Society for Optics and Photonics[14])

There are different types of image filters and segmentation algorithms. These steps require a-priori knowledge on the nature and content of the images, which must be integrated into the algorithms on a high level of abstraction. Thus, the process of image analysis is very specific, and developed algorithms can rarely be transferred directly into other domains of applications (Synopsys[13]).

Image processing software helps to automatically identify and analyze what might not be apparent to the human eye. Computerized algorithms can provide temporal and spatial analysis to detect patterns and characteristics indicative of tumors and other ailments (The International Society for Optics and Photonics[14]).

Biomedical image processing is a field in constant evolution, it is applied in many other areas apart from healthcare applications, such as research and education.

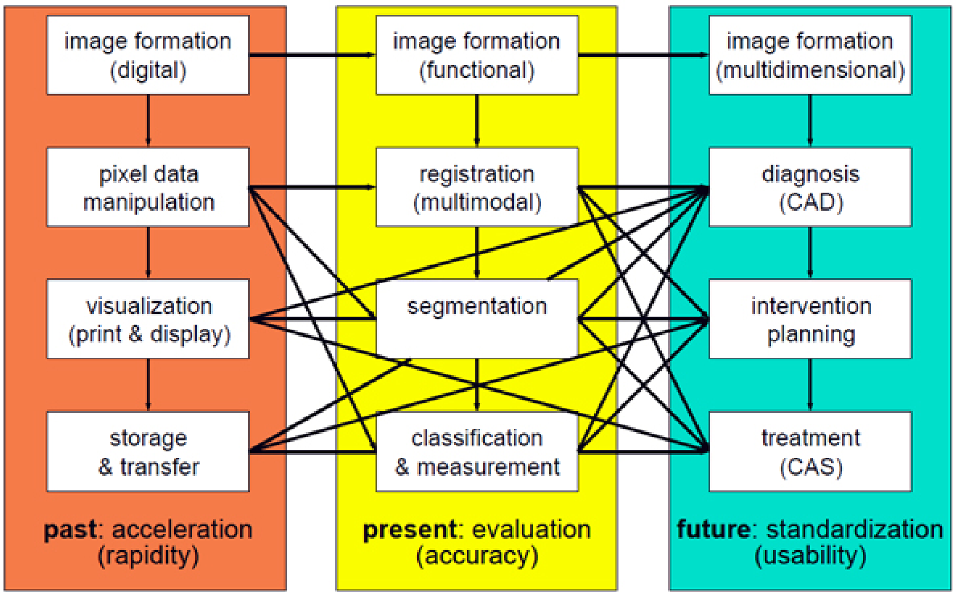


image formation (digital)
pixel data manipulation
visualization (print & display)
storage & transfer
past: acceleration (rapidity)
image formation (functional)
registration (multimodal)
segmentation
classification & measurement
present: evaluation (accuracy)
image formation (multidimensional)
diagnosis (CAD)
intervention planning
treatment (CAS)
future: standardization (usability)

Paradigms of medical image processing (The International Society for Optics and Photonics[14])

In this project, the images are processed, as detailed in the methodology section, in order to extract biomarkers of the morphological structure of the blood vessels and of the dynamics of the blood flow.

2.4 App Development

2.4.1 Python

Python is the computer language of choice to develop the application of this project. The main reason for choosing Python is that it is one of the most widely used programming languages in the field of data analysis and it provides a huge library of open source software ready to be used.

There are many different libraries available for biomedical image processing, some of the well known are *OpenCV*, *Insight Toolkit* (ITK*)* and *ImageJ*.

The libraries and algorithms used in this project for image processing are some of the most basic ones and can be found in many well known python packages: *numpy*, *scikit-image*, *pandas*, *pillow*...

There are also open Python frameworks that facilitate the process of web application development, one of them, called Dash, is used for this project.

2.4.2 Dash by Plotly

Dash is an open source framework, provided by Plotly and released under the MIT license, that allows to build full-stack web applications with interactive data visualization, while abstracting away the technologies and protocols that are required for that. Dash apps are rendered in a web browser and that makes it inherently cross-platform.

Dash provides built-in components as python objects that can be used to define the layout of the webpage. The user interaction is implemented with callback functions that can execute code and modify the app layout.

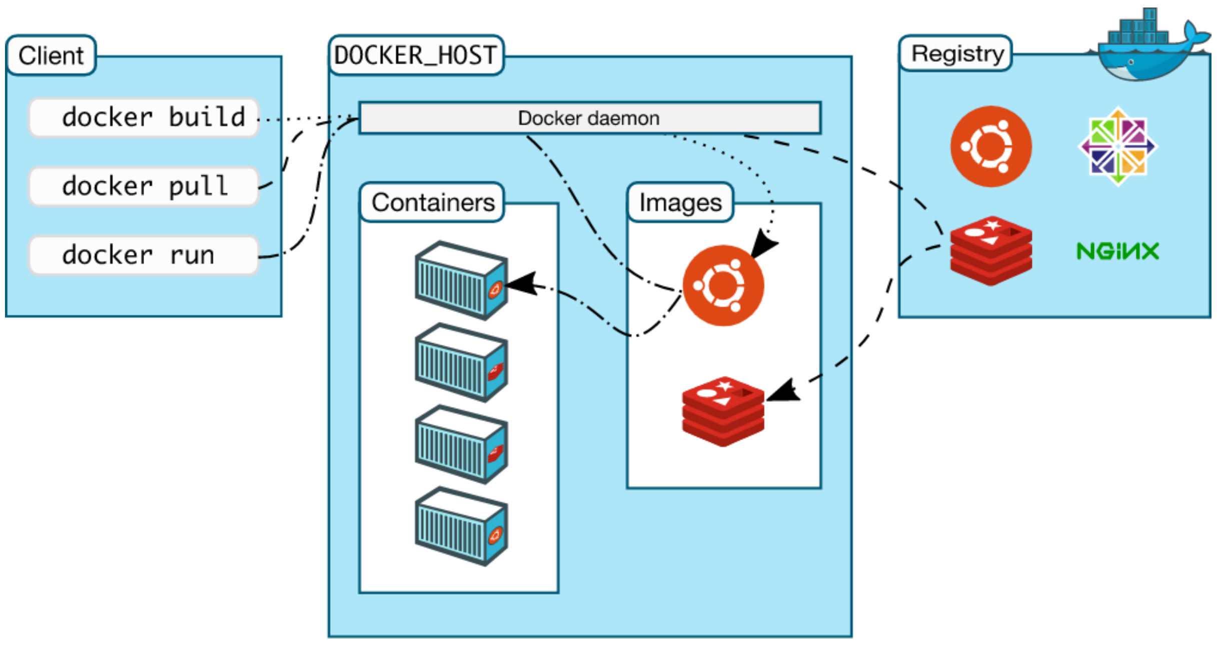
Dash is a stateless framework that makes it scalable and robust as it is trivial to add more compute power and to scale the application to serve more users or to run more computations in parallel. Stateless frameworks are more robust because one process can fail and other processes can continue serving requests. The app can be run in multiple containers or servers and balance the load between them.

2.4.3 Docker

Docker[15] is a platform as a service (PAAS) product that provides OS-level virtualization to deliver software in packages called containers. It enables separation of the application from the infrastructure making software delivery faster.

A container is a loosely isolated environment in which the application runs. It is isolated from the rest of the OS and can contain just the needed dependencies required to run the application. Therefore containers can be easily shared.

The architecture of docker has the next schema:



Docker architecture (Docker)

The *Docker Client* is the user interface of Docker, it is a command line interface (CLI) that allows the user to run different commands in order to manage the containers. The *Docker Host* is the core that manages Docker objects such as images and containers. A Docker image is a software package that includes everything needed to run an application, a Docker container is an execution instance of a Docker image. Finally, the Docker *Registry* is a repository that stores Docker images that can be used. Docker can run on multiple host operating systems including Linux, Mac and MS Windows.

The application developed in this project is “dockerized” (delivered as a Docker image) in order to be shared and installed easily. It can be easily configured to be executed on an external web server.

2.4.4 Integration In The Hospital Network

Hospitals have a complex IT infrastructure with dedicated networks to store, process and manage the patients’ data and to connect the medical devices used. The integration of everything is not a simple task, the DICOM standard establishes the protocols for it.

The IT systems used in a hospital include (O'Connor[16]):

- The Picture Archive and Communication System (PACS) that works as a medical image library to store, manage and retrieve medical images.
- The Radiology Information System (RIS) which is a software used for patient scheduling, resource management, reporting, etc.
- The Clinical Information System (CIS) that integrates information into a patient record that clinicians can consult.
- The Hospital Information System (HIS) which focuses on the administrational needs of hospitals: administrative, financial, legal and medical issues.

These definitions are mixed and blurred, but in general these are the global terms used to refer to the IT systems implemented in medical installations.

For the developed application to be functional, it should be properly integrated in the hospital’s computer network, and with its medical devices and servers.

2.5 Available Technologies

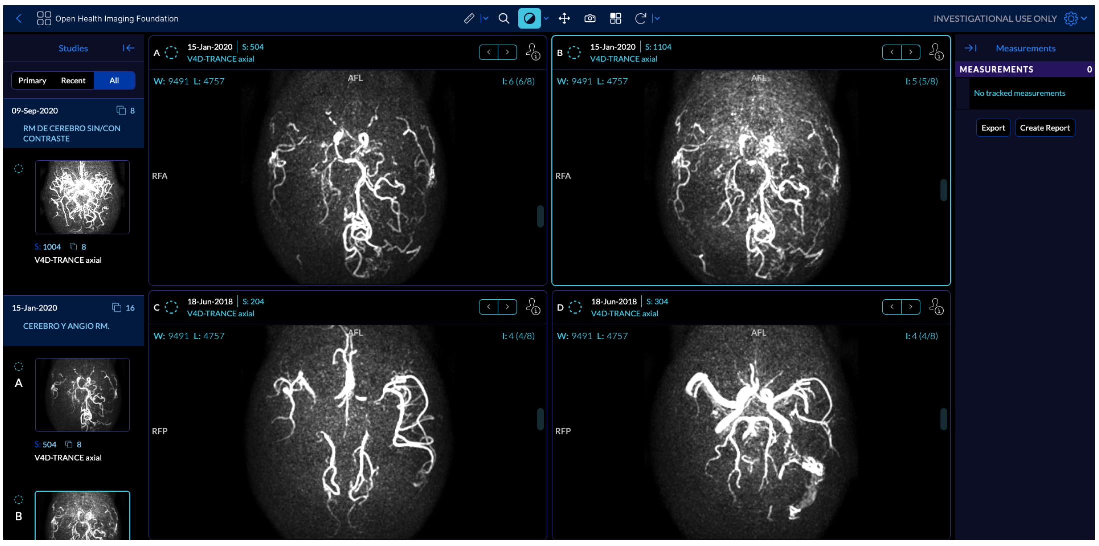
In the last few years, there has been an important integration of IT technologies in the medical sector; from new devices to software and applications serving different purposes in all the medical areas. The computer networks play a key role in the management of medical information in hospitals and clinics. The medical imaging systems include applications to process and visualize the acquired data. There are both proprietary and open software options available.

In terms of image processing, there are a lot of algorithms and software libraries accessible, for many different purposes and use cases. Some of the most important open projects in these area include:

- The *Insight Toolkit*[17] (ITK) which is a recognized library for the processing of scientific images.
- *ImageJ* is also an important library for image analysis provided by the National Institute of Health.

There are extensible and open platforms that can be used as a base for the development of a medical application.

- The web-based medical imaging platform offered by the Open Health Imaging Foundation (OHIF), which is an extensible software that can be customized to address many different needs or applications.



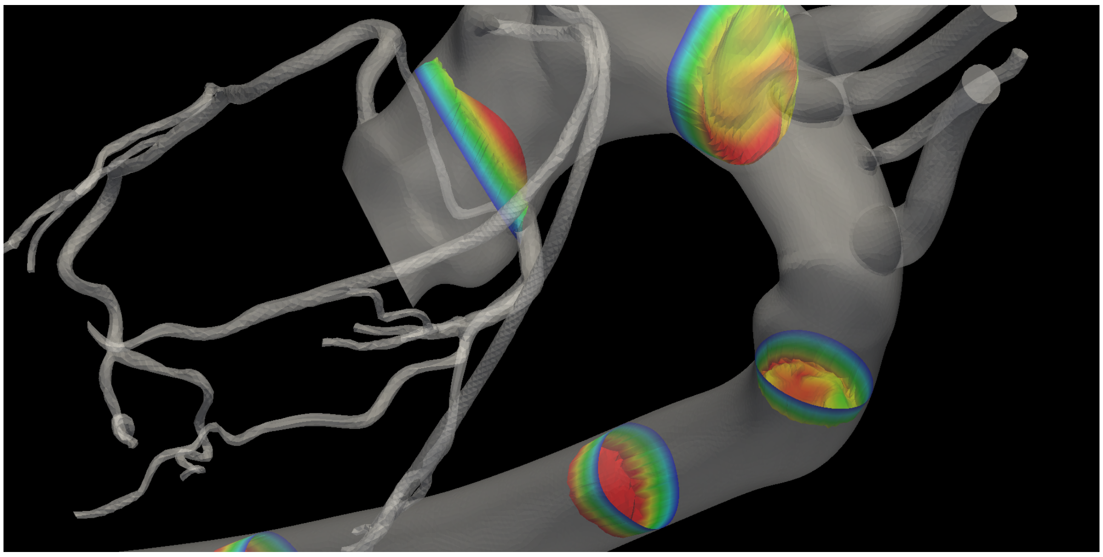
Screenshot of the OHIF DICOM Viewer

- The Medical Imaging Interaction Toolkit (MITK) is a free open-source software for the development of interactive medical image processing software.

Specifically for vascular analysis, there are a number of tools that should be considered:

- The *Vascular Modelling Toolkit* (VMTK) is a collection of libraries and tools for 3D reconstruction, geometric analysis, mesh generation and surface data analysis for image-based modeling of blood vessels.

- The *SimVascular*[18] software, which is a project of the SimTK initiative, allows to perform blood flow simulations and other analysis of biomedical data.



Simulation of the blood velocity distribution in CABG (SimVascular)

There are also comercial options available for specific purposes, such as:

- *RapidAI* for quantified CBCT perfusion imaging.
- *NeuroAI* for brain tumor diagnosis.
- *AVA* (from See-Mode), an AI analysis and reporting software of vascular ultrasound for the prevention of stroke.

These are examples of successful projects that are used. The open tools and software listed above have not been used for the development of this project, but could be considered for future implementations. As commented in the introduction, many of the procedures are standardized so the technological projects share a common roof under which to communicate and integrate.

The field is in constant growth and evolution and there are a lot of possibilities to study and options for the development of medical software. Some of the ideas that are currently being exploited are the application of data analysis and supervised learning methods for the analysis of medical data.

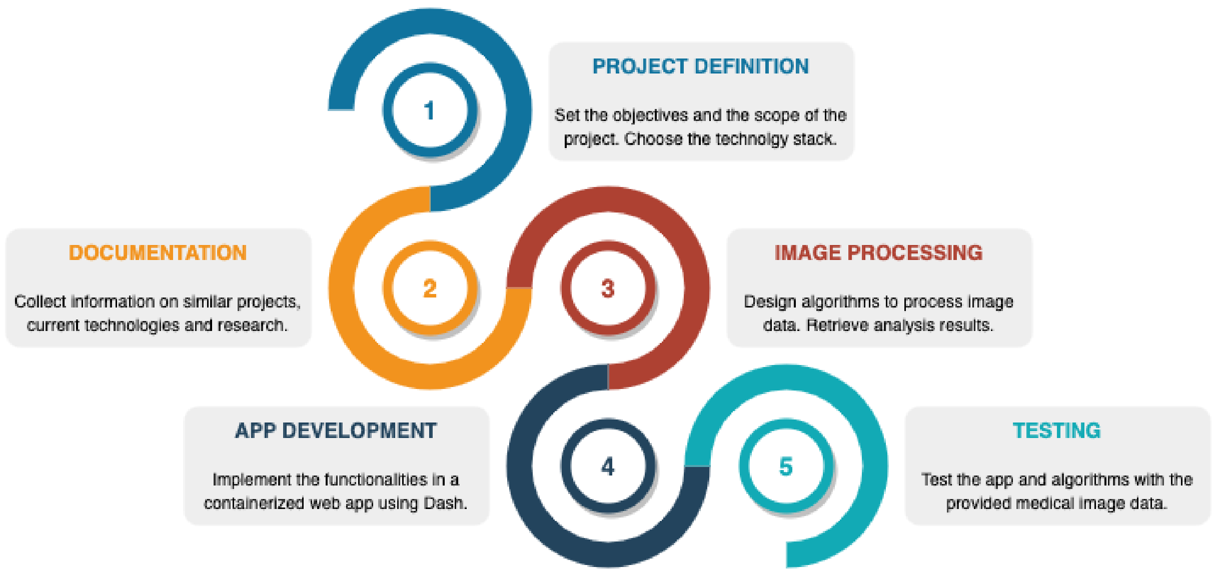
In this project, the time and budget constraints have limited the scope to the design of a simple prototype application making use of some open software libraries and platforms available.

There are a lot of open databases (i.e. MedPix from the National Institute of Health NIH) that provide anonymous medical data to use for study, modelling, testing, *etc*. With the irruption of *Big Data*, many different studies have been made in the field of biomedical data analysis, presenting and proposing new methods and applications covering a wide variety of topics of any medical area. These studies have been applied to research, diagnosis, prevention and treatment, and other uses.

3. METHODOLOGY

The objective of this project is to develop an open and custom application for the analysis of MRA medical images to serve as a diagnosis support tool for neurologists. The application is intended to be simple, with the ease of use as a priority. It is aimed to be multiplatform, easy to install and portable, so the users have a simple setup and direct access.

The development of this project consists of three main parts. In the first one the images are analyzed and processed to generate the desired outputs. In the second part, those image processing algorithms are implemented in a web application using the Python Dash framework. And in the third part, the provided data samples are tested in the application to evaluate its performance and usability.


1
PROJECT DEFINITION
Set the objectives and the scope of the project. Choose the technolgy stack.
2
DOCUMENTATION
Collect information on similar projects, current technologies and research.
3
IMAGE PROCESSING
Design algorithms to process image data. Retrieve analysis results.
4
APP DEVELOPMENT
Implement the functionalities in a containerized web app using Dash.
5
TESTING
Test the app and algorithms with the provided medical image data.

Roadmap of the project

This is the sequence followed because it allows to check first whether the image analysis is possible to be performed, and if it is viable to implement it in an application. In addition, by performing the image analysis first the algorithms can be rapidly tested on the data and the parameters tuned to get the best results.

3.1 Image Processing

The image data to be analyzed consists of MRA images of the brain vessels of different subjects, these samples are provided by the Hospital Sant Joan de Déu of Barcelona. For each subject there is a set of 8 frames sequentially acquired in a short time interval of 1.6 seconds. Each frame is encoded in a DICOM file.

The analysis is implemented using Python and some of its well known libraries.

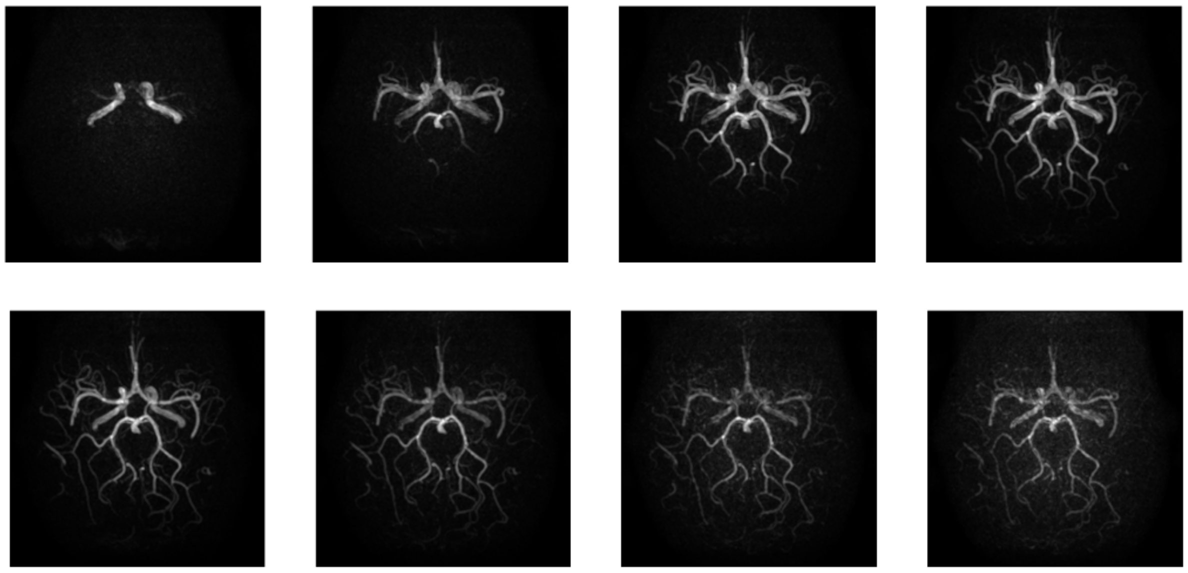
3.1.1 Data Loading

The DICOM files are loaded and read using a Python library called *pydicom* to extract the image pixel data, as follows:

```
import pydicom

dicom = pydicom.dcmread('filepath')
image = dicom.pixel_array
```

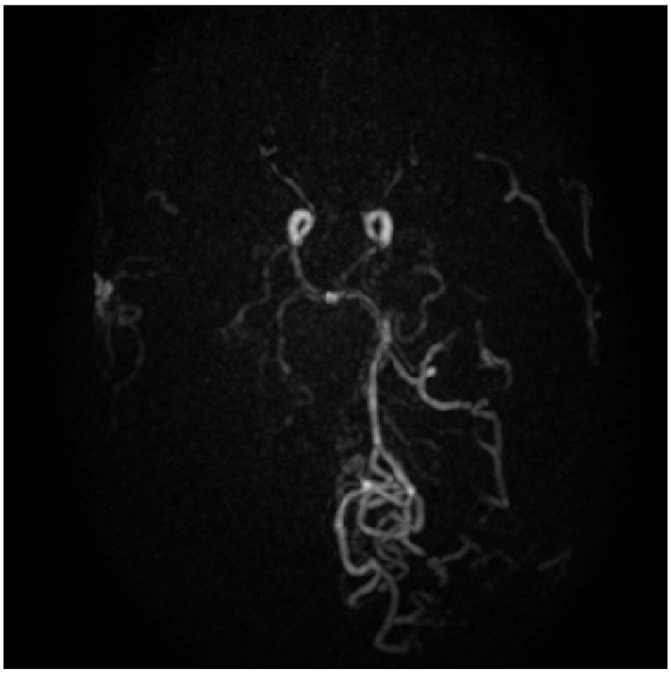
In the data provided, each DICOM file contains just one image/frame of the sequence for each subject. This is an example of a sequence of 8 frames of one given subject:



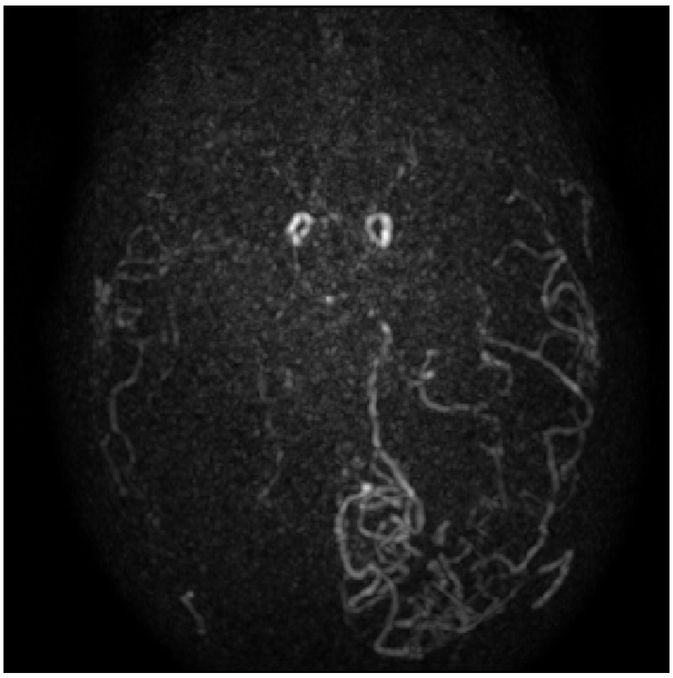
Sequence of MRA frames acquired for a subject

The sequence shows dynamically how the blood irrigates the vascular system of the brain. Such a sequence provides information to study the characteristics of the blood flow through the brain and the structure of its vascularity.

It is important to note that the first and last frames of the sequences are more affected by noise than the intermediate frames. For the pre-processing of the images this will be taken into account.



4th frame of a sequence showing less background noise

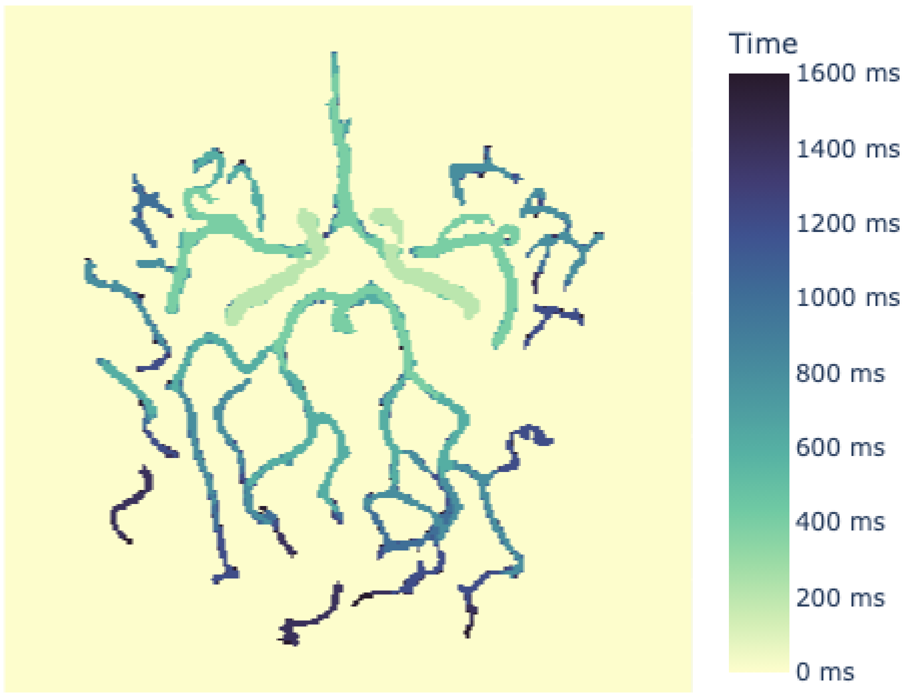


8th frame of a sequence showing higher level of noise

3.1.2 Expected Results

All the process for the image analysis attempts to retrieve two main results:

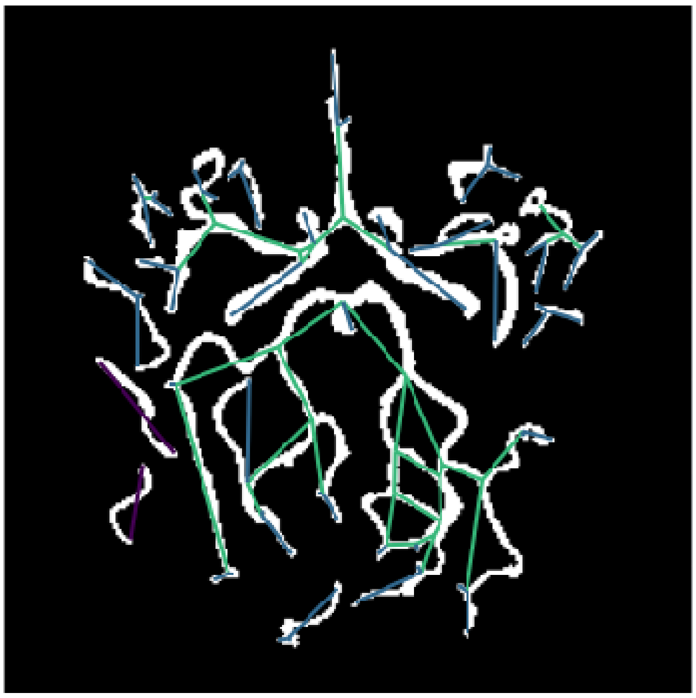
- An *isochronal map*, representing the blood flow in time just in one single picture:



Isochronal map result

To generate this map, the sequence frames are binarized and overlaid in temporal order.

- A *skeleton* map of the vascularity structure, generating also a dataset with the branch data:



Skeleton graph: branches colored by type (lines) and nodes (junctions)

To retrieve the skeleton data, the binarized frames are added to get a picture of all the vessels, then that picture (the total frame sum) is skeletonized to obtain the nodes and branches.

It is important to note that all the preprocessing will be automatically applied, without the user intervention or manual tuning, and that it should properly work for a variety of samples with different characteristics such as the amount of noise. In detriment of precision, the parameters should be tuned to work for all the possible inputs.

3.1.3 Preprocessing

The first step of the pre-processing is to filter the noise in order to facilitate the later binarization of the images for the detection of the vessels in each frame. As the first and last frames of the sequence are more affected by noise than the intermediate ones, the strategy followed to remove the noise is based on this fact. A median filter is applied with a specific mask. It must be remarked that the position of the vessels are considered not to change from frame to frame, only their visibility because of the blood flow. Having this into account the frames can be summed to enhance the vessels in the images and facilitate their detection or segmentation.

Once the noise is removed, a local threshold filter is applied as segmentation to obtain the binarized image, detecting and separating the vascular vessels from the background.

The steps followed for the preprocessing are summarized in the next diagram.

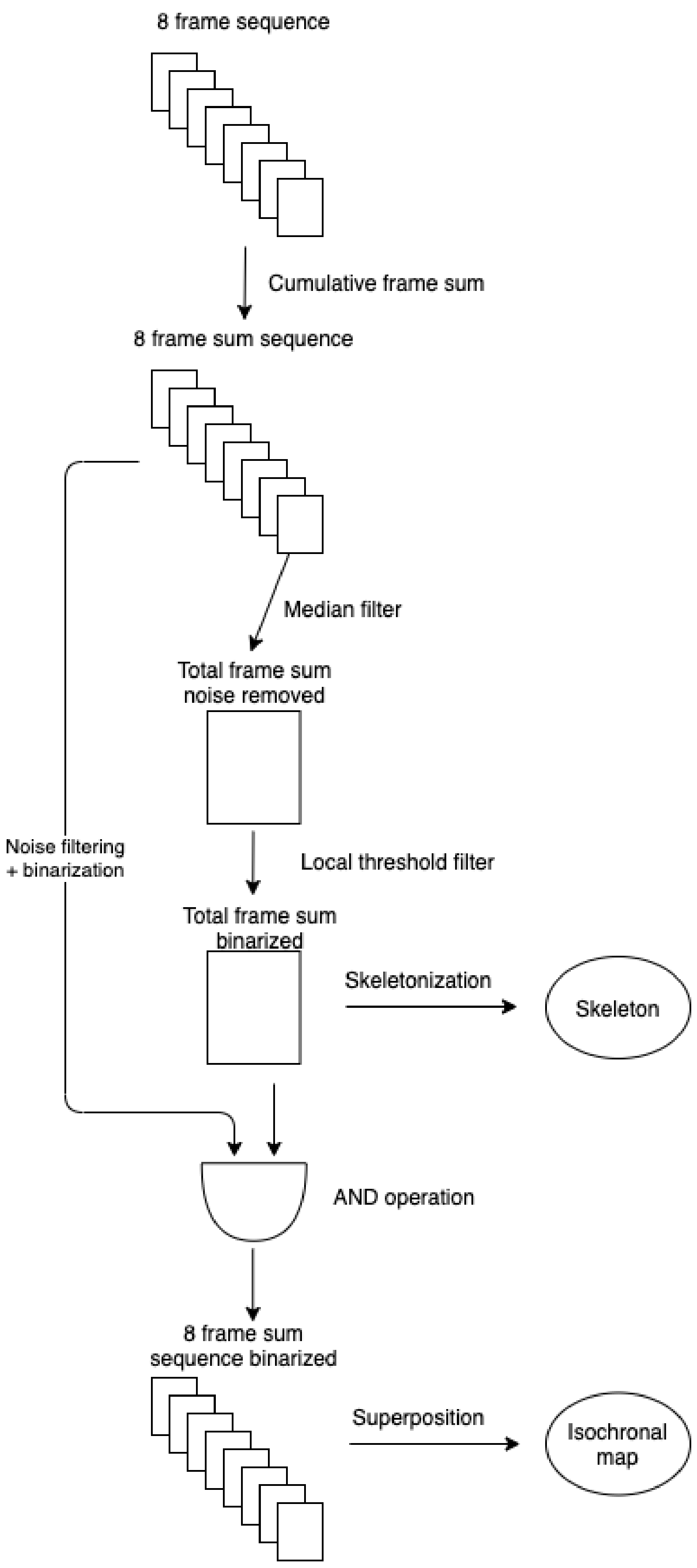


Image processing diagram

1. The 8 frames are cumulatively summed, that means the 1st frame of the output is the 1st frame of the input, the 2nd frame of the output is the sum of the 1st and 2nd frames of the input… and so for the rest of the frames, until the last output frame is the total sum of all the input frames of the given sequence.
2. That total frame sum is filtered using a median filter for noise removal, and then binarized (segmentation) using a local threshold filter (also the small artifacts are detected and removed). These filters use custom and manually tuned parameter values that are used for processing all the data samples provided. The binarized total frame sum is skeletonized to retrieve the skeleton.
3. Each output frame obtained in step 1 is filtered for noise removal and binarized, then an `AND` operation is made with the binarized total frame sum to get rid of unwanted artifacts. The result is a sequence with the binarized cumulative frames. Then the isochronal map is built from this sequence.

3.1.3.1 Noise Filtering

The noise removal is implemented by making use of a median filter with a specific mask. The images are treated and processed as *numpy* arrays, numpy is a mathematical processing library for Python. The Python library *scikit-image* (for image processing) is used for applying the filter. The function is coded as follows:

```
import numpy as np

img_filtered = median(img,
                      selem=[[0.3, 0.7, 0.3], [0.7, 1, 0.7], [0.3, 0.7, 0.3]])
```

A simple median filter is chosen because it removes the noise while keeping the edges of the image objects relatively sharp, facilitating the posterior segmentation of the image. A basic gaussian filter was also tested but it is not idoneous for the processing of these images as it blurs the edges and reduces contrast. There are other filters that could be useful for the task, such as the Hessian or Frangi filters which are used for ridge detection, but the median filter was chosen for simplicity. Some different filter masks were tested but a more exhaustive exploration could be performed for optimization.

3.1.3.2 Segmentation (Binarization)

The image is segmented in two parts (binarization), one being the background and the other the blood vessels. This step is critical as an accurate detection of the vessels will have an important impact on the final analysis results.

The binarization is applied making use of a function of scikit-image called *threshold_local*. This function allows to specify a custom function to calculate a local threshold around each pixel of the image in order to binarize it.

```
from skimage.filters import threshold_local, median

def binarize(img):
    def _filter_func(buffer):
        mean = np.mean(buffer)
        std = np.std(buffer)
        return np.maximum(1.1 * mean, 1.1 * std)
    return img > threshold_local(img, block_size=13, offset=-50,
                                 method='generic', mode='mirror', param=_filter_func)
```

After the binarization, the function *remove_small_objects()* of the *morphology* package of scikit-image is used to remove the small artifacts left.

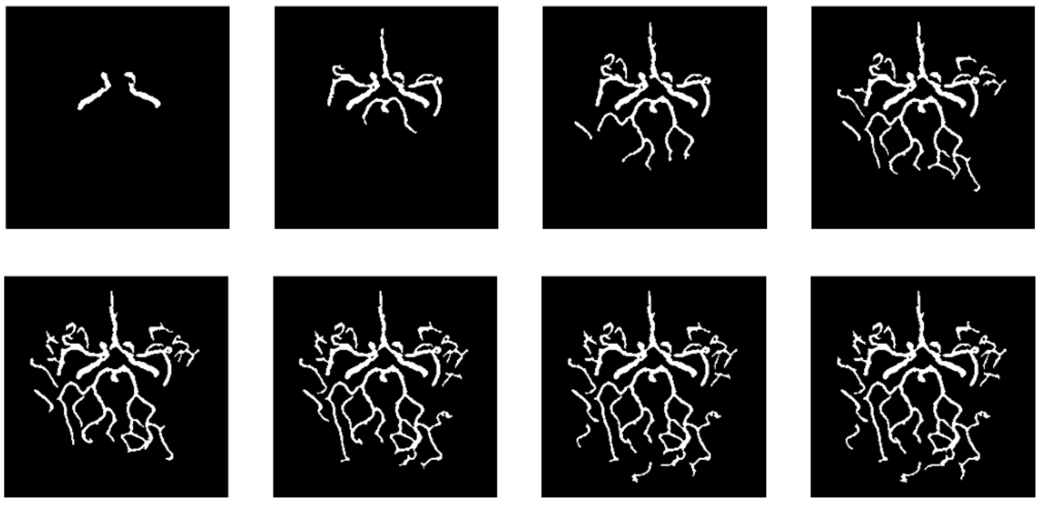
The local threshold segmentation algorithm is chosen because it is one of the simplest ways to binarize the image, by setting up a threshold. By using such a region based (local) method the vessels showing different intensity values can be detected at once.

There are many other segmentation methods that could be implemented such as clustering methods (K-means), edge detection methods, AI based techniques, *etc*. There are also ridge detection filters, such as the Hessian or Frangi filters, that could be useful to detect and separate the vascular system from the background. However this is a more complex approach, more difficult to tune.

These possible methods were not exhaustively tested, but the local threshold parameters were tuned to get acceptable results for all the analyzed images.

3.1.4 Isochronal Map

Once the frames have been processed and binarized, the isochronal map can be built. The isochronal map represents in one picture how the blood flow irrigates the vessels in temporal order. Each binarized frame has the vessels that have been irrigated upon that instant of time, the time lapse between frames is 200 ms.



Binarized frames showing the blood flow evolution

These frames can be superposed and coloured to build the isochronal map, the code implemented is:

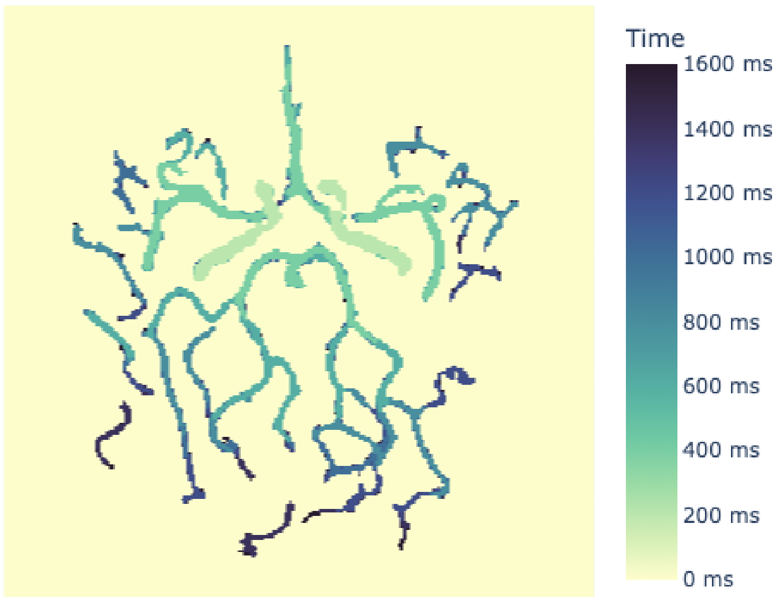
```
import plotly.express as px

def isochronal_map(frames_bin):
    size = frames_bin[0].shape
    map_array = np.zeros(size, dtype=np.uint16)
    frames_bin = np.flip(frames_bin, axis=0)
    step = 200  # time in ms between frames
    time = frames_bin.shape[0] * step
    for frame in frames_bin:
        map_array[frame] = time
        time -= step

    fig = px.imshow(map_array, color_continuous_scale='deep',
                    width=600)
    fig.update_xaxes(showticklabels=False)
    fig.update_yaxes(showticklabels=False)
    fig.update_layout(
        margin=dict(l=0, r=0, b=0, t=0),
        coloraxis_colorbar=dict(
            title='Time',
            ticksuffix=' ms'
        )
    )

    return fig
```

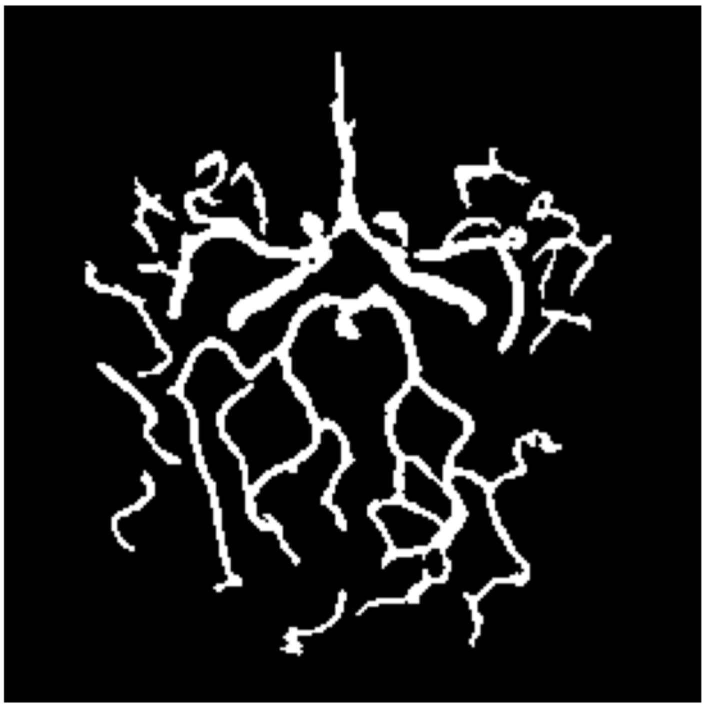
In order to generate the plot, the code makes use of a graphical library of Python called *Plotly*. The resulting map is:



Isochronal map

3.1.5 Skeletonization

In order to retrieve the skeleton data of the vascular structure, the binarized total frame sum is processed. An example of the binarized total frame sum is shown in the next figure:



Total frame sum binarized

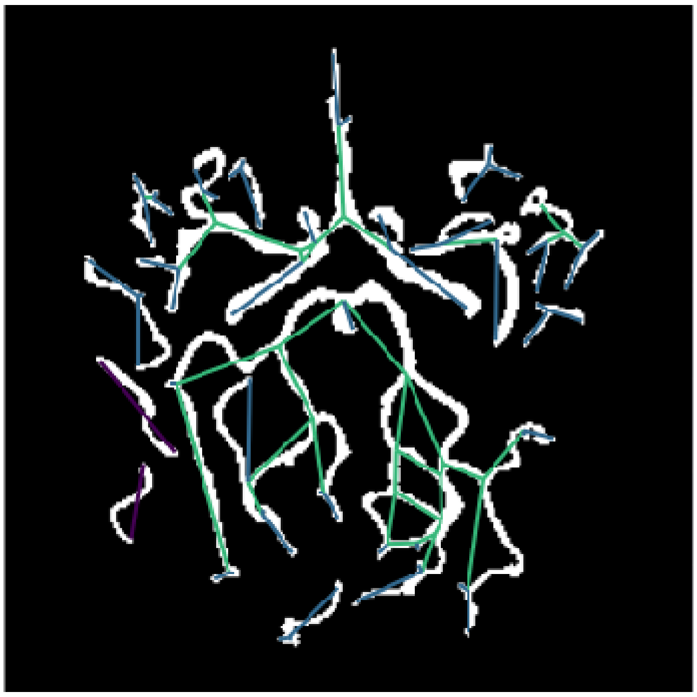
A Python function from the *morphology* package of the library scikit-image is used to generate the skeleton of the image. The skeleton is simply a pixel-wide representation of the branches and nodes of the binarized objects contained in the image. It is generated by recurrent operations of erosion and dilation of the image, until there is only a pixel wide structure, the skeleton. Then in that skeleton it is easy to identify the nodes (or joints) and branches and their attributes. To do so, another Python library called *skan*[19] is used, which automatically analyzes the input skeleton, plots the joints and branches of it, and generates a dataset with the branch data.

The implemented code is:

```
from skan import Skeleton, summarize, draw

def skeletonize(I_bin):
    skeleton = morphology.skeletonize(I_bin)
    branch_data = summarize(Skeleton(skeleton))
    fig = plt.figure(figsize=(10,20))
    draw.overlay_euclidean_skeleton_2d(I_bin, branch_data,
                         skeleton_color_source='branch-type', axes=plt.gca())
    plt.show()
    return branch_data
```

The plot generated is:



Skeleton plot showing branches and joints

The dataset generated with the branch data contains a table with different attributes for each branch, such as the branch distance, type, end points coordinates, *etc*. Some of those attributes are taken for the analysis, in addition, the tortuosity is calculated as the ratio between the branch distance and the euclidean branch distance. The tortuosity is an important biomarker that measures the curvature of a branch. A final dataframe is built using the *Pandas* library of Python, containing the following data:

branch	subject	skeleton-id	branch-type	branch-distance	euclidean-dist.	tortuosity
0	case	1	1	27.12	25.44	1.066
1	case	1	1	5.52	4.95	1.115
2	case	1	2	33.10	31.46	1.052
3	case	2	1	8.57	6.82	1.257

This data can be used to generate plots that compare the branch attributes for different samples.

3.2 Application Development

Once the image processing algorithms have been evaluated and validated, the next step is to develop an application in which to implement those algorithms. This application will serve the medical professional users in the exploration of the image data for different purposes such as diagnosis, treatment evaluation, *etc*.

The requirements for the application are:

- Simplicity of use: The application should be intuitive for its immediate understanding.
- Portability and easy setup: The application is meant to be multi-platform and packaged for sharing and direct installation without having to cope with external dependencies.

- Modularity and customization: It is aimed to serve as a template for future functionality enhancements or even for the development of other applications with different purposes.
- Open web app: The application should be open-sourced and make use of open-source software and web technologies, it should run in the web browser.

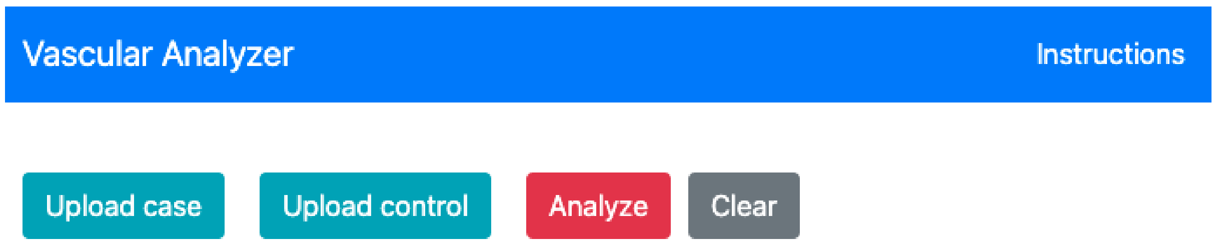
These objectives can be fulfilled by choosing the right technology stack to develop the application.

The technology stack chosen is based on the Python programming language. Apart from the Python libraries mentioned in the Image Processing section, used for the image processing algorithms, the development of the application makes use of other frameworks.

The platform chosen for the design of the web application is Dash. Dash was introduced in the Introduction, it is an open source Python framework provided by Plotly for the development of web applications. It provides abstraction from many of the technologies required to implement a web app. It also includes a library of a variety of modular components serving different purposes such as forms, graphs, menus, *etc*.

In order to package and deliver the application, and to abstract the management of the required dependencies, the chosen option is to containerize the application using a software called Docker. Docker was introduced in the Introduction, it is used to package the application in a unit of software called ‘container’ which includes every piece of software required by the application to run. It isolates the app from the rest of the operating system and as Docker runs in many different operating systems it directly makes the app a multi-platform and portable one, without the need for any installation as long as Docker is installed.

The functionalities of the app allow the user to load two sets of DICOM files from two different subjects, case and control, for comparison. Once the files are loaded the image data is automatically extracted and the frames are visualized in the dashboard. Then the user can start manually the analysis of the data that processes the images to generate all the results: the isochronal map, the skeleton and the graph plots. Once the analysis is finished the user can download any of the plots and the dataset containing the branch data for the subjects loaded.



Screenshot of the Dash application

The app interface consists basically of a dashboard with some basic user interaction functionalities:

- There is a top bar showing the app name and an option to open the instructions, that shows a box with the instructions on how to use the app.
- All the controls are included at the top, with 5 different action buttons:
    - 2 to load the DICOM files for the two different subjects
    - 1 to start the analysis of the data.
    - 1 to clear the results and the files loaded to reset the app.

- 1 to download the branch data as a CSV file, this button appears automatically once the analysis is complete.
- A grid containing and showing the results.
- The resulting graphs are interactive; with zoom options, axis selection, *etc*.

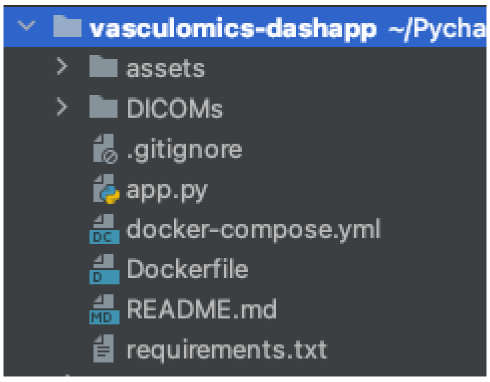
The Dash framework is used to define the layout of the web app, with all its visual and interactive HTML components such as bars, buttons, containers, *etc*. The functionalities for the user interaction are also defined in Dash, by the use of *callbacks*.

Once the application is developed, it is configured to be containerized using Docker.

The development process is summarized now, step by step, in a tutorial style.

3.2.1 Python Project

The first step is to create a Python project, with a directory structure such as the one shown in the next figure:



```
vasculomics-dashapp ~/Pycha
  assets
  DICOMs
  .gitignore
  app.py
  docker-compose.yml
  Dockerfile
  README.md
  requirements.txt
```

Project directory structure

Then a Python virtual environment is created, in order to install the python packages required by the app to run. This can be easily setup by using a *requirements.txt* file that lists all the dependencies:

requirements.txt

```
dash
dash_bootstrap_components
flask-caching
matplotlib
numba
numpy
pandas
pillow
plotly
pydicom
scikit-image
skan
```

The terminal instructions to create the virtual environment (using the *Conda* package manager) and to install the dependencies are:

```
conda create --name myenv python=3.6
conda activate myenv
pip install -r requirements.txt
```

For the development of this application, the Python version 3.6 is required in order to be able to use the library *skan*.

The next step is to create the Python script *app.py*, that contains the app code.

3.2.2 Import Libraries

The Python libraries used to run the app are imported at the beginning of the script. Here are included both the image processing packages and the ones for Dash.

```
import dash
from dash.dependencies import Input, Output, State
import dash_core_components as dcc
import dash_html_components as html
import dash_bootstrap_components as dbc
import plotly.express as px
from flask_caching import Cache
import base64
import io
import pydicom
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from skimage import morphology
from skimage.filters import threshold_local, median
from skan import Skeleton, summarize, draw
from assets.instructions import generate_instructions
```

One important thing is to set the Matplotlib graphics rendering backend to a non-interactive mode, because it makes the Dash runtime crash:

```
plt.switch_backend('Agg')
```

3.2.3 App Instantiation

The next step is to include the application declaration, which serves to define some parameters such as the app title, callback exception options, and others.

```
external_stylesheets = [dbc.themes.BOOTSTRAP]
app = dash.Dash(__name__, external_stylesheets=external_stylesheets,
                title='Vascular Analyzer', update_title='Processing...',
                suppress_callback_exceptions=True)
```

The `Dash` class of the `dash` library is used to instantiate the app. The `update_title` corresponds to the browser's tab title shown during the execution of the app callbacks. The `external_stylesheets` attribute serves to target a CSS file that specifies the style of the web components.

The app makes use of the browser's cache memory, making use of the `flask_caching.Cache` library:

```
CACHE_CONFIG = {
    'CACHE_TYPE': 'filesystem',
    'CACHE_DIR': './cache'
}
cache = Cache()
cache.init_app(app.server, config=CACHE_CONFIG)
cache.clear()
```

The cache is cleared when the app is initialized. The filesystem is used to store the cached values.

3.2.4 Dash Layout Definition

The next step is to define the Dash layout for the application, that means specifying the HTML elements that compose the interface of the web app, and their disposition and behavior on the screen.

The basic elements of the layout are taken from the `dash_bootstrap_components` which contains predefined components with an intrinsic Bootstrap design and functionalities. *Bootstrap* is a web styling framework intended for responsive web development. The `dash_html_components` includes all the standard HTML tags and the `dash_core_components` is the standard library with predefined components provided by Dash.

```
app.layout = html.Div([
    navigation_bar,
    instructions,
    analysis_grid,

    dcc.Store(id='frames-case', storage_type='memory'),
    dcc.Store(id='frames-control', storage_type='memory'),
    dcc.Store(id='signal-case', storage_type='memory'),
    dcc.Store(id='signal-control', storage_type='memory'),
    dcc.Store(id='branch-data-case', storage_type='memory'),
    dcc.Store(id='branch-data-control', storage_type='memory'),
    dcc.Store(id='branch-data', storage_type='memory'),
    dcc.Store(id='signal-box-axes', storage_type='memory')
])
```

The layout is defined in the `app.layout` object. Everything is inserted in a HTML container (`Div`). The `dcc.Store` component is used to store JSON data in the browser, in order to share data between callbacks.

Each layout component that has user interaction functionalities, needs to be uniquely identified by specifying the `id` tag of the HTML element. This allows to reference those elements in the callback functions.

The `navigation_bar` is defined as:

```
navigation_bar = dbc.NavbarSimple(
    children=[
        dbc.NavItem(dbc.Button("Instructions", id='open-instructions',
                    color='primary')),
    ],
    brand="Vascular Analyzer",
    brand_href="#",
    color="primary",
    dark=True,
)
```

The `instructions` modal box as:

```
instructions = dbc.Modal(
    [
        dbc.ModalHeader("Instructions"),
        dbc.ModalBody(
            generate_instructions(),
            style={'text-align':'justify'}
        ),
        dbc.ModalFooter(dbc.Button("Close", id="close-instructions",
                        className="ml-auto")),
    ],
    id="instructions-modal",
    scrollable=True,
)
```

The `generate_instructions` function returns the instructions in Markdown format, which Dash can also render.

The `analysis_grid` is the ‘biggest’ component in the layout:

```
analysis_grid = dbc.Container(
    [
        dbc.Row(
            [
                dbc.Col(
                    [
                        dcc.Upload(id='upload-case',
                                   children=dbc.Button("Upload
                                                        case", color='info'),
                                   multiple=True, className='upload'),
                        dcc.Upload(id='upload-control',
                                   children=dbc.Button("Upload control",
                                                       color='info'),
                                   multiple=True, className='upload'),
                        dbc.Button("Analyze", id='start-analysis',
                                   color='danger'),
                        dbc.Button("Clear", id='clear-data', color='secondary',
                                   type='reset'),
                        html.Div([
                            dbc.Button("Download CSV", id='download-button',
                                       color='success',
                                       style={'visibility':'hidden'}),
                            dcc.Download(id='download-csv')
                        ])
                    ]
                )
            ]
        ),
        dbc.Row([dbc.Col(id='case-original'), dbc.Col(id='control-original')]),
        html.Div('Isochronal map', className='row-title'),
        dbc.Row([dbc.Col(id='case-map'), dbc.Col(id='control-map')]),
        html.Div('Skeleton', className='row-title'),
        dbc.Row([dbc.Col(id='case-skeleton'), dbc.Col(id='control-skeleton')]),
        html.Div('Graphs', className='row-title'),
        dbc.Row(dbc.Col(id='box-plot'))
    ],
    fluid=False
)
```

The `dbc.Container` sets a Bootstrap grid that is divided in rows, each row (`dbc.Row`) containing different columns (`dbc.Col`). The first row is filled with the control buttons for the user. The rest of the rows show consecutively different results, each row is divided in two columns: one to show the result for the 'case' subject, and the other column to show the result for the 'control' subject.

The Dash libraries provide many more options and possibilities with different components available with different built in functionalities for a wide variety purposes.

3.2.5 Dash Callbacks

Once the initial layout of the dashboard is defined, the next step is to implement the interactive functionality by means of *callback functions*. This is an intrinsic characteristic of the functioning of Dash. The user interaction is defined by events and changes in the layout elements. These events trigger the so-called callback functions that can perform any function including making changes in other components of the layout (and therefore maybe calling other callbacks).

Toggle Instructions

To begin with, let's see the callback defined to toggle/show the instructions modal:

```
@app.callback(Output("instructions-modal", "is_open"),
              [Input("open-instructions", "n_clicks"),
               Input("close-instructions", "n_clicks")],
              [State("instructions-modal", "is_open")])
def toggle_instructions(n1, n2, is_open):
    if n1 or n2:
        return not is_open
    return is_open
```

A callback is defined using the Python decorator `@app.callback`. The `Input`, `Output` and `State` objects of the `dash.dependencies` package are used to specify the trigger events of the callback, and the layout element that is going to be changed as output.

This callback function runs when:

- The user clicks on the `open-instructions` button.
- The user clicks on the `close-instructions` button.

This corresponds to the two `Input` arguments defined in the callback decorator. The callback is triggered when the `nclicks` attribute of the `open-instructions` or `close-instructions` layout elements changes.

The callback function, which can have any valid custom name, is named `toggle_instructions`. This function has as arguments the values of the attributes of the `Input` and `State` objects included, in order of declaration (`n1`, `n2`, `is_open`). The `State` objects are used to access the attribute values of certain layout elements, but are not involved in the callback triggering.

The function toggles the `is_open` variable value and returns it.

The returned object will be the new value assigned to the `is_open` attribute of the `Output` layout element `instructions-modal`. And this is the way in which the layout is updated, in this case the instructions prompt is open or closed.

This is the basic structure and behavior of a callback. There are more advanced features available to the programmer to manage more aspects, such as identifying which is the `Input` element that has triggered the code (that is done with the `dash.callback_context` built in object).

Load DICOMs

The callbacks for loading the files of the case and control subjects are triggered once the upload buttons are clicked and some files are loaded:

```
@app.callback(Output('frames-case', 'data'),
              Input('upload-case', 'contents'),
              State('upload-case', 'filename'))
def load_case(contents, filenames):
    if contents is None:
        return None
    files = zip(contents, filenames)
    files = sorted(files, key=lambda t: t[1])
    frames = parse_dicoms(files)
    return frames
```

The `frames` object returned is a list containing the frames of the sequence in order and as pixel 2D Numpy arrays. The result is passed to the `dcc.Store` with `id='frames-case'` element of the layout, that is used to store the value and use it in other functions. This object is generated in the `parse_dicoms` function:

```
def parse_dicoms(files):
    frames = []
    for c, n in files:
        content_type, content_string = c.split(',')
        dicom = base64.b64decode(content_string)
        ds = pydicom.dcmread(io.BytesIO(dicom))
        frames.append(ds.pixel_array)
    return frames
```

This function makes use of the `pydicom` library to read the DICOM files and the `base64` and `io` to decode the input `files`.

The application does not make use of global variables as Dash and this is because Dash is a stateless framework designed to work in a multi-user environment where multiple users may view the app at the same time and will have independent sessions. Dash also can run with multiple python workers so that callbacks can be executed in parallel, but their memory is not shared. If global variables were used, a user session could affect the global values for the next session, also a modification of a global variable by one worker would not be shared or applied to other workers running in parallel.

Show Frames

Once the data has been loaded, the change in the `frames-case` data triggers a callback to show the frames in the dashboard:

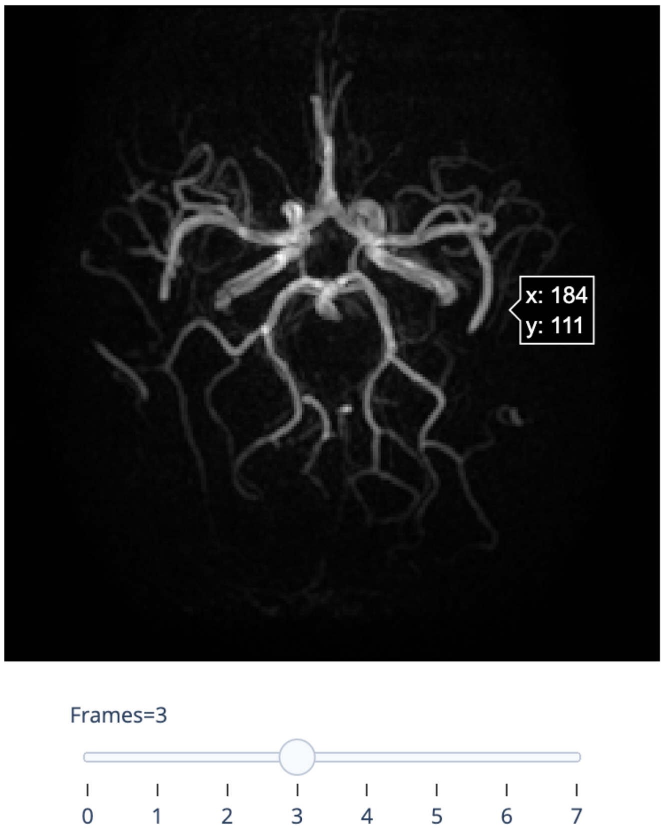
```
@app.callback(Output('case-original', 'children'),
              Input('frames-case', 'data'))
def show_case(data):
    if data is None:
        return None
    frames = np.array(data)
    fig = plot_original(frames)
    graph = html.Div([
        html.Div('Case', className='frames-title'),
        dcc.Graph(figure=fig)
    ])
    return graph
```

The object returned is a graph containing a figure generated by using the `plotly.express` plotting library, this is done in the `plot_original` function:

```
def plot_original(frames):
    fig = px.imshow(np.array(frames), animation_frame=0, binary_string=True,
                    labels=dict(animation_frame="Frames"), width=450, height=540)
    fig.update_xaxes(showticklabels=False)
    fig.update_yaxes(showticklabels=False)
    fig.update_layout(
        margin=dict(l=40, r=40, b=0, t=0),
        updatemenus=[dict(visible=False)],
        sliders=[dict(
            len=0.8,
            pad=dict(t=10)
        )]
    )

    return fig
```

The resulting graph is:



Frames displayed with slider

Process Data

The next step is to begin the analysis by clicking the 'Analysis' button. The callback associated to it is:

```
@app.callback(Output('signal-case', 'data'),
              Input('start-analysis', 'n_clicks'),
              Input('clear-data', 'n_clicks'),
              State('frames-case', 'data'))
def processing_case(clicks1, clicks2, data):
    if data is None:
        return None, None

    ctx = dash.callback_context

    if not ctx.triggered:
        return None, None
    else:
        button_id = ctx.triggered[0]['prop_id'].split('.')[0]

    if button_id == 'clear-data':
        return None, None

    frames = np.array(data)
    [frames_bin, frame_sum_bin] = binarize(frames)
    return frames_bin, frame_sum_bin
```

In this callback the `dash.callback_context` allows to see which is the `Input` that has triggered the callback.

This function calls the binarize function to binarize the frames, returning a list with two items: first a list containing the binarized frames, and second the pixel array representing the total frame sum binarized.

The binarize function is described in the Image Processing section. It makes use of the `@cache.memoize` decorator, which saves the results of the function calls in the cache.

Plot Isochronal Map

Once the frames are processed, they are stored in a `dcc.Store` layout element and the callbacks to generate and show the isochronal map and skeletonization results are triggered:

```
@app.callback(Output('case-map', 'children'),
              Input('signal-case', 'data'))
def show_case_map(data):
    if data[0] is None:
        return None
    frames_bin = np.array(data[0])
    fig = isochronal_map(frames_bin)
    return dcc.Graph(figure=fig)
```

The isochronal map figure is generated by calling the `isochronal_map` function, which was explained in the Image Processing section.

Visualize Skeleton

The skeleton graph is also generated in the following callback:

```
@app.callback(Output('case-skeleton', 'children'),
              Output('branch-data-case', 'data'),
              Input('signal-case', 'data'))
def show_case_skl(data):
    if data[1] is None:
        return None, None
    frame_sum_bin = np.array(data[1])
    branch_data, fig = skeletonize(frame_sum_bin)
    branch_data['subject'] = 'case'
    branch_data = [branch_data.columns.values.tolist()] +
                  branch_data.values.tolist()
    return dcc.Graph(figure=fig), branch_data
```

The results include both the plot figure that is passed to the `case-skeleton` output, which is a HTML container, and a list containing the branch data that is stored in the `dcc.Store` with `id=branch-data-case`.

The skeleton is generated with the `skeletonize` function that returns the skeleton graph and the branch data. The function that was introduced in the Image Processing section needs some modifications to work properly in Dash as the `skan` library makes use of the Matplotlib library:

```
def skeletonize(I_bin):
    if I_bin is None:
        return None, None
    skeleton = morphology.skeletonize(I_bin)
    branch_data = summarize(Skeleton(skeleton))

    mpl_fig = plt.figure()
    draw.overlay_euclidean_skeleton_2d(I_bin, branch_data,
skeleton_color_source='branch-type', axes=plt.gca())
    buf = io.BytesIO()
    plt.savefig(buf, format='png', bbox_inches='tight', pad_inches=0)
    buf.seek(0)  # rewind file
    img = Image.open(buf)
    img_array = np.array(img)

    fig = px.imshow(img_array, width=450, height=343, binary_string=True)
    fig.update_xaxes(showticklabels=False)
    fig.update_yaxes(showticklabels=False)
    fig.update_layout(
        margin=dict(l=0, r=0, b=0, t=0),
        coloraxis_showscale=False
    )

    buf.close()
    plt.close(mpl_fig)

    return branch_data, fig
```

The skeleton is generated using the `morphology.skeletonize` function of the `scikit-image` library. This is just a pixel wide representation of the binary image. Then that skeleton is analyzed using the `Skeleton` and `summarize` functions of the skan library.

The Matplotlib figure generated by `draw.overlay_euclidean_skeleton_2d` is rendered to an image and written in a buffer (`buf`) that then is read (using the `Image.open` function of the Pillow library) to retrieve the data. From this image data a Plotly figure is created using the plotting function `imshow`.

Build Dataset

The `branch_data` returned by `skan` is a `pandas` dataframe containing data for each branch in the skeleton, such as the end point coordinates, the length, the euclidean length, *etc*. This is explained in the Image Processing section. The data should be processed to extract some parameters or biomarkers such as the branch tortuosity (curvature), and in order to join the datasets of the two subjects in a single one to be used for the plots.

```
@app.callback(Output('branch-data', 'data'),
              Input('branch-data-case', 'data'),
              Input('branch-data-control', 'data'))
def build_dataframe(data_case, data_control):
    if data_case is None and data_control is None:
        return None
    elif data_control is None:
        branch_data = pd.DataFrame(data_case[:][1:], columns=data_case[:][0])
    elif data_case is None:
        branch_data = pd.DataFrame(data_control[:][1:],
                                    columns=data_control[:][0])
    else:
        data_case = pd.DataFrame(data_case[:][1:], columns=data_case[:][0])
        data_control = pd.DataFrame(data_control[:][1:],
                                     columns=data_control[:][0])
        branch_data = pd.concat([data_control, data_case])

    branch_data = branch_data[['subject', 'skeleton-id', 'branch-type',
                               'branch-distance', 'euclidean-distance']]
    branch_data['tortuosity'] = branch_data['branch-distance'] /
                                branch_data['euclidean-distance']

    branch_data = [branch_data.columns.values.tolist()] +
                  branch_data.values.tolist()
    return branch_data
```

The callback is executed when the `dcc.Store` components, that store the branch data of the subjects, are updated. The output dataset is saved in another `dcc.Store` element. It is important to note that the `pandas` dataframes are converted to lists before being returned, this is because dcc.Store can store Python primitive lists but not `pandas` objects.

Download CSV

Once the final dataset is built an option to download the data as a CSV file is enabled. This option appears by displaying a button that was initially hidden in the layout.

```
@app.callback(Output("download-csv", "data"),
              Input("download-button", "n_clicks"),
              State('branch-data', 'data'),
              prevent_initial_call=True)
def download_csv(n_clicks, branch_data):
    if branch_data is None:
        return None
    branch_data = pd.DataFrame(branch_data[:][1:], columns=branch_data[:][0])
    return dcc.send_data_frame(branch_data.to_csv, "branch_data.csv")


@app.callback(Output('download-button', 'style'),
              Input('branch-data', 'data'),
              prevent_initial_call=True)
def toggle_download(branch_data):
    if branch_data is None:
        return {'visibility':'hidden'}
    else:
        return {'visibility':'visible'}
```

The `pandas` library allows to generate a CSV file with the `to_csv` function. The visibility of the download button is toggled by modifying the style parameter using CSS clauses.

The `prevent_inital_call` callback attribute set to `True`, makes the callback not to execute when the app is initialized. By default, all the callbacks execute when the app first loads, but as seen, this can be configured.

Box Plot

The final dataset is used to generate a box-plot that reflects the branch data for a graphical comparison of the parameter values retrieved for each subject.

```
@app.callback(Output('box-plot', 'children'),
              Input('branch-data', 'data'),
              Input('signal-box-axes', 'data'))
def box_plot(branch_data, data_axes):
    if branch_data is None:
        return None

    branch_data = pd.DataFrame(branch_data[:][1:], columns=branch_data[:][0])

    if data_axes is None:
        data_axes = [branch_data.columns[0], branch_data.columns[3]]
    elif data_axes[0] is None:
        data_axes[0] = branch_data.columns[0]
    elif data_axes[1] is None:
        data_axes[1] = branch_data.columns[3]

    fig = px.box(branch_data, x=data_axes[0], y=data_axes[1], color='subject')

    plot = html.Div([
        dcc.Graph(figure=fig),
        generate_dropdown(branch_data, "X axis", 'x-bar', data_axes[0],
                          "X-axis"),
        generate_dropdown(branch_data, "Y axis", 'y-bar', data_axes[1], "Y-axis")
    ])

    return plot


@app.callback(Output('signal-box-axes', 'data'),
              Input('x-bar', 'value'),
              Input('y-bar', 'value'),
              prevent_initial_call=True)
def select_box_axes(x, y):
    return [x, y]


def generate_dropdown(branch_data, title, id, value, placeholder):
    if branch_data is None:
        return None

    @cache.memoize()
    def generate_options(branch_data):
        dropdown_options = []
        for column in branch_data.columns:
            item = {
                'label': column,
                'value': column
            }
            dropdown_options.append(item)
        return dropdown_options

    return html.Div([title, dcc.Dropdown(id=id,
                     options=generate_options(branch_data),
                     value=value, placeholder=placeholder)],
                     className='dropdown-div')
```

This code generates a box-plot using the `plotly express` (`px.box`) library. It also generates two dropdown elements automatically (`generate_dropdown`) that allow choosing the axis parameters, the plot is automatically updated (`select_box_axes`). The results are coloured by subject.

Clear Data

Finally, there is another callback added that is used to clear all the data loaded in order to be able to reset the dashboard layout to begin a new analysis.

```
@app.callback(Output('upload-case', 'contents'),
              Output('upload-control', 'contents'),
              Input('clear-data', 'n_clicks'))
def clear_data(clicks):
    return None, None
```

The `clear-data` button click `Input` is also used in the processing callbacks (i.e. `processing_case`) in order to remove the data stored and that consequently update and clear the graphs/images loaded.

3.2.6 Executing The Application

The final step is to run the app, a method of the `app` Dash object (declared when the app was instantiated) is invoked to do so, different arguments are passed to launch the web server:

```
if __name__ == '__main__':
    app.run_server(host='0.0.0.0', debug=True, port=8050)
```

The host IP address is set to `0.0.0.0` to make the web server accessible from docker. By default, the port configured is `8050`.

The `debug=True` option is used when developing and debugging the application, it allows to make code modifications while running the server so the changes can be immediately tested.

3.2.7 Assets & Style

The contents in the `assets` folder created in the project are automatically indexed by Dash and can be used in the app by their file names without referring to their path. The `favicon.ico` file included is used by Dash to render the browser icon of the web app.

The `images` folder and the `instructions.py` script are used to render the instructions, the script is imported in the `app.py` code.

Finally there is another file in the assets folder, which is the style.css file that defines the CSS declarations to specify the display style of some components of the app. CSS is a styling language for the web that provides a lot of options and possibilities for customization. The code included is simple:

```
.navbar {
    margin-bottom: 40px;
```

```
.row-title {
    margin-bottom: 20px;
    position: center;
}
img[alt=instructions] {
    width: 100%;
    margin-left: auto;
    margin-right: auto;
}
.row {
    margin-bottom: 70px;
}
.row-title, .frames-title {
    font-size: large;
    text-transform: uppercase;
    color: grey;
    text-decoration: underline;
    text-underline: grey;
}
```

```
}
button, .upload {
    display: flex;
    float: left;
    margin-right: 10px;
}
.dropdown-div {
    display: inline-block;
    width: 200px;
    margin-right: 10px;
}
.dash-dropodown {
    display: flex;
    align-self: flex-start;
}
```

By using classes (`class=`), ids (`id=`) and other HTML attributes to reference the layout elements, the style is configured by different key-value pairs that define the styling properties. In this code the position, size and margins of some elements are modified.

It is important to remark that another default CSS file (`dbc.themes.BOOTSTRAP`) was added as `external_stylesheet` in the app instantiation, this file contains a lot of style specifications that are default to the Bootstrap CSS framework.

3.2.8 Dockerization

Once the application is functioning when executed locally in a Python environment, the final step is to configure it to run in a Docker container. To do so a `Dockerfile` should be defined in the root directory of the project. In addition, a `docker-compose.yml` file is added to automate the Docker commands to build the Docker image and run it in a container. Docker was briefly introduced in the Background section.

The Dockerfile contains the next instructions:

```
FROM python:3.6

WORKDIR /usr/src/app

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8050

CMD [ "python", "./app.py" ]
```

This file basically instructs Docker to build an image based on the `Python:3.6` one, setting a working directory (`/usr/src/app`) in which the app files are copied, and installing the required packages to run the app (`requirements.txt`) and then opening a connection in the port 8050 to access the app running in the container, finally the last instruction executes the application (`app.py`).

The docker-compose.yml is a configuration file for setting up the container:

```
version: "3.8"

services:
  app:
    build: .
    image: vasculomics:dashapp
    ports:
      - 8050:8050
    container_name: dashapp_container
```

This file specifies the name of the image that is going to be built, it maps the host and container ports to make the connection, and it also defines the name of the container that is created.

To build and run the application in docker the user just needs to execute the following command:

```
docker-compose up
```

The docker-compose command has many more options: to pause or stop the execution, to check its logs, to list the images being used by the containers, *etc*.

3.3 Code Practices

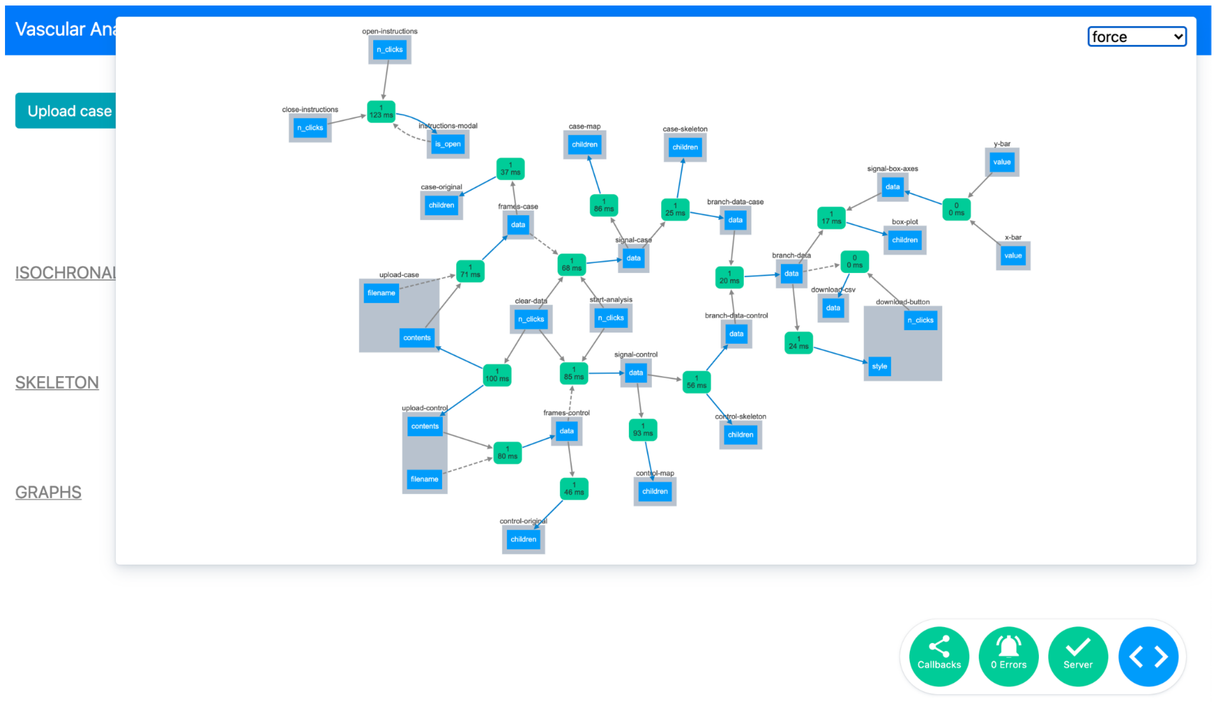
For the development of the application, there are a number of tools and practices that make the process easier and clearer.

The first practice is to search properly the documentation and APIs of the different libraries, software packages and frameworks used to build the application. The recommendation here is to consult the official documentation when possible.

Dash has some intrinsic characteristics that make programming with it a little bit different than programming other Python scripts or programs. For example, the use of global variables is not recommended in Dash, as commented in the Dash Callbacks section. These tips are explained in the official documentation; before putting hands in the project, it is important to understand the basics on how Dash works by following the fundamental sections of the documentation.

For developing the application it is recommended to make use of a Python IDE with debugging options. In general it is recommended to separate the application functionalities in modules and functions that can be reused and make the code look clearer. For the development it is also very recommended to use a version control tool such as Git. The code for this project is hosted in a GitHub repository and Git was used for its development.

When programming the application callbacks and functions, having in mind the execution/flow process makes it easy to handle the possible errors. Dash also provides development tools that make it easy to debug the application; these include a debug mode for testing live code modifications, the callback dependency graph, the error reporting, *etc*. A capture of these Dash tools is shown below.



Dash debugging tools

For configuring the style of the app, it is also useful to use the built in web development tools that are integrated in the web browsers.

One final but nor least important aspect of the application is its modularity. The app was designed with flexibility in mind with the aim of serving as a template for the development of other similar apps for different purposes. Therefore the functions implemented are as loosely coupled as possible and the layout elements are as separated as possible from the rest of the code.

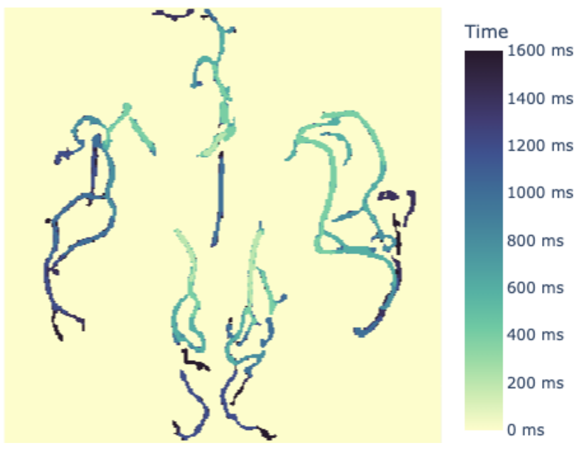
3.4 Evaluation Of The App

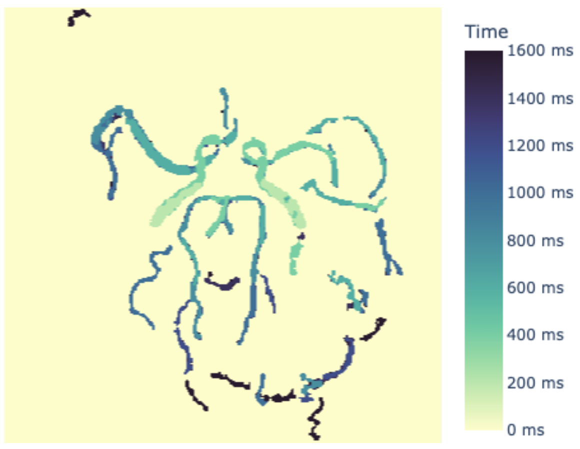
The app functionality is evaluated by testing it for the analysis of some data samples provided by the Hospital Sant Joan de Déu. These samples are included in the DICOM folder of the project root directory.

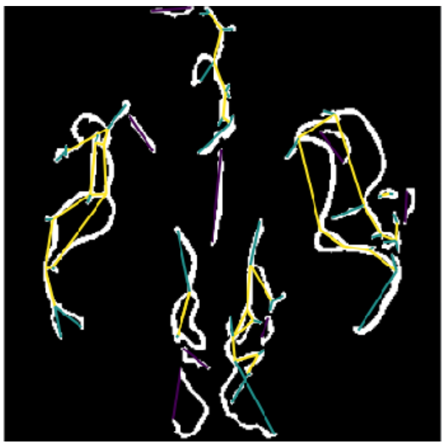
There are a total of 7 sequences to analyze. They are loaded in the app by pairs for analysis in order to make comparisons.

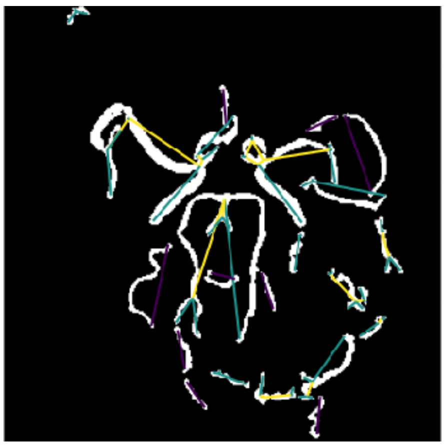
The resulting comparative graphs are included in the following tables:

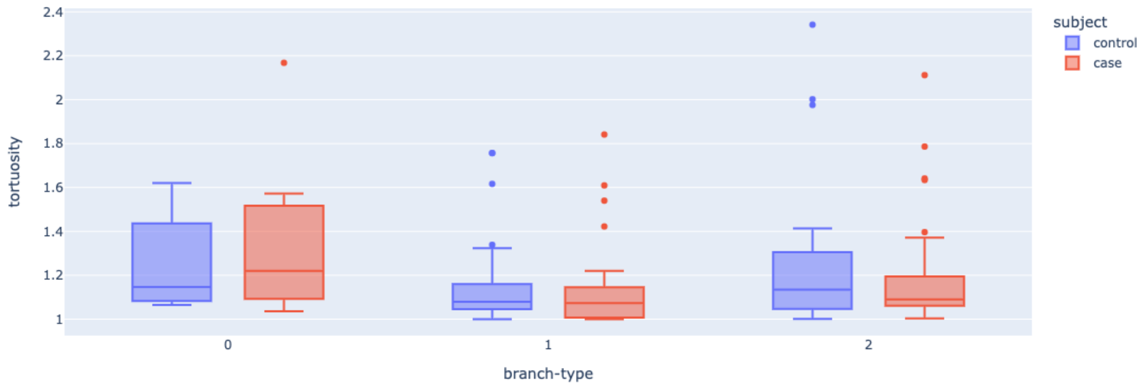
S2040 vs S3040



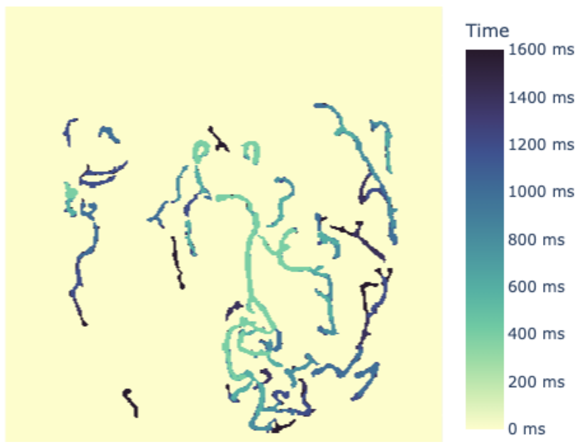


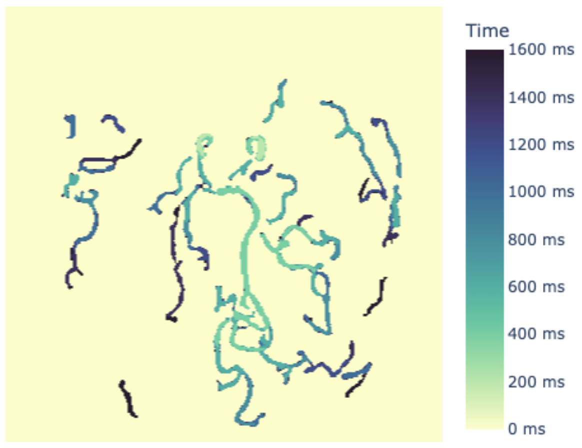


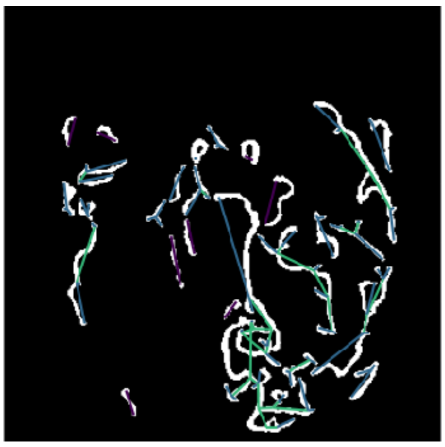


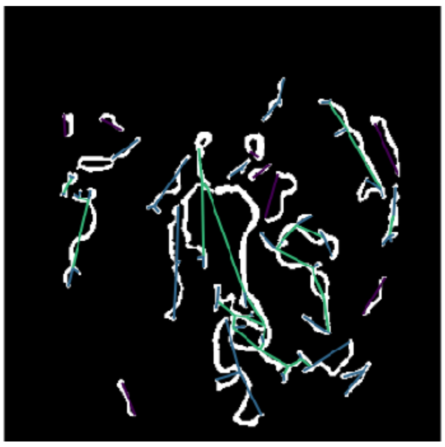


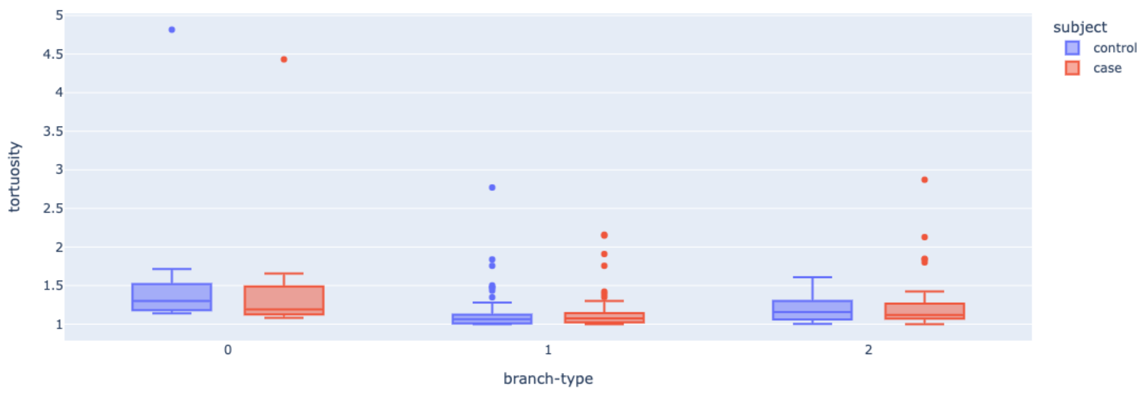
S5040 vs S5040.2

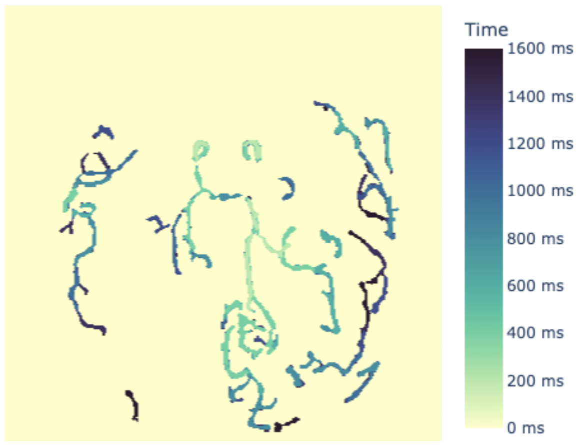
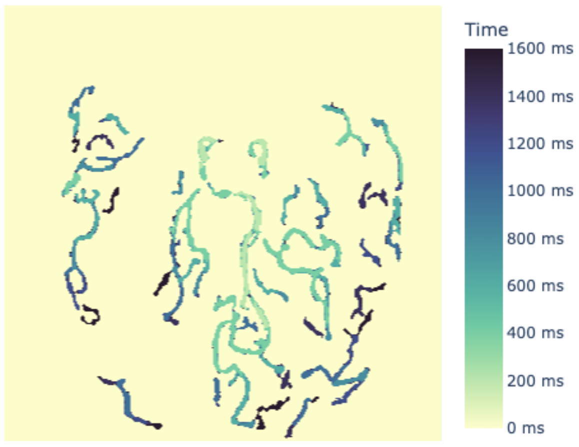
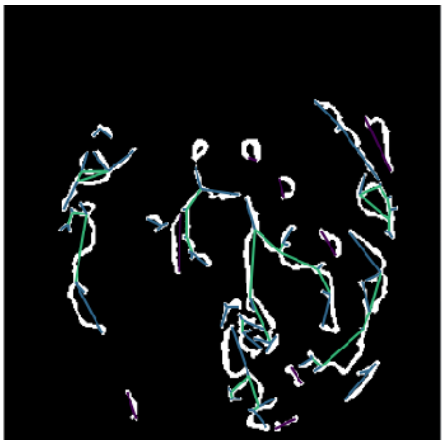
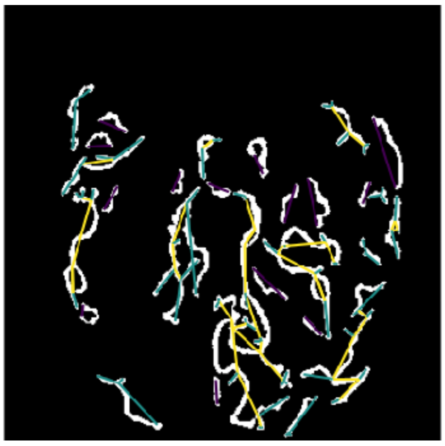


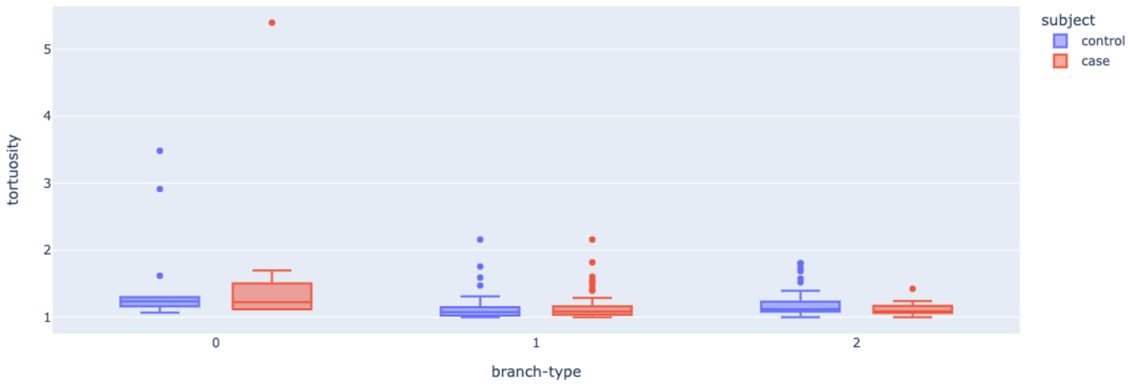


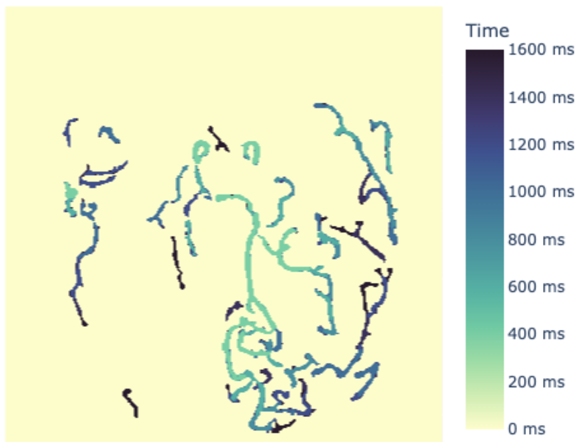
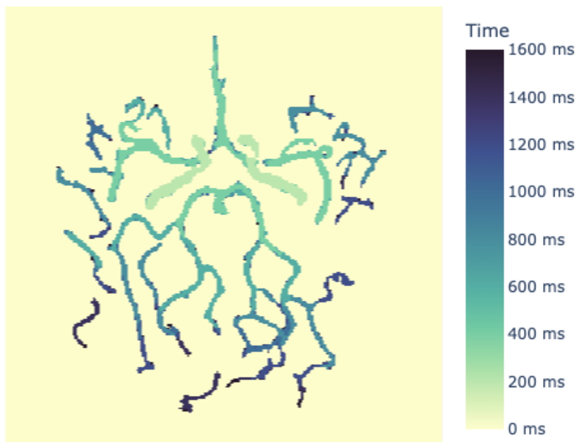
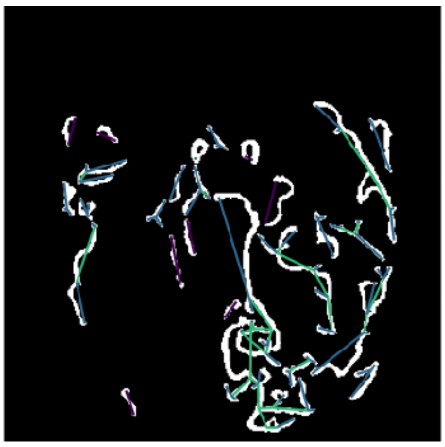
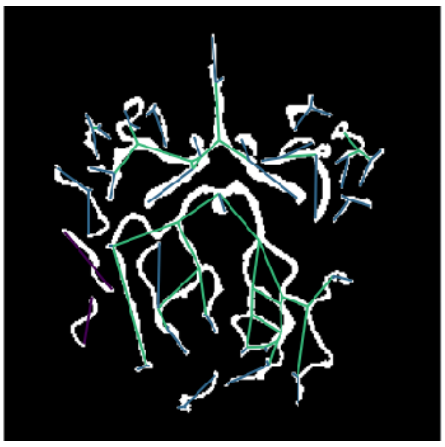


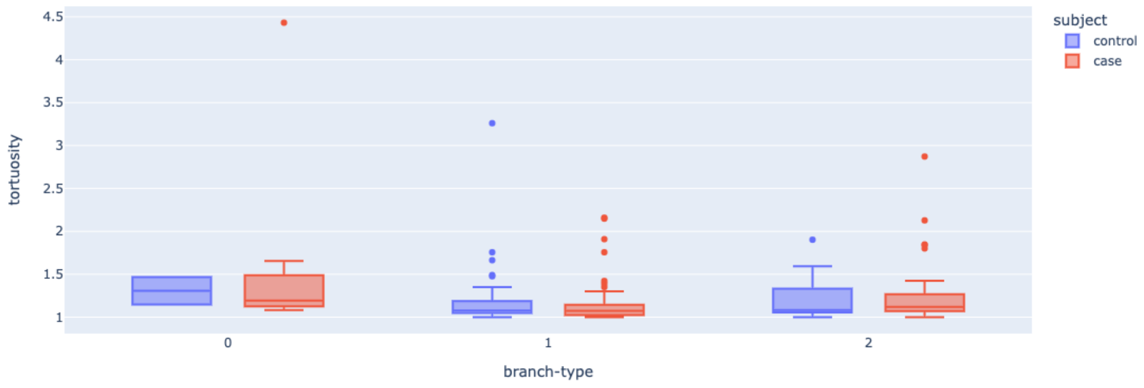




S11040 vs S11040.2	
	
	



S5040 vs S10040	
	
	



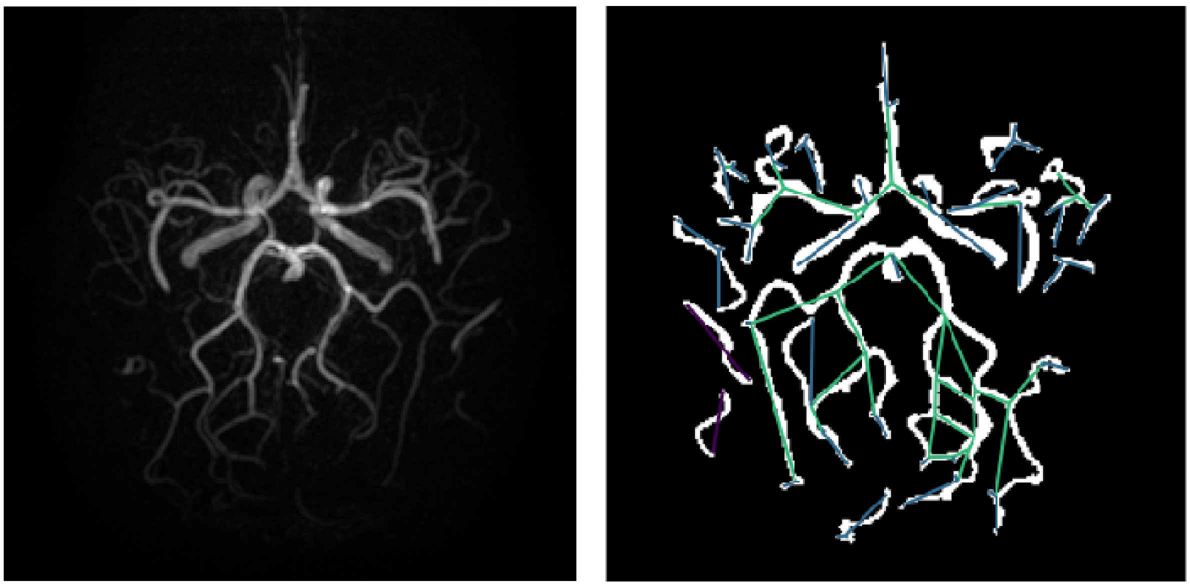
4. RESULTS AND DISCUSSION

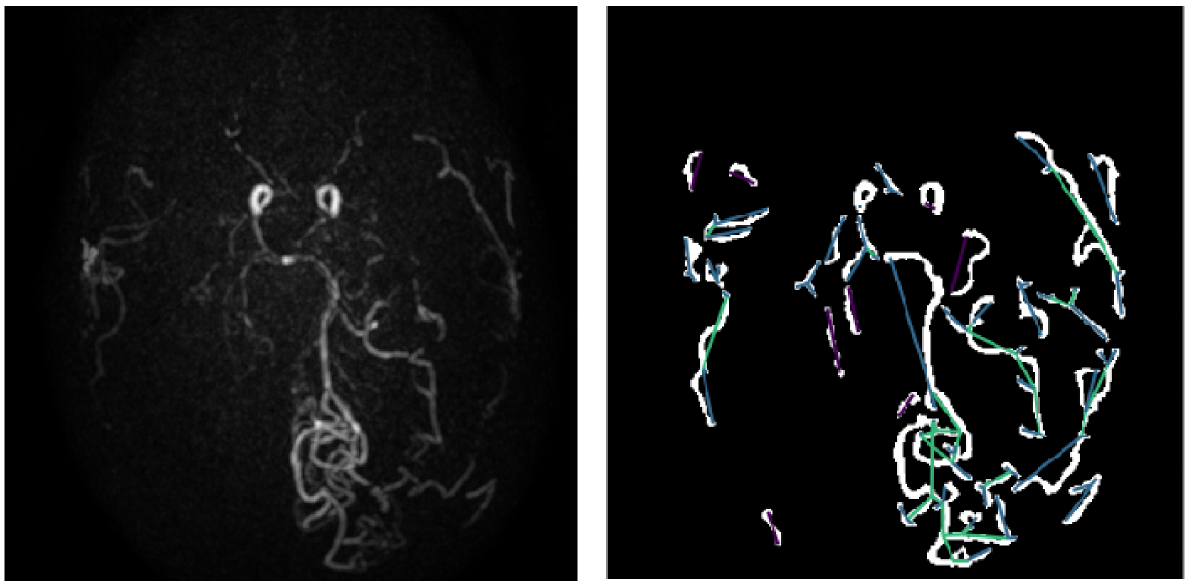
The application can be useful to compare an ill case and a sane control subject, but also for evaluating the efficacy or evolution of a treatment in the same subject, before and after surgery or after the administration of a drug such as a vasoactive drug.

The clinic cases studied in the Evaluation Of the App section reflect that the isochronal maps and skeleton graphs retrieved provide a good global picture for comparing morphological characteristics of the samples, such as symmetry, and to have a basic idea of the blood flow.

The box-plot for comparing the skeleton data between subjects is not as clear as it could be expected. The parameters/biomarkers data extracted for each subject may not be enough to represent and distinguish properly the morphological characteristics of interest. However this depends on the samples under study and of the objective of the observation by the user.

The image processing results were satisfactory, but to make the processing automatic the same filtering parameter values are used for all the samples, so the quality of the processing results depends on the features of the input images that may vary for different acquisitions. As it can be seen in the following cases:


Clear image processing result



Blurry image processing result

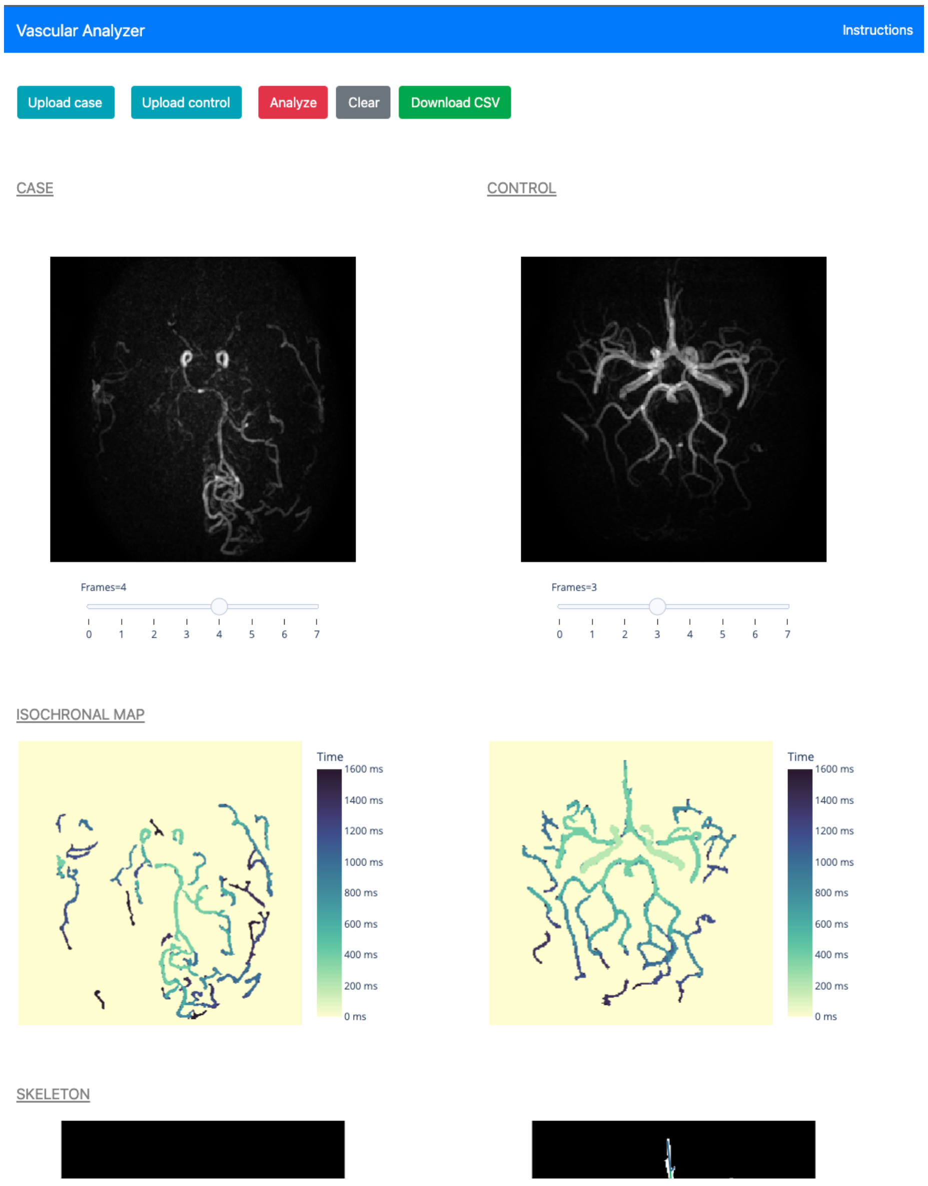
The application works properly, it is responsive and therefore works in a wide variety of screen sizes. The interface is intuitive and the app is easy to use. Maybe a pitfall is that the processing time to load the results may be a little bit high but that depends on the processing power of the computer. Nonetheless, the time loads may not be easily improved and optimized as the image processing algorithms generally require a high number of computations.

This application is a satisfactory prototype but it is not ready for deployment in a Hospital production environment where a lot of IT security compliances should be met and other connections and functionalities would be required for integration and usability.

5. CONCLUSIONS AND FUTURE WORKS

The objective of this project was to develop a web application that would serve medical professionals in the study of medical image data of cerebral vascularity. Such application was successfully developed and tested, and the results obtained were satisfactory.

The application makes use of image processing algorithms, implemented using Python libraries, for the noise filtering and the binarization (segmentation) of the images, as well as for the skeletonization. The results retrieved are an 'isochronal map' reflecting the temporal evolution of the blood flow, and a skeleton graph showing the branches and endpoints of the vascular network. In addition, the skeleton is analyzed and some numerical branch parameters are extracted such as the tortuosity. A graphical box-plot is added representing this numerical data. This analysis is launched by the user in a dashboard web app built based on the Dash Python Framework and running in a Docker container.



Screenshot of the application interface

This application serves as a prototype or template for future developments of related projects. The app developed has some limitations:

- The image processing algorithms (median filter and local threshold) used are simple and not sophisticated. This aspect could be optimized for obtaining better results and to adapt automatically to the quality (i.e. amount of image noise) of the input data.
- The application does not make use of many of the components included in the Dash framework. More controls, graphs results and functionalities could be implemented.
- The application is not integrated with other medical systems and is not ready to function in a production environment.
- The choice for plots to represent the retrieved branch data for comparison between subjects is not that clear. Also, the medical relevance of this data should be assessed by a medical professional.

Despite these limitations, the resulting prototype has proved to be viable and useful. These limitations and other issues could be addressed and the application improved in future works.

The tests of the application were successful, even though the amount of input data available was low, so deeper testing needs to be performed. Not specifically in the context of this project, but it should be remarked that there is a lack of medical image data for training machine learning models. As Kohli, Summers and Geis report: “There is an urgent need to find ways to collect, annotate, discover, and, ideally, reuse adequate amounts of medical imaging data.”

For future works, there are a lot of new possible functionalities that could be implemented:

- Possibility of reading and processing different input file formats.
- Processing of different types of medical images in the same app.
- Manual tuning of some parameters.
- Integration with medical devices and other software.
- Improved modularity and extensibility of the application.
- Write development documentation.
- Optimization of speed and stability (bug fixing).

Future works could also include cloud based and data driven services that could comply with the certifications to manage sensitive clinical data. There are a lot of possible purposes for medical applications such as for administration, diagnosis, research, *etc*.

The technology stack chosen consists of well known open sourced technologies such as Python, Dash or Docker. This is important because it makes the development of the application transparent and based on robust technologies.

Of course, it is important to note that medical knowledge is intrinsically required for the proper functioning of a medical application, and its development should go hand by hand with the counsel and supervision of medical professionals.

Working in this project has served me to learn the basics on how to develop a web application using Python and Dash. What Docker is, how containerization works and how to run the app in a container. Also the basics of front-end web design is covered. Finally, it was useful to study and learn about the image processing algorithms for noise filtering, segmentation and skeletonization of cerebrovascular MRA images.

BIBLIOGRAPHY

[1] Rebollo Ayuso, Carlos. *Dynamic characterization of blood flow through the cerebral artery circle by magnetic resonance imaging*. Jun 2020.

[2] S Shah, Rahul, and Deva S Jeyaretna. *Cerebral vascular anatomy and physiology*. Elsevier, 2018.

[3] American Association of Neurological Surgeons (AANS). "Cerebrovascular Disease." https://www.aans.org/en/Patients/Neurosurgical-Conditions-and-Treatments/Cerebrovascular-Disease. Accessed Sep 2021.

[4] Chamarthy, Murthy R et. al. *Pulmonary arteriovenous malformations: endovascular therapy*. Jun 2018, DOI: 10.21037/cdt.2017.12.08.

[5] National Organization for Rare Disorders. "Vascular Malformations of the Brain." https://rarediseases.org/rare-diseases/vascular-malformations-of-the-brain/. Accessed Sep 2021.

[6] Imperial College London. "Types of Medical Imaging." https://www.doc.ic.ac.uk/~jce317/types-medical-imaging.html. Accessed Sep 2021.

[7] Johns Hopkins Medicine. https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/magnetic-resonance-angiography-mra. Accessed Sep 2021.

[8] Hartung, Michael P., et al. "Magnetic resonance angiography: current status and future directions." *Journal of Cardiovascular Magnetic Resonance*, vol. 13, no. 19, 2011.

[9] Philips. *4D-TRANCE*, https://www.philips.co.uk/healthcare/product/HCNMRB970/4d-trance-mr-clinical-application. Accessed Sep 2021.

[10] National Electrical Manufacturers Association (NEMA). *Digital Imaging and Communication in Medicine (DICOM)*. https://www.dicomstandard.org/current. Accessed Sep 2021.

[11] DICOM. *Key Concepts of the DICOM Standard*, https://www.dicomstandard.org/concepts. Accessed Sep 2021.

[12] World Health Organization Europe. *The protection of personal data in health information systems - principles and processes for public health*. 2021.

[13] Synopsys. *Medical Image Processing*, https://www.synopsys.com/glossary/what-is-medical-image-processing.html. Accessed Sep 2021.

[14] The International Society for Optics and Photonics. *Medical Image Processing*, https://spie.org/publications/deserno-medical-image-processing. Accessed Sep 2021.

[15] Docker. *Docker overview*, https://docs.docker.com/get-started/overview/. Accessed Sep 2021.

[16] O'Connor, Stephen. *What Are the Differences Between PACS, RIS, CIS, and DICOM ?*, 10 May 2017, https://www.adsc.com/blog/what-are-the-differences-between-pacs-ris-cis-and-dicom.

[17] McCormick, M., et al. "ITK: enabling reproducible research and open science." *Front Neuroinform*, 2014, 8:13. doi:10.3389/fninf.2014.00013.

[18] Updegrove, Adam, et al. "SimVascular: An Open Source Pipeline for Cardiovascular Simulation." *Annals of Biomedical Engineering*, no. 45, 2016, pp. 525-541. https://doi.org/10.1007/s10439-016-1762-8.

[19] Nunez Iglesias, Juan, et al. "A new Python library to analyse skeleton images confirms malaria parasite remodelling of the red blood cell membrane skeleton." 2018. doi:10.7717/peerj.4312.

Cipolla, Marilyn J. *The Cerebral Circulation*. San Rafael (CA), Morgan & Claypool Life Sciences, 2009, https://www.ncbi.nlm.nih.gov/books/NBK53086/.

IEEE Engineering in Medicine & Biology Society. *Biomedical Imaging & Image Processing*, https://www.embs.org/about-biomedical-engineering/our-areas-of-research/biomedical-imaging-image-processing/. Accessed Sep 2021.

Kohli, Marc D., et al. "Medical Image Data and Datasets in the Era of Machine Learning." 2017, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5537092/. Accessed Sep 2021.

Miyazaki, Mitsue, and Masaaki Akahane. "Non-Contrast Enhanced MR Angiography: Established Techniques." *Journal of Magnetic Resonance Imaging*, vol. 35, 2012.

Nishimura, Dwight G., et al. *Magnetic Resonance Angiography*. IEEE Transactions on Medical Imaging, VOL. MI-5, NO. 3, Sep 1986.