USER MANUAL

ARTERIAL *An AI framework for mechanical thrombectomy planning through the automated characterization of vascular tortuosity*

Vall d'Hebron Institut de Recerca (VHIR)
Universitat de Barcelona (UB)
July 2021

MSc Pere Canals Canals
PhD Simone Balocco
PhD Oliver Díaz Montesdeoca
PhD, MD Marc Ribó Jacobi

Stroke Research, Neurosciences, Fundació Hospital Universitari Vall d'Hebron - Institut de Recerca (VHIR), Pg. Vall d'Hebron 119-129, 08035 Barcelona, Spain
Department of Mathematics and Computer Science, University of Barcelona, Gran Via 585, 08007 Barcelona, Spain
Department of Neurology, Hospital Vall D'Hebron, 08035 Barcelona, Spain

Contents

1 Introduction

The Arterial©2021, VHIR and UB. All rights reserved, framework is an AI-powered medical image analysis framework for stroke surgical planning through the quantitative assessment of vascular tortuosity, derived from medical imaging prior to intervention in the stroke context. The ultimate goal of the Arterial©2021, VHIR and UB. All rights reserved, is to optimize endovascular treatment for acute ischemic stroke by delivering relevant predictions regarding the difficulty of endovascular treatment in stroke patients prior to intervention, such that neurointerventionalists can make supported decisions regarding the treatment possibilities for each individual patient, and advanced towards a more personalized medicine through the power of AI.

This document is the user manual for Arterial©2021, VHIR and UB. All rights reserved. In this document the methodology behind Arterial©2021, VHIR and UB. All rights reserved, the results for the models that make up the core technology behind the analysis process, as well as all the code within all the framework and the usage of the code are described. This document is intended for internal use and should not be publicly shared nor published without restricted access.

2 Dataset organization

The data that has been used for training and testing was acquired and is property of the VHIR and the Hospital Universitari Vall d'Hebron (HUVH). A dataset of 147 patients was gathered from the internal database of the hospital. The data extracted from the internal database consisted only on the computed tomography angiography (CTA) scans from stroke patients that underwent mechanical thrombectomy (MT), and all data was completely anonymized. For each patient, a series of annotated data was generated from the original CTA scans to train, validate and the AI models involved in the Arterial©2021, VHIR and UB. All rights reserved, framework, including:

- 147 binary maps for the supra-aortic and intracranial arteries observed in the CTA scans, manually segmented by expert engineers and neurologists using 3D Slicer [1].
- 147 labeled graphs corresponding to the centerline models derived from the predicted segmentation database generated after processing of all CTA scans with a trained segmentation AI model. Graph labeling includes detailing the node and edge types for all graphs in the database. Edge types, which are the main target of the labeling model, refer to the individual segment types for all arteries that make up the arterial tree relevant to MT.
- Manual measurements of a series of anatomical features made by two expert observers over 30 CTA volumes, part of the 147 patient database.

All this information is used to train and validate a segmentation network for binary map extraction of the arterial tree in stroke patients, and a graph neural network used for vessel labelling. Both are described in detail in sections 3 and 5, respectively.

3 Segmentation by deep learning: nnU-Net implementation

nnU-Net [2] has been employed as the base framework for the automatic segmentation of the volumes of interest. nnU-Net is a comprehensive framework for the training and deployment of U-Net models for medical image segmentation, with the main particularity that it includes a set of preprocessing operations the goal of which is to extract information that is used to define parameters that affect the network's architecture and the training session. The decisions made according to the data of interest make it possible to adapt a complete model to a given task, which in our case is the segmentation of the arterial tree in head-and-neck CTA volumes.

3.1 Data-derived parameters

nnU-Net has been designed to perform a thorough preprocessing of the training data, not only including spatial and intensity normalization, but also the automated setting of several relevant inferred hyperparameters of the resulting U-net. These include, for example, the 3D patch and batch sizes that are passed to the network for training, the CNN's depth or the oversampling rules followed during training. All these are derived from the shape and size of the training data. These are referred to as the dataset fingerprints, following the convention adopted by the authors of nnU-Net.

After the preprocessing applied over the dataset, a batch size of 2 and a patch size of [112, 112, 192] are chosen for the dataset. Once a patch size is set, the number of downsampling operations is chosen as [4, 4, 5], meaning that the z-axis undergoes an additional downsampling compared to the x- and y-axes (see figure 1).

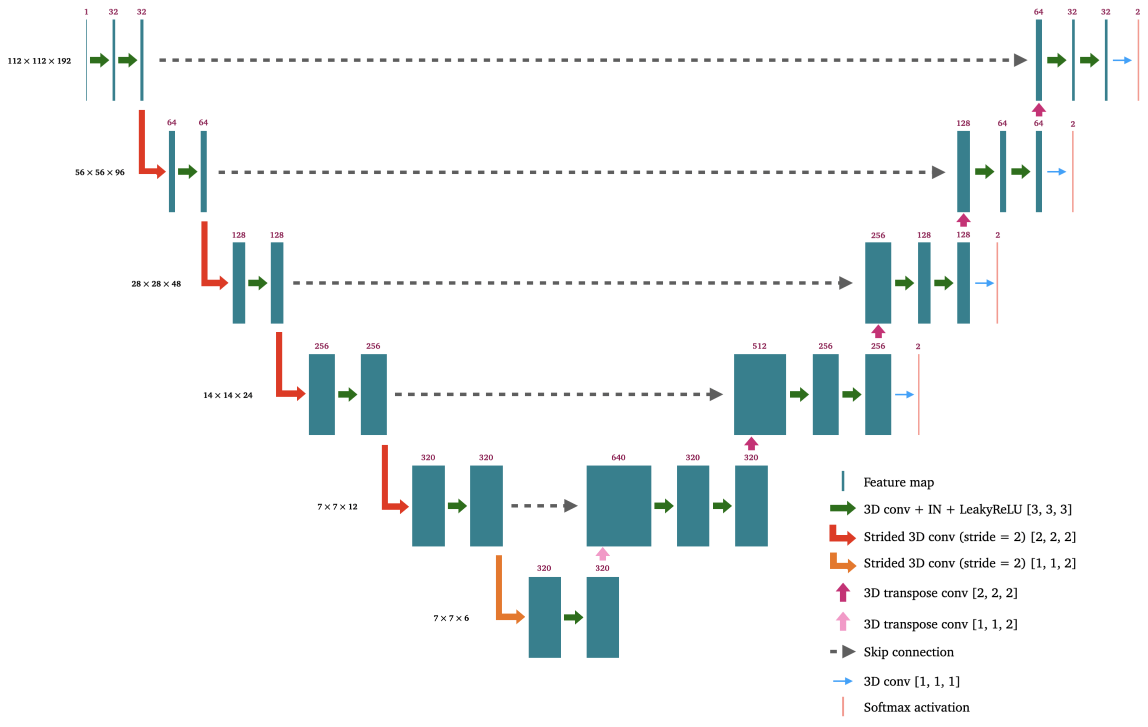
112 × 112 × 192
56 × 56 × 96
28 × 28 × 48
14 × 14 × 24
7 × 7 × 12
7 × 7 × 6
1
32
64
128
256
320
512
640
2
Feature map
3D conv + IN + LeakyReLU [3, 3, 3]
Strided 3D conv (stride = 2) [2, 2, 2]
Strided 3D conv (stride = 2) [1, 1, 2]
3D transpose conv [2, 2, 2]
3D transpose conv [1, 1, 2]
Skip connection
3D conv [1, 1, 1]
Softmax activation

Figure 1: nnU-Net architecture derived from the preprocessing of the dataset used for training. This network is trained to perform binary segmentation of the upper-trunk arterial tree from CTA volumes.

3.2 Blueprint parameters

Apart from the data-derived parameters, the network design includes a wider number of blueprint hyperparameters, designated by the author and unspecific to the task at hand.

The blueprint hyperparameters of the network include several features of the CNN and its training scheme that are potentially very influential to the network's performance and are not decided as a function of the dataset by nnU-Net, but are rather chosen by the authors following their performance with numerous and varied datasets instead.

Regarding architectural features, the authors opted for the use of leaky ReLUs instead of the more traditional ReLUs with a negative slope of 0.01 as the activation function after each convolution.

Instance normalization is preferred over batch normalization, which is beneficial with the dataset used here as it is not sensitive to small batch sizes, something penalized with the usually more robust batch normalization.

Skip connections between encoding and decoding levels are kept the same as in the original U-Net configuration. Deep supervision is implemented to enable losses to penetrate deeper into the network. This is achieved by adding activation layers at each of the decoding levels (except from the two lowest resolution ones) and computing the partial losses from lower resolution version of the labels. These are then added up in a weighted sum, with decreasing weights for lower resolutions (weights are halved at each size reduction, and all weights are normalized to sum to 1).

The kernels in the convolutional layers of the 3D U-Net generated have a shape of [3, 3, 3]. Downsampling is performed with strided convolutions (stride = 2) with kernel size of [2, 2, 2], and upsampling is done with transposed convolution with a kernel size of [2, 2, 2]. A softmax layer is placed after the second convolutional block in all resolution levels considered for deep supervision. Each convolutional block is formed by successive operations of convolution, instance normalization and non-linear activation with leaky ReLU.

The training scheme has several blueprint hyperparameters and functions that can influence optimization and training performance. By default, a full training cycle is defined by nnU-Net as a total of 1000 epochs, with each epoch being conformed by iteration over 250 mini-batches, plus additional 50 mini-batches for validation. In this study, these will be referred to as (training and validation) mini-batch sizes. This means that the trainer will process a total of 500 training patches in batches of 2 within each epoch.

Stochastic gradient descent (SGD) with Nesterov momentum ($\mu = 0.99$) is used as the default optimizer for the network, and the loss function is computed as a combination of the cross entropy (CE) and the Dice loss.

The learning rate schedule was modified to PyTorch's ReduceLROnPlateau[1] routine following an optimization study performed with a reduced dataset.

3.3 Results

5-fold cross validated trainings were performed with the full dataset, achieving a Dice score of 0.93 ± 0.02 and a recall of 0.93 ± 0.03 on the testing set (table 1).

Model	**Dice coefficient**	**Recall**
Fold 0	0.93 (0.02)	0.93 (0.03)
Fold 1	0.93 (0.02)	0.93 (0.03)
Fold 2	0.93 (0.02)	0.93 (0.03)
Fold 3	0.93 (0.02)	0 93 (0.03)
Fold 4	0.93 (0.02)	0.94 (0.03)
Average	0.93 (0.02)	0.93 (0.03)

Table 1: Resulting Dice coefficient and recall across all folds for the 5-fold cross-validation for the segmentation nnU-Net across all the testing set. Standard deviations for all averaged values are displayed within parentheses.

Results from the model ensembling using the 5 different folds for the cross-validation study are displayed in table 2. Quantitatively, the segmentations from ensembling do not improve from the single model inference.

[1]ReduceLROnPlateau parameters: factor = 0.2, patience = 5, threshold = 0.05, mode = "min", threshold_mode = "rel"

Model	Dice coefficient	Recall
Ensemble	0.93 (0.02)	0.93 (0.03)

Table 2: Dice coefficient and recall resulting from the ensembling of the 5 folds from the cross-validation study. Standard deviations for all averaged values are displayed within parentheses.

4 Centerline extraction and branch clipping

Automatic centerline extraction has been developed using the vascular modelling toolkit (VMTK) [3] using Slicer Python scripting. The SlicerJupyter extension was very useful in the development process. A post processing was applied to the binary volumes after the segmentation model. First, the voxels from the upper 20% of the foreground's bounding box are set to 0. This is done to preserve robustness in the centerline extraction process and preserve a lower number of vascular segment times (excluding cerebral arteries), to ease the training process and boos performance of the following processes. The rest of the post-processing consists on the application of a Gaussian smoothing filter with a standard deviation of 0.5 mm and the selection of the largest islands in terms of number of voxels (all islands smaller than 10,000 voxels are removed). Then all centerline models are saved (separately). Figure 2 shows the comparison between the raw segmentation resulting from nnU-Net and that upon post-processing completion.

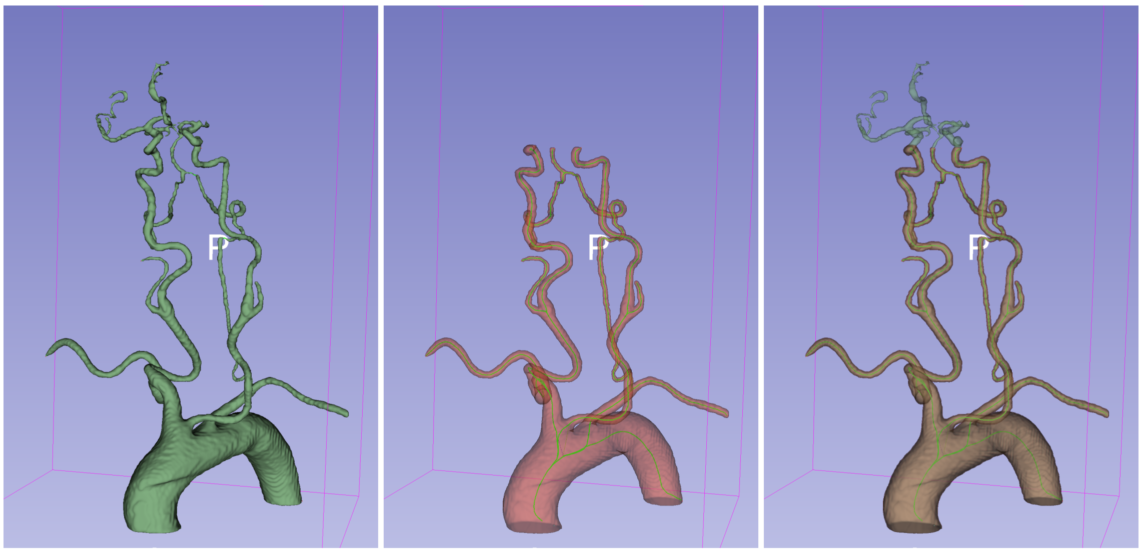

Figure 2: (Left) Binary volume output by the segmentation nnU-Net presented in figure 1, segmented by thresholding using the Segment Editor module in Slicer, via Python Scripting. (Center) Extruded binary volume with the voxels of the upper 20% of the bounding box removed, and the centerline model extracted as a set of green lines. (Right) Superposition of both models for comparison purposes.

After centerline extraction, VMTK modules *vmtkbranchextractor* and *vmtkbranchclipper* are used to convert the centerline and surface models to branched and clipped versions respectively, to split these models into individual segments. All branch and clipped models from the multiple islands derived from each case are then unified into one model, and the resulting branch model is used to generate a graph, where nodes correspond to bifurcations and edges to centerlines of individual vascular segments.

5 Vessel labelling with graph nets

After centerline extraction and branch clipping of the surface 3D model, the focus is shifted to labelling individual vascular segments on the surface and centerline models. In order to do that, a graph neural network (GNN) model proposed by Chen et al. (2020) [4] was adapted, and a new dataset with labeled graphs derived from the centerline models was generated. The GNN model consists of a classic encoder decoder architecture, with an encoder block designed to input the number of features for both nodes and edges, a core layer that processes the input graph 10 times, and a decoder layer that outputs the activation for each of the possible node and edge categories. Node features include position of the bifurcation point radius corresponding to that centerline point, directional embedding for each of the departing edges (one-hot encoded for each of the 26 major directions, each 45 degrees apart in all three spatial directions) and degree of the node (equal to the number of edges connected by the node), for a total of 31 features. Edge features include the normalized direction of the edge (from the start- to the endpoint of centerline segment), the distance between nodes (euclidean distance between start- and endpoint of the centerline segment) and mean radius of the edges for a total of 5 features. Data augmentation is applied only in the form of a random translation of the nodes to a nearby region (within a 10% of 0.1 mm). No global graph features were used.

Regarding the training process, the Adam optimizer was used with a constant learning rate of 10^{-3}, and a combined weighted cross entropy loss for both nodes and edges was used, using the inverse of the frequency of each of the node and edge types within the training set as weights. The training set was the same used for the training of the segmentation nnU-Net, with a total of 102 graphs for training and validation, and the same 45 cases as for the segmentation network were reserved for testing. A 5-fold cross-validation study was performed, and testing results in the form of classification accuracy are displayed in table 3.

Model	**Prediction accuracy**	**Node accuracy**	**Edge accuracy**
Fold 0	0.82 (0.11)	0.86 (0.08)	0.76 (0.18)
Fold 1	0.84 (0.08)	0.86 (0.08)	0.81 (0.13)
Fold 2	0.75 (0.11)	0.77 (0.11)	0.72 (0.16)
Fold 3	0.83 (0.09)	0 87 (0.08)	0.79 (0.14)
Fold 4	0.85 (0.08)	0.89 (0.07)	0.80 (0.12)
Average	0.82 (0.10)	0.85 (0.09)	0.78 (0.15)

Table 3: Resulting accuracy from each of the folds and overall results for the GNN model regarding overall predictions (nodes and edges together), nodes and edges. Standard deviations for all averaged values are displayed within parentheses.

6 Code description

In this section, we review the current repository (currently private in GitHub: `https://github.com/perecanals/arterial`), describing its organization and providing details of every independent code that is used sequentially during processing of the data, as well as the files that are returned and/or written for each code.

Cloning the GitHub repository as a local copy will create the base directory named `arterial`. Within this directory, one can find the `README.md`, `LICENSE` (to be ignored at the moment, will be changed once a suitable license is chosen), `.gitignore` and the `setup.cfg` and `setup.py` files (last two to be ignored at the moment), as well as an empty `tests` directory and the main `arterial` directory. To describe the organization of the different modules, we will be taking the last `arterial` directory is used as reference (`"arterial/arterial"` when cloning the repository). Inside the `arterial` directory, the `performAnalysis.py` script can be found, a wrapper for all successive functions that compose the complete analysis.

We separate the repository in four different directories, each in charge of a different task. The four tasks are, in order of sequence:

- `segmentation` Performs segmentation from original CTA NIfTI, and creates a new NIfTI file containing the binary map. This module contains the trained nnU-Net models and can perform the segmentation via regular inference with one model or via ensembling with five different models. Code for this module is found in `arterial/segmentation`, and it is composed of a single Python script (`nnunetSegmentation.py`), as well as two directories: `models` (containing all the trained nnU-Net models) and `nnUNet_base` (containing necessary data for the nnU-Net commands to set up the models).
- `centerlineExtraction` Extracts centerline-derived models. The raw centerline is extracted with Slicer's VMTK module. Once the centerline models are generated, VMTK's `vmtkbranchextractor` and `vmtkbranchclipper` are used to create the branch model (a branched version of the centerline model) and the clipped model (a clipped version of the segmentation surface model, with branch information linked to the branch model), respectively, for each separate segment found, and these are unified into one single model for the case analyzed (one branch model and one clipped model). Code for this module is found in `arterial/centerlineExtraction` and it is organized into four different Python scripts: `performSegmentationsFromBinaryMask.py`, `centerlineExtraction.py`, `performSeg-mentationAndCenterlineExtraction.py` and `branchAndClippedModelUnification.py`.
- `vesselLabelling` The centerline models are used to generate a graph that represents the unified centerline model, storing node and edge features as described in [4]. This graph is then processed by the trained GNN and a new graph is output, with predicted node and edge types. An auxiliary array (segmentsArray) is generated when building the graph, storing information of the centerline points, separating each segment between bifurcations, and attributing each centerline segment (associated to a graph edge) with an identifier. This identifier is used to trace back the predicted graph information and relate it to the branch model information. The main output of this section is the predicted graph and a dictionary relating all branch model group identifiers with the segmentsArray identifiers. Code for this module is organized into three different Python scripts (`centerlineGraph.py`, `segmentsArray.py` and `graphBranchModelLink.py`) and a directory (`graphNet`) containing an additional script (`gnnInference.py`) as well as the trained GNN model in the `model` directory.
- `featureExtraction` Collects all generated information from the branch and clipped models, as well as the predicted graph and computes tortuosity features. This section is solely formed by an object class that performs a series of computations derived form the following data: segmentsArray, predicted graph, branch model, clipped model and branch model group identifier to segmentsArray identifier dictionary. At the moment, this is the list of features that can be automatically extracted from a case:
  - Presence of bovine aortic arch.
  - Presence of aberrant RSA (ARSA).
  - Proximal diameter[2].
  - Relative length[3].
  - Absolute departure angles (polar and azimuth)[2].
  - Relative departure angle (polar and azimuth)[2].

  This module is formed by a single Python script: `automaticFeatureExtraction.py`.

[2]For AA (diameter at apex), BT, RCCA, RSA, RVA, LCCA, LSA and LVA.
[3]For BT, RCCA, RSA, RVA, LCCA, LSA and LVA.

6.1 Input data

In order to process a case, we only need to provide a NIfTI file (.nii.gz) of the CTA volume, without any additional information. For consistence, it is preferred to use the case identifier (caseId) for the original NIfTI as well as for its parent directory. Therefore, the only input that needs to be provided as an argument for the `performAnalysis.py` script is `casePath`, *i.e.* the path to the original NIfTI:

```
casePath = "/path/to/caseId/caseId.nii.gz"
```

The `caseDir` derived from `casePath` is then:

```
caseDir = "/path/to/caseId"
```

`caseDir` acts as the main directory for each case, where intermediate results will be stored and organized.

6.2 Segmentation

Regarding segmentation, a total of 6 different models are provided in the repository, including 5 different folds derived from a single training run, as well as a separate model, which is selected as the best performer within the 5 folds in terms of test metrics. There is only a script in this module, `nnunetSegmentation.py`

6.2.1 nnunetSegmentation.py

`nnunetSegmentation.py` acts as a wrapper for the nnU-Net built-in commands (corresponding to the pip-installable `nnunet` package). Two modes can be input as arguments (`inference` or `ensemble`) for this script, that determine how the image is processed for segmentation.

- Description: wrapper for nnU-Net built-in commands that performs segmentation with trained models.
- Input data: original CTA (`casePath`).
- Output: generates a new NIfTI file with path `"caseDir/caseId.nii.gz"` . Changes original CTA to `"caseDir/caseId_CTA.nii.gz"` .
- Arguments: `-mode` mode (options: inference, ensemble); `-casePath` casePath.
- Returns: -

6.3 Centerline extraction

The main goal of this section is to generate the centerline models and surface models that provide meaningful information such as the branching into single segments, the position for all centerline points and other useful information. The only input data of this section is the predicted segmentation made by the trained nnU-Net, and expected main outcomes are the centerline models, the branch model and the clipped surface model of the head-and-neck-arterial tree imaged in the original CTA acquisition.

6.3.1 performSegmentationsFromBinaryMask.py

This script inputs the predicted segmentation as a NIfTI file through Slicer, and utilizes the SegmentEditor tools to generate the segmentation node inside the Slicer context used for centerline analysis. The post-processing of the segmentation resulting from the nnU-Net forward pass consists of four different steps: first, the Thresholding tool is used to trivially segment the foreground voxels from the binary map (minimum and maximum thresholds are set to 1). After that first step, the a Gaussian smoothing filter with a standard deviation of 0.5 mm is applied. 0.5 mm was used following examination of several filter sizes. It was found that 0.5 mm is able to improve smoothness of the surface model without compromising the integrity of smaller vessels. After the smoothing filter, the Islands tool is used to eliminate all segmented bodies smaller than 10,000 voxels (chosen by observation if a large number of cases) and to separate all islands left into different segments. The resulting segmentation node is output for centerline extraction.

Ignoring cerebral arteries. At this stage, we are disregarding the foreground voxels inside the upper 20% of the bounding box of the segmentation. In practice, this means that we are not including the cerebral arteries in this first phase of the project. Several reasons are behind this decision: the segmentation network's performance with the smaller cerebral arteries is not up to the standards that we are seeking with the current training dataset and centerline extraction is made considerably more difficult when including cerebral arteries. Moreover, the size of the vessels in this region are very small in comparison to the voxel size of the dataset, which makes it so that radius measurements for arteries smaller than 2 mm are not reliable (vessel diameter should at least be 5× the size of the voxel size, which is around 0.4 mm in the three directions). Upsampling would be needed to tackle this issue, which can significantly increase the computational cost of the complete analysis. This issue is to be dealt with in future versions.

This script has to be called through Slicer's Python interpreter for the Slicer functions and context to be available. Path to Slicer's interpreter in defined in the `performAnalysis.py` main script. This should be adapted to each system (en environment variable could be defined when installing the software).

- Description: obtains processed Slicer segmentation node from binary NIfTI.
- Input data: NIfTI file (`casePath`) with predicted segmentation.
- Output data: -
- Arguments: `masterVolumeNode` Slicer volume node with loaded binary NIfTI.
- Returns: `segmentationNode` Slicer segmentation node with resulting segmentation.

6.3.2 centerlineExtraction.py

Uses the VMTK Slicer extension to extract centerline models from all segments in the resulting segmentation node returned by `performSegmentationsFromBinaryMask.py` . For each of the segments in the segmentation node, first a decimated version of the surface model (down to 70% of the original number of triangles) is generated and stored in `"caseDir/decimatedSegmentations/dec-imatedSegmentations{idx}.vtk"` , with *idx* being the numeration from 0 (included) to the number of segments (excluded). Then, the Extract Centerline module from the VMTK Slicer extension is used to automatically detect the start- and endpoints for the centerline module, which are then relocated by a custom algorithm to add robustness to this feature. Without this relocalization, errors are often found when drawing centerlines as VMTK's automatic endpoint detection methods sometimes places these points in the border or outside the surface model, which causes the centerline tracing to fail constantly. Our custom algorithm identifies the closest vessel component within a local region ($10 \times 10 \times 10$ voxels around the endpoint) and places the endpoint at the center of mass of the closest group inside this region. This practically reduces to 0 the frequency to which errors are found as compared to the original method.

Once endpoints are relocated, VMTK's extract centerline method is applied and floating centerlines (an infrequent error that causes some centerlines not to originate from the designated startpoint due to the morphology of the segmentation) are removed. Each centerline model is saved separately as `"caseDir/centerlines/centerlines{idx}.vtk"`.

Again, this script has to be called through Slicer's Python interpreter for the Slicer functions and context to be available.

- Description: extracts centerline models and decimated segmentation surface models from each of the segments in the segmentation node.
- Input data: -
- Output data: decimated segmentation (`"caseDir/decimatedSegmentations/decimatedSegmentations{idx}.vtk"`) and centerline model (`"caseDir/centerlines/centerlines{idx}.vtk"`) for each of the segments in `segmentationNode`.
- Arguments: `segmentationNode` Slicer degmnetation node with the post-processed segmentation from the binary NIfTY.
- Returns: -

6.3.3 performSegmentationAndExtractCenterline.py

This script acts as a wrapper for the `performSegmentationsFromBinaryMask.py` and `centerlineExtraction.py` scripts. This is the script called from the `performAnalysis.py` main script and its function is to load the predicted segmentation as a volume node in the Slicer application and call both scripts described above.

- Description: wrapper for `performSegmentationsFromBinaryMask.py` and `centerlineExtraction.py` scripts.
- Input data: NIfTI file (`casePath`) with predicted segmentation.
- Output data: decimated segmentation (`"caseDir/decimatedSegmentations/decimatedSegmentations{idx}.vtk"`) and centerline model (`"caseDir/centerlines/centerlines{idx}.vtk"`) for each of the segments in `segmentationNode`.
- Arguments: binary NIfTI of the predicted segmentation, output by the (`-casePath`).
- Returns: -

6.3.4 vmtkbranchextractor and vmtkbranchclipper

After generating the centerline and decimated surface models, the `vmtkbranchextractor` and `vmtkbranchclipper` commands included in the VMTK package are used to generate the branch models and clipped models, respectively. First these are generated independently (the `"caseDir/centerlines/centerlines{idx}.vtk"` model is used to generate the `"caseDir/branchModels/branchModel{idx}.vtk"`, which is then used as input together with the `"caseDir/decimatedSegmentations/decimatedSegmentation{idx}.vtk"` surface model to generate the `"caseDir/clippedModels/clippedModel{idx}.vtk"`).

- Description: VMTK commands used in a section of `performAnalysis.py` to obtain the branch models and clipped surface models.
- Input data: decimated segmentation (`"caseDir/decimatedSegmentations/decimatedSegmentations{idx}.vtk"`) and centerline model (`"caseDir/centerlines/centerlines{idx}.vtk"`) for each of the segments in `segmentationNode`.
- Output data: branch model `"caseDir/branchModels/branchModel{idx}.vtk"` and clipped model `"caseDir/clippedModels/clippedModel{idx}.vtk"` for each of the segments in `segmentationNode`.
- Arguments: -
- Returns: -

6.3.5 branchAndClippedModelUnification.py

Detection of the centerline model, as well as the branch and clipped models with several islands is not reliable, as our experience has shown. Therefore we have opted for a strategy that involves analyzing each of the generated large segments individually. This script joins all branch models and clipped models into one single file for each model type, generating one branch model (`"caseDir/branchModel.vtk"`) and one clipped model (`"caseDir/clippedModel.vtk"`).

- Description: branch models and clipped models are unified into one single model for each type.
- Input data: branch model `"caseDir/branchModels/branchModel{idx}.vtk"` and clipped model `"caseDir/clippedModels/clippedModel{idx}.vtk"` for each of the segments in `segmentationNode`.
- Output data: unified branch model `"caseDir/branchModel.vtk"` and unified clipped model `"caseDir/clippedModel.vtk"`.
- Arguments: -
- Returns: -

6.4 Vessel labeling

The purpose of this section is to attribute all individual branched segments with meaningful labels to automatically recognize the vessel type for each branch of the centerline and surface models. To do that, the centerline models, the branch model and the clipped surface model are used. A graph is generated following these models, which is processed by a trained graph net model that returns a labeled version of the graph. This information is used to label all branches recognized in the branch and clipped models.

This module also includes a separate directory (`"arterial/vesselLabeling/graphNet"`) that stores the trained GNN model (in `"arterial/vesselLabeling/graphNet/model"`).

6.4.1 segmentsArray.py

This scripts reads the centerline models in `"caseDir/centerlines"` and creates a `.npy` file with all centerline points grouped in sections between bifurcations as well as the radius data associated with those points. This array is used to generate the graphs that are to be labeled by the graph net *a posteriori*.

The point positions are passed from the RAS coordinate system to the IJK coordinates by means of the affine matrix associated with the original NIfTI file. This is done for normalization purposes regarding the graph. In this script, the potentially multiple centerline models are successively analyzed. These models include as many cells as endpoints, and each cell stores all centerline points between the startpoint and each endpoint. Virtually in all cases, all cells overlap with each other along a certain duration, so the centerline models do not contain information of the bifurcation points directly. To generate the graph, we need this information so as to be able to draw all nodes and edges in the corresponding positions, so a small processing is needed to obtain the `segmentsArray.npy` file, containing centerline information (position and radius) of all non-overlapping segments between bifurcations.

- Description: analyzes all centerline models and generates an array grouping all centerline coordinates and radius information between individual segments (between two bifurcation-s/endpoints).
- Input data: centerline models in `"caseDir/centerlines"` .
- Output data: segments array as `"caseDir/segmentsArray.npy"` .
- Arguments: -
- Returns: segmentsArray.

6.4.2 centerlineGraph.py

Here, the segments array is loaded to generate a graph using the `networkx` package. Bifurcation point positions correspond to graph nodes, and the information of the centerline points, along with the radius values, are used to generate node and edge features that are input to the graph net.

The segment order in the segments array defines the cellID for the centerline edges, which is an identifier that is used later on for feature extraction.

- Description: reads the segments array and builds a graph using the point positions of the centerline model's bifurcations as well as other data to define the node and edge features.
- Input data: segments array as `"caseDir/segmentsArray.npy"` .
- Output data: graph as `"caseDir/graph.pickle"` and graph image as `"caseDir/graph.png"` .
- Arguments: segmentsArray.
- Returns: -

6.4.3 graphNet/gnnInference.py

This script reads the graph output by `centerlineGraph.py` and sets up a Tensorflow session, loading the model stored in `"arterial/vesselLabeling/graphNet/model"`. After processing the model through the graph net, the predicted graph is saved.

- Description: performs a forward pass with the graph generated in `centerlineGraph.py` through the trained graph net to obtain the final graph, with predictions node- and edgetypes.
- Input data: graph as `"caseDir/graph.pickle"` and model files in `"arterial/vesselLabeling/graphNet/model"`.
- Output data: graph with predicted node- and edgetypes as `"caseDir/graph_pred.pickle"`.
- Arguments: -
- Returns: -

6.4.4 graphBranchModelLink.py

The objective for this script is to associate the labels of the predicted centerline graph to the group identifiers (groupId) of the branch and clipped models, computed by VMTK's commands. The groupId can be a useful variable for feature extraction, so it is in our interest to trace back the edgetypes predicted by the graph net to the groupId for each branch of the vascular tree.

This code inputs the branch model, the segments array and the predicted graph, and performs a series of operations to find the relationships between segments in the segments array (complete centerline segments between bifurcations) and the branch model cells corresponding to the groupIds (these identifiers do not usually correspond to complete segments between bifurcations, but are rather contained in the central part between bifurcations, when the vessel diameter becomes stabilized, and disregard transitions between vessels).

The pairing process consists on localizing all branches with blanking equal to 0 (blanking indicates branch model cells that correspond to surface model segments. Transitions between different vascular segments have blanking equal to 1) and searching for the branch model cell inside the segments array segments. Each branch model cell will at least be contained inside a segments array segment.

It is possible that for a given groupId, when a bifurcation falls inside of a segment defined by a groupId, several branch model cells with the same groupId correspond to different edges of the predicted graph, with potentially different associated edgetypes. Several rules are defined to determine the prioritization of the preferred edgetype-groupId correspondence. In these cases, there are generally two different branch model cells involved, both with blanking equal to 0. If the parent vessel exists (both cells have multiple centerline points in common), then the edgetype of the parent vessel is chosen for the groupId. Otherwise, the longest child branch model cell is chosen for the groupId-edgetype correspondence. Other criteria could be more accurate here (e.g., radius assessment for the branch model cells compared to the registered mean radius for the predicted graph edges).

- Description: links all edgetypes corresponding to segments between bifurcations (from the segments array) to all branch model cells' groupIds.
- Input data: segments array as `"caseDir/segmentsArray.npy"`, predicted graph as `"caseDir/graph_pred.pickle"` and branch model as `"caseDir/branchModel.vtk"`.

- Output data: dictionary for the groupId to vessel types (edgetypes) as `"caseDir/groupIdsToVesselTypesDict.json"`.
- Arguments: -
- Returns: -

6.5 Feature extraction

Once linking of the predicted edgetypes by the graph net to the data of the branch and clipped models has been done, all that is left is to apply operations to extract tortuosity features from each case. To recapitulate, this is the data that we should have from a single case up to this point of the analysis (excluding all intermediate centerlines, decimated segmentation, branch models, clipped models, as well as irrelevant images):

- Original NIfTI CTA volume as `"caseDir/caseId_CTA.nii.gz"`.
- Predicted segmentation made by nnU-Net as `"caseDir/caseId.nii.gz"`.
- Branched centerline model as `"caseDir/branchModel.vtk"`.
- Clipped surface model as `"caseDir/clippedModel.vtk"`.
- Segments array as `"caseDir/segmentsArray.npy"`.
- Raw graph derived from segments array as `"caseDir/graph.pickle"`.
- Predicted graph after graph net processing as `"caseDir/graph_pred.pickle"`.
- Dictionary with vessel type to groupId links as `"caseDir/groupIdsToVesselTypesDict.json"`.

6.5.1 automaticFeatureExtraction.py

To perform the features extraction process, the `featureExtractor` class is built is this script. When an object of this class is instanced with a case directory, all necessary data is loaded as object attributes. These attributes include:

- `self.caseDir` *str* – System path to case directory.
- `self.aff` *numpy array* – 4x4 affine transformation matrix associated to the original CTA volume (IJK to RAS).
- `self.segmentsArray` *numpy array* – Loads `"caseDir/segmentsArray.npy"`.
- `self.segmentsArrayAff` *numpy array* – Loads positions of segments from segments array transformed to RAS coordinates with `self.aff`.
- `self.graph` *networkx graph* – Loads predicted graph from `"caseDir/graph_pred.pickle"`.
- `self.cellIdToVesselType` *dict* – Keeps relation between cellIds (position of segment in `self.segmentsArray`) and predicted edgetypes.

- `self.branchModel` *vtkPolyData* – Loads branch model from `"caseDir/branchModel.vtk"` as a vtkPolyData object.
- `self.branchModelCoordinates` *numpy array* – Stores all branch model centerline point coordinates in a flat array.
- `self.blanking` *numpy array* – Stores blanking from branch model cells for each point of `self.branchModelCoordinates`.
- `self.groupIdBranchModel` *numpy array* – Stores groupId from branch model cells for each point of `self.branchModelCoordinates`.
- `self.radius` *numpy array* – Stores radius for each point of `self.branchModelCoordinates`.
- `self.clippedModel` *vtkPolyData* – Loads clipped model from `"caseDir/clippedModel.vtk"` as a vtkPolyData object.
- `self.clippedModelCoordinates` *numpy array* – Stores all clipped model mesh point coordinates in a flat array.
- `self.groupIdClippedModel` *numpy array* – Stores groupId from clipped model for each point of `self.clippedModelCoordinates`.
- `self.groupIdsToVesselTypesDict` *dict* – Loads dictionary with groupId to vessel type (i.e., edgetype) link from `"caseDir/groupIdsToVesselType.json"`.
- `self.featureExtractorDict` *dict* – Initializes empty dict to store all automatic measurements.
- `self.featureExtractorExtendedDict` *dict* – Initializes empty dict to store all automatic measurements as well as additional data.

After all these attributes are initialized, the featureExtractor object should be ready to perform the automatic feature extraction and generate the output file with all measurements. Separate methods are defined for each of the different measurements that are performed. We now visit in detail each of these methods.

self.extractFeatures()

This method acts as a wrapper or all other feature extraction methods. Here, respective methods are called to find presence of bovine arch and ARSA, find aortic arch type, and perform measurements of proximal diameter, relative length, absolute departure angles and relative departure angles. This method writes all these measurements into a dictionary which is saved as a json file at `"caseDir/features.json"`.

- Description: wrapper for all feature extraction methods.
- Input data: all object attributes initialized when creating a new object of the `featureExtractor` class.
- Output data: dictionary with all features as `"caseDir/features.json"`. Also, there is the possibility to save an extended version of the dictionary including extra indirect measurements as `"caseDir/extendedFeatures.json"`.
- Arguments: -

- Returns: -

self.getBovineArch()

This method recognizes the presence of a bovine aortic arch. In order to do that, it performs the following process:

1. Selects all those segments form the segments array that make up the LCCA, using the `self.cellIdToVesselType` dict. If no LCCA is found, returns `False`.
2. Gets all segments that make up LCCA in an ordered, flat array, which we call singleSegment[4].
3. The first point of the singleSegment should be the proximal bifurcation point. We search which other segments include this bifurcation point. This will indicate which segments are in contact at the bifurcation of the LCCA.
4. If the BT (edgetypes 2 and 14) is found among the segments in contact at the proximal bifurcation of the LCCA, searches for a BT transition segment. In all other cases, returns `False`.
5. If multiple BT segments are found associated to groupIds, then returns `True`. Otherwise, returns `False`.

The method returns a Boolean variable (`True` if bovine arch is detected and `False` otherwise).

- Description: detects presence of bovine aortic arch.
- Input data: all object attributes initialized when creating a new object of the `featureExtra-ctor` class.
- Output data: -
- Arguments: self.
- Returns: presence of bovine aortic arch (*bool*).

self.getARSA()

This method recognises the presence of an ARSA. In order to do that, it performs the following process:

1. Selects all those segments form the segments array that make up the RSA, using the `self.ce-llIdToVesselType` dict. If no RSA is found, returns `False`.
2. Gets the RSA singleSegment.
3. Gets RSA proximal bifurcation point. We search which other segments include this bifurcation point. This will indicate which segments are in contact at the bifurcation of the RSA.

[4]The singleSegment convention will be continuously used during this description. The singleSegment is obtained through ordering the multiple segments that share the same vessel type. To order these segments and join them in a sensible manner, we first search for the segments with the closest start- or endpoint to the AA (the segments is flipped if necessary). Once that is set, the next segment is chosen as the one with a start- or endpoint closest to the first segment's endpoint, and so on with the rest of the segments.

4. If the AA (type 1) is found among the segments in contact at the proximal bifurcation point, returns `True`. Otherwise returns `False`.

The method returns a Boolean variable (`True` if ARSA is detected and `False` otherwise).

- Description: detects presence of ARSA.
- Input data: all object attributes initialized when creating a new object of the `featureExtractor` class.
- Output data: -
- Arguments: self.
- Returns: presence of ARSA (*bool*).

self.findAAType()

Finds AA type according to the relative vertical position of the highest AA surface point (point A) and the origin of the BT (point B), in terms of the proximal diameter of the LCCA (D). We follow the usual convention:

- Type I: if axial distance between A and B is less than D.
- Type II: if axial distance between A and B is more than D and less than 2×D.
- Type III: if axial distance between A and B is more than 2×D.

In order to determine the AA type, the method performs the following process:

1. Gets all AA segments (edgetype 1).
2. Gets the S coordinate of the BT origin (B) and LCCA proximal diameter (D) from `self.featureExtractorDict` (this method is called after these are computed). Gets point with highest S coordinate from `self.clippedModelCoordinates` (A).
3. Computes ratio between (A - B) / D and returns the corresponding AA type following the convention described above.
4. If no AA segments are found, the BT origin failed to be located, or the LCCA diameter failed to be computed, `math.nan` is returned and the AA type is undetermined.

This method returns an integer in [1, 2, 3] for the AA type, or `math.nan` if it can't be determined.

- Description: recognizes aortic arch type.
- Input data: all object attributes initialized when creating a new object of the `featureExtractor` class. Additionally, the BT origin and the LCCA proximal diameter have to be computed when calling this method and written in the `self.featureExtractorDict`.
- Output data: -
- Arguments: self.

- Returns: AA type (*int*). `math.nan` if failed computation.

self.findProximalDiameter()

Finds proximal diameter of the vessel defined by the input variable `vesselName`. For the AA, it returns the diameter at the AA apex, following the convention that was employed for the manual measurements that were used for validation of the feature extraction methods. In order to determine the position of the centerline point form which we can extract the diameter at the proximal end of the vessel, we use the blanking variable from the branch model. The process goes as follows:

For the AA (`vesselName == "AA"`):

1. All segments array AA (edgetype equal to 1) segments are identified.
2. If no AA segments are found, returns `math.nan`. Otherwise, pool all AA centerline points from `self.segmentsArrayAff` and select point with highest S coordinate.
3. Returns twice the radius corresponding to that centerline point from `self.radius`.

For all other segments (`vesselName in ["RCCA", "RSA", "RVA", "LCCA", "LSA", "LVA"]`):

1. All segments array segments with the corresponding vessel type are identified.
2. If no segments are found, returns `math.nan`. Otherwise, the singleSegment is generated.
3. Starting from the proximal bifurcation point, iteratively analyzes the singleSegment points and selects the first one that fulfills the following conditions: 1) The point has blanking equal to 0, 3) the groupId can be found in `self.groupIdsToVesselTypesDict.keys()` and 3) the groupId from the branch model corresponds to the vessel type as in `self.groupIdsToVessel-TypesDict`. This point is recognized as the origin of the vessel, and is recorded in `self.feat-ureExtractorDict` as it is used for other measurements (e.g., `self.findAAtype()`).
4. Returns twice the radius corresponding to the origin point from `self.radius`.

This method returns the proximal diameter of the analyzed vessel or `math.nan` if it can't be determined.

- Description: measures the proximal diameter of the input `vesselName`. For the AA, it measures the diameter at the apex.
- Input data: all object attributes initialized when creating a new object of the `featureExtra-ctor` class.
- Output data: -
- Arguments: self, `vesselName` (*str*).
- Returns: proximal diameter of `vesselName` (*float*). `math.nan` if failed computation.

self.computeRelativeLength()

This method computes the relative length of a vessel, understood as the ratio between the Euclidean distance between the origin (A) and the distal end of a vessel (B), and the distance along the centerline of the vessel along those two points. For convention, we use the origin computed through the `self.findProximalDiameter()` method as point A, and the distal bifurcation of the vessel as point B. The relative length is always comprised between 0 and 1. High relative lengths (close to 1) are linked to vessels with very straight paths (non-tortuous), while relative lengths closer to 0 are typical from very tortuous anatomies. The computation process goes as follows:

1. All segments array segments with the corresponding vessel type are identified.
2. If no segments are found or the origin failed to be computed, returns `math.nan`. Otherwise, the singleSegment is generated.
3. The startpoint is selected as the origin point.
4. For the endpoint, the last point of the singleSegment is chosen. For the SAs, however, the closest singleSegment point to the corresponding VA origin is chosen (if it is not `math.nan`).
5. Once both points are chosen, the Euclidean distance is measured, and it is divided by the distance covered along the singleSegment between both points. The resulting ratio is returned.

The relative length for the input `vesselName` is returned.

- Description: measures the relative length of the input `vesselName`.
- Input data: all object attributes initialized when creating a new object of the `featureExtractor` class. Additionally, the origin points for all vessels should be pre-computed.
- Output data: -
- Arguments: self, `vesselName` (*str*).
- Returns: relative length of `vesselName` (*float*). `math.nan` if failed computation.

self.computeAbsoluteAngle()

For a given input `vesselName`, this method searches for a centerline point at a distance from the origin equal to the proximal diameter towards the distal end. The goal is to define a vector between the vessel's origin and this point, which is chosen by convention, to determine an approximation to the spherical departure angles (polar and azimuth) of the vessel. The polar and azimuth angles are computed using Cartesian to spherical coordinate transformations:

$$\phi = \arctan \frac{\sqrt{x^2 + y^2}}{z} \tag{6.1}$$

$$\Theta = \arctan \frac{y}{x} \tag{6.2}$$

Here, $\phi \in [\frac{-\pi}{2}, \frac{\pi}{2}]$ and $\Theta \in (-\pi, \pi]$ are the polar and azimuth angles, respectively. x, y and z correspond to the R, A and S coordinates in our system.

The method performs the following computations:

1. All segments array segments with the corresponding vessel type are identified.
2. If no segments are found or the origin failed to be computed, returns `math.nan`. Otherwise, the singleSegment is generated.
3. The origin point and the proximal diameter are obtained form `self.featureExtractorDict`.

4. The absolute angle point is found by searching the point along the singleSegment that is at a distance closest to the diameter.
5. The vector described starting from the origin and ending at the absolute angle point is used to compute the absolute polar and relative angles.

This method returns the polar and azimuth absolute departure angles of the input `vesselName`.

- Description: measures the absolute departure angles of the input `vesselName`.
- Input data: all object attributes initialized when creating a new object of the `featureExtra-ctor` class. Additionally, the origin points and proximal diameter for all vessels should be pre-computed.
- Output data: -
- Arguments: self, `vesselName` (*str*).
- Returns: polar absolute angle `vesselName` (*float*) (`math.nan` if failed computation), azimuth absolute angle `vesselName` (*float*) (`math.nan` if failed computation).

self.computeRelativeAngle()

For a given input `vesselName`, this method searches for a centerline point at a distance from the origin equal to the proximal diameter of the preceding vessel towards the proximal end, in the direction of foreseen catheter access. The goal is to define a vector between this point and the vessel's origin, which is chosen by convention, to determine an approximation to the spherical departure angles (polar and azimuth) of the vessel, relative to the orientation of the preceding vessel. The differences between the resulting polar and azimuth angles and the previously computed absolute angles will be the relative angles. The procedure is as follows:

1. All segments array segments with the corresponding vessel type are identified.
2. If no segments are found or the origin failed to be computed, returns `math.nan`. Otherwise, the singleSegment is generated.
3. The origin point and the proximal diameter are obtained form `self.featureExtractorDict`.
4. Gets all segments in contact with bifurcation point (first point of the singleSegment) and all AA segments. Filter out those that do not have a point within a diameter from the `vesselName`'s origin. Choose the point along each remaining segment at a distance closest to the proximal diameter of such segment. This leaves us with a list of potential candidates for the relative point position.
5. Now, for each `vesselName`, a decision tree has been defined to choose the final relative point among the candidates:
    - For BT: the point should always be on the AA centerline. There could be multiple candidates; as a deciding criteria, we choose the point closest to the IJK coordinates origin (origin of the RAS system), as it it always closer tot he descending aorta (where the catheter would be introduced if transfemoral access is used).
        – If AA is found among the potential candidates (type 1), choose point closest to the RAS origin.
        – Otherwise, return `math.nan` for the relative point.

- For RCCA: first of all, we have to check if ARSA is present, as if it is, the BT is not present and the RCCA originates from the AA. If it is, the preceding vessel should be the AA, and we would choose the point closest to the RAS origin. If ARSA is not present, the BT is selected as the preceding vessel and, from the potential relative points on the BT, the one closest to an AA segment is chosen.
  - If ARSA, find AA segments (types 1 and 14) and select point closest to RAS origin.
  - If ARSA is found, but no AA segments are found among the potential relative points, return `math.nan` for the relative point.
  - If not ARSA, check for BT segments (types 2 and 14), and select the point closest to the AA.
  - If no BT segments are found and not ARSA, return `math.nan` for the relative point.
- For RSA: same criteria as for the RCCA are used.
- For RVA: the only contemplated possibility is for the RVA to originate from the RSA, so the presence of the RSA is asserted and the closest point to the AA is chosen as the relative point.
  - If the RSA is present, choose the closest point to the AA.
  - Otherwise, return `math.nan` for the relative point.
- For LCCA: first check if bovine arch is present. If it is, check for relative point candidates on the BT (types 2 or 14) and choose closest to AA. If it is not, choose a point over the AA (closest to RAS origin). If the arch is not bovine, choose a point over the AA (types 1 or 14) ans select point closest to RAS origin.
  - If bovineArch, choose point over BT and select one closest to AA.
  - If bovineArch but no candidate over BT, choose the AA point closest to the RAS origin.
  - Otherwise, return `math.nan` for the relative point.
  - If no bovineArch, choose AA point closest to the RAS origin.
  - Otherwise, return `math.nan` for the relative point.
- For LSA: only contemplated possibility is for the LSA to originate from the AA.
  - If available, choose AA point closest to RAS origin.
  - If none are found, return `math.nan` for the relative point.
- For LVA: first check if the LVA originates from the AA, by checking the vessels in contact at the bifurcation point. If so, choose the AA point closest to the RAS origin. Otherwise, and in most cases, the LVA will originate from the LSA, so choose the LSA point closest to the AA.
  - If AA in contact with LVA proximal point, choose AA point closest to origin.
  - If not, choose LSA point closest to AA.
  - Otherwise, return `math.nan` for the relative point.

6. Once the relative point is chosen, the spherical angles are computed corresponding to the vector originating at the relative point and ending at the origin of the `vesselName`.
7. The absolute angles are extracted from `self.featureExtractorDict` and are subtracted from those newly obtained with the relative point. The resulting values are set as the relative polar and azimuth angles.
8. If the relative point failed to be computed, then returns `math.nan` for both angles.

This method returns the relative polar and azimuth angles of the input `vesselName`.

- Description: computes the relative spherical departure angles of the input `vesselName`.

- Input data: all object attributes initialized when creating a new object of the `featureExtra-ctor` class. Additionally, the origin points, proximal diameter for all vessels and absolute polar and azimuth angles should be pre-computed.
- Output data: -
- Arguments: self, `vesselName` (*str*).
- Returns: polar relative angle `vesselName` (*float*) (`math.nan` if failed computation), azimuth relative angle `vesselName` (*float*) (`math.nan` if failed computation).

7 Usage

In this section, the installation process and the use instructions for the Arterial©2021, VHIR and UB. All rights reserved framework are explained in detail. This was tested in MacOS Big Sur 11.3 Beta (nnU-Net could not be tested due to lack of Nvidia GPU compatibility) and Linux Ubuntu 20.04.

7.1 Installation

7.1.1 System requirements

For deployment, the Arterial©2021, VHIR and UB. All rights reserved, framework has been partly tested on Ubuntu 18.04 and 20.04. Check for Cuda compatibility (cuDNN, cudatoolkit) between your Nvidia GPU and the PyTorch version installed through the nnU-Net package[5]. Your GPU should have at least 3GB of VRAM to correctly perform inference with the segmentation models (otherwise, you could run into memory problems).

For development, Arterial©2021, VHIR and UB. All rights reserved, has been tested on MacOS 10.15 or newer, excluding the segmentation module due to the lack of Cuda support. The installation process is the same for Linux and MacOS.

7.1.2 Downloads

Download and install 3D Slicer[6] (latest stable release) and Anaconda[7]. Follow the instructions in each website to complete the installation of these packages.

7.1.3 Installation process

After the Anaconda installation, create a Conda environment installing VMTK following VMTK's indications[8] with the following commands:

```
$ conda install anaconda-client
$ conda update conda anaconda-client
$ conda config --set restore_free_channel true
$ conda create -n arterialenv -c vmtk python=3.6 itk vtk vmtk llvm=3.3
```

[5]Check `https://github.com/MIC-DKFZ/nnUNet` for more information.
[6]`https://download.slicer.org/`
[7]`https://www.anaconda.com/products/individual`
[8]`http://www.vmtk.org/download/`

The VMTK package is, at the time of writing in the process of updating to VTK9 (it was formerly based off VTK8) and the installation process through Conda is not properly updated. During the coming weeks, the developers are planning to incorporate the following version of VMTK (1.5) into a PyPI submission, which will enable the installation through `pip` and will eliminate our current dependence on Anaconda. This section will be updated shortly when a stable version of VMTK is available for installation through `pip`.

Regarding the nnU-Net framework, there installation through `pip` is already available. We also use `pip` to install additional packages needed for the Arterial©2021, VHIR and UB. All rights reserved, framework, including:

- nnunet.
- nibabel.
- networkx.
- ipykernel (if development through Jupyter notebook is needed).
- pytorch==1.6 (later versions incompatible with Python 3.6.1, needed for VMTK).
- vtk==9.0.3 (needs updating after VMTK installation).

Additionally, `pip` is also used to install the packages needed for GNN processing, including:

- graph_nets.
- "tensorflow_gpu>=1.15,<2"
- "dm-sonnet<2"

All these packages are included in a `requirements.txt` file within the GitHub repository of the project. To perform the installation, first clone the repository into a known location within your system. For conciseness, we recommend to clone the Arterial©2021, VHIR and UB. All rights reserved, repository inside the Conda environment directory that will have been created after executing the commands listed above[9]. Change to the directory of the Conda environment through the command line by use of the `cd` command, clone the GitHub repository[10] and install the PiPY available packages listed in `requirements.txt`:

```
$ git clone https://github.com/perecanals/arterial.git
$ cd arterial
$ python -m pip install -r requirements.txt
```

Make sure that the Conda environment is activated before executing the `pip` installation.

If Jupyter notebook, will be used, you can use the following command to create a new visible kernel corresponding to the created Conda environment:

```
$ python -m ipykernel install -user -name arterialenv -display-name "Python 3.6 -
(arterialenv)"
```

[9]This will be at `/path/to/anaconda3/envs/arterialenv/`

[10]At the moment, the GitHub repository is private, so access should granted before executing this commands.

Arterial©2021, VHIR and UB. All rights reserved, uses certain paths as environment variables, which have to be set up in the `.bashrc` or `.zshrc` file (depending on the shell you are using). To set up these paths, first locate the absolute paths to the `arterialDir` directory within your environment. This directory should be that containing the `performAnalysis.py` file as well as the directories for the different modules described in section 6. If you just successfully installed the packages in the `requirements.txt`, execute the following commands:

```
$ cd arterial
$ pwd
```

The path returned when executing these commands should be set as `arterialDir`. Then, edit the `.bashrc` (or `.zshrc`, we will only use the `.bashrc` file from this point onward as example):

```
$ open ∼/.bashrc
```

And now, write the following lines at the end of the `.bashrc` file:

```
EXPORT arterialDir="/path/to/arterialDir"
EXPORT nnUNet_base_raw_data="/path/to/arterialDir/segmentation/nnUNet_base/nnU-
Net_raw_data"
EXPORT nnUNet_preprocessed="/path/to/arterialDir/segmentation/nnUNet_base/nnUN-
et_preprocessed"
EXPORT RESULTS_FOLDER="/path/to/arterialDir/segmentation/models"
```

Additionally, locate the path to the Slicer executable in your system and add it at the end of the `.bashrc` file:

```
EXPORT slicerPath="/path/to/Slicer/executable"
```

Once these environmental variables are set, execute the following command:

```
$ source ∼/.bashrc
```

To check if the path definition was successful, executing the following command should return the `arterialDir` that was set:

```
$ echo $arterialDir
```

Finally, additional extensions and packages should be installed within the Slicer application and environment. Open the Slicer application, open the Extension Manager and install the SlicerVMTK extension. If development with Jupyter notebook is desired through the Slicer environment, install the SlicerJupyter extension as well.

Lastly, several packages have to be install through `pip` in the Slicer Python environment, including:

- nibabel.
- scikit-image.
- ipykernel (only needed if development through SlicerJupyter is desired).

To install these packages, open the Python interpreter within Slicer and run the following command:

```
slicer.util.pip_install("nibabel", "scikit-image", "ipykernel")
```

In some versions of Slicer, there are some packages that need updating from the pre-installed versions. Use commands `slicer.util.pip_uninstall()` and `slicer.util.pip_install()` afterwards for these packages, inputting the package names as strings in the Slicer Python prompt (e.g., this has been known to happen with scipy).

7.2 Use instructions

After the installation process has been completed, Arterial©2021, VHIR and UB. All rights reserved, is ready for use. To test it, set a CTA volume as a NIfTI file in a new directory. For conciseness, we recommend that the NIfTI file and the directory containing it share the same name (the `caseId`. Use `demo/demo.nii.gz`, for example). Then, to perform the complete analysis, run the `performAnalysis.py` script adding the absolute path to the NIfTI file as the `-casePath` argument as:

```
$ python /path/to/performAnalysis.py -casePath /path/to/demo/demo.nii.gz
```

Make sure that the Conda environment is activated (`$ conda activate arterialenv`). Running this command should start the computation process, which can take between 4 to 10 min depending on the machine used.

If you are running Arterial on a Linux headless server, something fairly common, we advise the use of a dummy X server to avoid problems when using Slicer for lack of context when a display is not connected to the machine. We have used the package Xvbf before with successful results. Xvbf should be first installed:

```
$ sudo apt-get update
$ sudo apt-get install xvfb
```

Then, the command used to perform the complete analysis becomes:

```
$ xvfb-run -auto-servernum -server-num=1 python /path/to/performAnalysis.py
-casePath /path/to/demo/demo.nii.gz
```

In these cases, line 53 in `performAnalysis.py` should be commented and line 54 should be uncommented (this line calls Slicer's Python interpreter, and in some occasions, Slicer arguments for avoiding the main window to be displayed cause some trouble for the execution of the code.

References

[1] A. Fedorov, R. Beichel, J. Kalpathy-Cramer, J. Finet, J-C. Fillion-Robin, S. Pujol, C. Bauer, D. Jennings, F.M. Fennessy, M. Sonka, J. Buatti, S.R. Aylward, J.V. Miller, S. Pieper, and Kikinis R. 3d slicer as an image computing platform for the quantitative imaging network. *Magn Reson Imaging*, 30(9):1323–41, 2012 Nov.

[2] Fabian Isensee, Paul F. Jaeger, Simon A.A. Kohl, Jens Petersen, and Klaus H. Maier-Hein. nnU-Net: a self-configuring method for deep learning-based biomedical image segmentation. *Nature Methods*, 18(2):203–211, 2021.

[3] L. Antiga, M. Piccinelli, L. Botti, B. Ene-Iordache, A. Remuzzi, and Steinman D.A. An image-based modeling framework for patient-specific computational hemodynamics. *Medical and Biological Engineering and Computing*, 46(9):1097–1112, Nov 2008.

[4] Li Chen, Thomas Hatsukami, Jenq Neng Hwang, and Chun Yuan. Automated intracranial artery labeling using a graph neural network and hierarchical refinement. *arXiv*, 1:1–11, 2020.