OpenCCO: An Implementation of Constrained Constructive Optimization for Generating 2D and 3D Vascular Trees

Bertrand Kerautret, Phuc Ngo, Nicolas Passat, Hugues Talbot, Clara Jaquet

Published in Image Processing On Line on YYYY–MM–DD.
Submitted on YYYY–MM–DD, accepted on YYYY–MM–DD.
ISSN 2105–1232 © YYYY IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
https://doi.org/10.5201/ipol

OpenCCO: An Implementation of Constrained Constructive Optimization for Generating 2D and 3D Vascular Trees

Bertrand Kerautret[1], Phuc Ngo[2], Nicolas Passat[3], Hugues Talbot[4], Clara Jaquet[5]

[1] Université Lumière Lyon 2, LIRIS, Imagine, F-69365 Lyon, France (bertrand.kerautret@univ-lyon2.fr)
[2] Université de Lorraine, CNRS, LORIA, 54000 Nancy, France (hoai-diem-phuc.ngo@loria.fr)
[3] Université de Reims Champagne Ardenne, CReSTIC EA 3804, 51100 Reims, France (nicolas.passat@univ-reims.fr)
[4] CentraleSupelec, Inria, Université Paris-Saclay, 9 Rue Joliot-Curie, 91190 Gif-sur-Yvette, France (hugues.talbot@centralesupelec.fr)
[5] Université Gustave Eiffel, CNRS, ESIEE Paris, 77454, Marne-la-Valle, France

PREPRINT October 15, 2023

Abstract

In this article, we focus on the algorithm called CCO (Constrained Constructive Optimization), initially proposed by Schreiner and Buxbaum [17] and further extended in [7]. This algorithm can be considered as one of the gold standards for vascular tree structure generation. Modeling and/or simulating the morphology of vascular networks is a challenging but crucial task that can have a strong impact on different applications such as fluid simulation or learning processes related to image segmentation. Various implementations of CCO were proposed over the last years. However, to the best of our knowledge, there does not exist any open-source version that faithfully follows the native CCO algorithm. Our purpose is to propose such an implementation both in 2D and 3D.

Source Code

The reviewed source code and documentation associated to the proposed algorithms are available from the web page of this article[1]. The correspondance between algorithms and source codes are mutually given and compilation with usage instructions are included in the `README.md` file of the archive. The archives also contains the script allowing to reproduce experiments included in various figures.

[1] https://ipolcore.ipol.im/demo/clientApp/demo.html?id=5555531082026

1 Introduction

In image processing and analysis, thin structures are among the most difficult to handle. This is especially true for line-like structures in 2-dimensional images or tube-like structures in 3-dimensional images. Such structures are particularly present in the context of (bio)medical imaging. A wide literature has been dedicated to the preprocessing (e.g. filtering), processing (e.g. segmentation) and analysis (e.g. quantification) of vascular structures in so-called angiographic images, which are typical examples of 1-dimensional structures in 2- and 3-dimensional images. Recent surveys on such image processing / analysis approaches [10, 14] summarize the investigation of a wide range of methods for tackling the issues related to blood vessel processing and analysis (from classical image processing to more recent deep learning paradigms).

By contrast with this rich literature, little effort has been dedicated to the generation of tree structures as geometrical objects which could fairly model vascular trees. Being able to build ex nihilo such objects is, however, a crucial research issue. Indeed, the availability of vascular models that present realistic properties with respect to actual vascular trees would open the way to the design of efficient benchmarks for comparing image processing / analysis methods [9], for modeling vascular structures that cannot be easily observed (e.g. at the microvascularization level) [16], for carrying out efficient experiments in computational fluid dynamics [13] or for generating realistic virtual medical images natively endowed with ground-truth [2].

In 1993, Schreiner and Buxbaum [17] proposed an algorithm for building tree structures in $\mathbb{R}^2$ with the purpose of endowing them with correct properties related to vascular networks. In 1999, Karch et al. [7] extended this algorithm to the construction of tree structures in $\mathbb{R}^3$. In 2010, VascuSynth[2] [3] was proposed as an open-source software building upon CCO. Although VascuSynth data are frequently used in the angiographic image analysis literature, the underlying software presents various limitations compared to the native version of CCO (e.g. regarding spatial domain definition, optimization strategy for bifurcation geometry computation). In 2019, Jaquet et al. revisited the CCO algorithm [5] with the purpose of using it for cardiovascular modeling. Unfortunately, the associated software was not made available to the scientific community. A couple of years ago, we decided to provide an open-source version of this algorithm. Doing so, we also carefully revisited the algorithm, in order to efficiently deal with some tricky aspects, computationally expensive steps, and to try to simplify its formulation, especially by discriminating its algorithmic and physiology-guided layers.

In this article, we describe our own interpretation of the CCO algorithm, that we aim to make completely clear and as close as possible to the original one, and the details of its implementation. The sequel of this article is organized as follows. In Section 2, we propose a synthetic summary of the proposed algorithm. Section 3 explains the principal key points of the algorithm. Section 4 describes one representative iteration of the construction algorithm. Section 5 describes the main parameters of the algorithm. Details related to our implementation of the algorithm are given in Section 6. Experimental results are proposed in Section 7. Section 8 provides concluding remarks and perspectives.

2 Overview of the method

2.1 Description of the constructed tree

The proposed method builds a tree $\mathfrak{T}$ in $\mathbb{R}^d$ ($d = 2$ or 3) inside a bounded, closed connected domain $\Omega \subset \mathbb{R}^d$.

[2] https://vascusynth.cs.sfu.ca

From a topological point of view, $\mathfrak{T}$ is composed by a set $\mathcal{V}$ of $n = 2k$ vertices ($k \geq 2$), including one vertex called the starting vertex, $k \geq 1$ vertices called ending vertices, and $k - 1$ vertices called inside vertices; and a set $\mathcal{E}$ of $n - 1$ edges linking two distinct vertices. Each ending / starting vertex is incident to exactly one edge. Each inside vertex is incident to exactly three edges. Each edge is incident to at least one inside vertex and at most one ending / starting vertex. The induced graph $(\mathcal{V}, \mathcal{E})$ is connected and acyclic, i.e. it is a tree.

The path between any vertex $v \in \mathcal{V}$ and the starting vertex being unique, we can non-ambiguously define a "parenthood" function $\varphi : \mathcal{V} \to \mathcal{V}$ such that $\varphi(v) \in \mathcal{V}$ is the successor of the vertex v in the path between v and the root (which is the only vertex where φ is not defined). In particular, for any $v \in \mathcal{V}$ except the starting vertex, $(v, \varphi(v))$ is an edge of $\mathcal{E}$, and this specific edge will also be noted $e(v)$. Conversely, we note $\Phi(v) = \{u \in \mathcal{V} \mid \varphi(u) = v\}$.

Let $v_i \in \mathcal{V}$ ($i \in [\![0, n-1]\!]$) be a vertex of the tree $\mathfrak{T}$. From a geometrical point of view, the vertex v_i is embedded as a point $p_i \in \mathbb{R}^d$, whereas the (putative) edge $e(v_i) \in \mathcal{E}$—also noted e_i for the sake of concision—is embedded as a straight segment $S_i \subset \mathbb{R}^d$. In particular, if the edge e_i is incident to the two vertices $v_i, v_j \in \mathcal{V}$, then the segment S_i is defined as $S_i = [p_i, p_j] = \{\alpha.p_i + (1 - \alpha).p_j \in \mathbb{R}^d \mid \alpha \in [0, 1]\}$.

From a morphological point of view, each segment S_i has a given thickness defined by a radius parameter $r_i \in \mathbb{R}_+^*$ associated to the edge e_i. In practice, the 1-dimensional segment S_i is then modeled as a d-dimensional "tubular" object C_i that can be seen as the dilation $C_i = S_i \oplus B_{r_i}$ of S_i by the Euclidean ball of radius r_i, namely $B_{r_i} = \{x \in \mathbb{R}^d \mid \|x\|_2 \leq r_i\}$. The whole tree $\mathfrak{T}$ is then defined as the union of these tubular shapes, i.e. $\mathfrak{T} = \bigcup_{i=0}^{n-1} C_i$. This tree $\mathfrak{T}$ lies inside Ω, i.e. $\mathfrak{T} \subseteq \Omega$. If two edges e_j and e_k are not incident to a same vertex v_i, then we should have $C_j \cap C_k = \emptyset$, so that the acyclicity of the graph $(\mathcal{V}, \mathcal{E})$ remains valid in the tree $\mathfrak{T} \subset \mathbb{R}^d$.

From now on, for the sake of readability, we will sometimes make no distinction between the notions of vertex (v_i) vs. points (p_i) and the notions of edges (e_j) vs. segments (S_j). In other words, we will make no distinction of notation between the tree $\mathfrak{T}$ considered as a graph and as an embedded volume.

2.2 Input / output

The method has the following inputs:

- $\Omega \subset \mathbb{R}^d$: the domain where the tree has to lie;
- $p_0 \in \Omega$: the position of the starting point / vertex;
- $k \in \mathbb{N}_+$: the number of ending points / vertices;

and provides the following outputs:

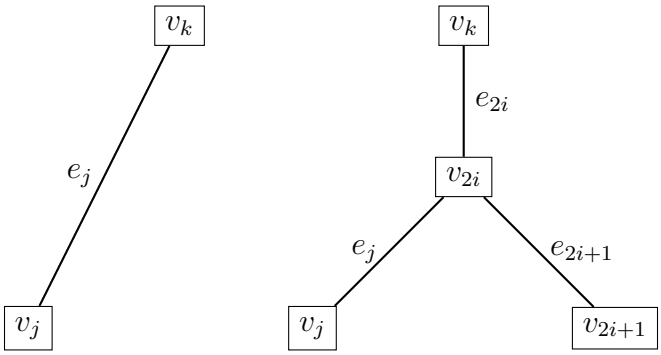
- $\mathcal{V} = \{p_i\}_{i=0}^{n-1}$, the set of the $n = 2k$ points / vertices of the tree $\mathfrak{T}$;
- $\mathcal{E} = \{S_i\}_{i=1}^{n-1}$, the set of the $n - 1$ edges / segments S_i together with their respective radii $r_i \in \mathbb{R}_+^*$.

2.3 Sketch of the algorithm from a topological perspective

The tree $\mathfrak{T}$ is incrementally constructed from a trivial tree—composed by two vertices and one edge—by iteratively adding new edges composed of two new vertices. At each iteration, each new edge is connected to the current tree by "breaking" a current edge of the tree into two parts. From the only topological point of view—i.e. if we consider the vertices v_i and the edges e_i, without dealing with

Algorithm 1: Tree construction (sketch: topological point of view).

Input: $k \in \mathbb{N}_+^*$
Output: $(\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \{v_i\}_{i=0}^{n-1}$ and $\mathcal{E} = \{e_i\}_{i=1}^{n-1}$ $(n = 2k)$
// Initialisation of the tree
Let v_0, v_1 be two vertices
Let $e_1 = (v_1, v_0)$
Let $(\mathcal{V}, \mathcal{E}) = (\{v_0, v_1\}, \{e_1\})$
// Incremental addition of new vertices and edges
foreach $i \in [\![1, k-1]\!]$ **do**
    Let v_{2i}, v_{2i+1} be two vertices
    Set $\mathcal{V} = \mathcal{V} \cup \{v_{2i}, v_{2i+1}\}$
    Choose $e_j = (v_j, v_k) \in \mathcal{E}$
    Set $e_j = (v_j, v_{2i})$
    Let $e_{2i} = (v_{2i}, v_k)$
    Let $e_{2i+1} = (v_{2i+1}, v_{2i})$
    Set $\mathcal{E} = \mathcal{E} \cup \{e_{2i}, e_{2i+1}\}$



(a) Initial (b) Final

Figure 1: Addition of a new edge in the tree (see Algorithm 1).

their spatial embeddings, namely the points p_i and the segments S_i of radii r_i—then the algorithmic sketch of CCO is given in Algorithm 1 (with an illustration of edge updating in Figure 1).

This algorithm is basically the construction of a (nearly) binary tree. The relevance and the main difficulty of CCO lie in the additional handling of the spatial embedding of this tree. It is indeed required to generate this tree by associating to each vertex v_i a point p_i and to each edge e_i a segment S_i associated with a radius r_i, while satisfying geometrical and morphological constraints related to the domain Ω and various parameters. These constraints are briefly summarized hereafter. Their algorithmic handling will be explained in detail in the next section.

2.4 Geometrical and morphological constraints

The construction of the tree $\mathfrak{T}$ is enriched / constrained by various factors.

(1) Each edge $e_i = (v_i, v_j)$ is associated to a segment S_i with a radius r_i that depends on its spatial embedding (i.e. the distance between the positions of p_i and p_j in Ω) but also on the positioning of the edge into the tree and the radii associated to the other segments. This radius handling is discussed in Section 3.1.

(2) For each edge $e_i = (v_i, v_j)$ associated to two new vertices v_i, v_j and associated points p_i, p_j, one of these points, say p_i, which is an ending point, has a position randomly chosen under

constraining criteria. The position of the second point, say p_j which is an inside point, is computed by an optimization process. This optimization process is guided by parameters which allow one to model physiological priors. This optimization procedure is discussed in Section 3.2.

(3) During the construction of the tree, it is forbidden that (i) the segment associated to a new edge be too short and (ii) that two ending points be too close. It is also forbidden that (iii) two segments intersect. (Here, the segments are viewed as volumes.) This requires a segment intersection checking at various stages of the process. These checking procedures are discussed in Section 3.3.

3 Main key points

3.1 Radius computation

In the tree $\mathfrak{T}$ induced by $(\mathcal{V}, \mathcal{E})$, each vertex v_i ($i \in [\![0, n-1]\!]$) is associated to a point $p_i \in \mathbb{R}^d$, and each edge $e_i \in \mathcal{E}$ ($i \in [\![1, n-1]\!]$) is associated to a segment $S_i = [p_i, p_j] \subset \mathbb{R}^d$ (with $v_j = \varphi(v_i)$) of length $\ell_i = \|S_i\|_2 = \|p_i - p_j\|_2$. In order to complete this spatial embedding, we also aim to determine a radius $r_i \in \mathbb{R}_+^\star$ ($i \in [\![1, n-1]\!]$) for each segment S_i.

3.1.1 Relative radii

In a first time, we do not focus on absolute but on relative radii. More precisely, we define for each segment S_i ($i \in [\![1, n-1]\!]$) a value $\rho_i \in \mathbb{R}_+^\star$ which is the ratio between the radius r_i of the segment S_i and the radius r_1 of the segment S_1 which is defined as the only segment containing the starting point p_0. In particular, for any segment S_i we have

$$r_i = \rho_i \cdot r_1 \tag{1}$$

These values ρ_i are defined recursively in a bottom-up fashion by

$$\rho_i = \begin{cases} 1 & \text{if } i = 1 \\ \beta_i \cdot \rho_{\varphi(i)} & \text{if } i > 1 \end{cases} \tag{2}$$

where, by abuse of notation, $j = \varphi(i)$ means $v_j = \varphi(v_i)$. For any vertex $v_i \in \mathcal{V}$ ($i \in [\![2, n-1]\!]$), we set

$$\beta_i = \left(1 + \alpha_i^\gamma\right)^{-\frac{1}{\gamma}} \tag{3}$$

$$\alpha_i = \left(\frac{L_{b(i)}}{L_i} \cdot \frac{R_{b(i)}}{R_i}\right)^{\frac{1}{4}} \tag{4}$$

with $b(i) \neq i$ being the unique index such that $\varphi(b(i)) = \varphi(i)$, and

$$L_i = \left(1 - \frac{|\Phi(v_i)|}{2}\right) + \sum_{v_j \in \Phi(v_i)} L_j \tag{5}$$

$$R_i = \kappa \cdot \ell_i + \left(\sum_{v_j \in \Phi(v_i)} \frac{\beta_j^4}{R_j}\right)^{-1} \tag{6}$$

In particular, these equations involve only two (hyper)parameters, namely γ and κ, that are discussed in Section 5.

It is important to notice that despite the mutually recursive links that exist between some of the values (R, β, α), there is no deadlock due to a putative circular definition. More precisely, in Equations (3–4), the values of β and α depend on the value of R at the same level, whereas in Equation (6), the value of R at the current level depends on the values of R and β (and thus α) at the lower level of the children nodes (Φ). The computation of these values is then non-ambiguous and can be carried out in a bottom-up fashion, from the leaves of the tree to its root.

The meaning of the computed values is the following:

- ρ_i is the radius ratio between segments S_i and S_1;
- β_i is the radius ratio between the segments S_i and the "parent" $S_{\varphi(i)}$;
- α_i is a parametric ratio between the "brother" segments S_i and $S_{b(i)}$;
- L_i is the number of ending points in the subtree starting at segment S_i;
- R_i is a value that models the hydraulic resistance in the segment S_i, viewed as a vascular structure.

3.1.2 Absolute radii

The radius of the segment S_1 evolves during the iterative part of Algorithm 1. In particular at iteration $i \in [\![1, k-1]\!]$ (i.e. at a stage where $i+1$ terminal vertices have been set), the radius r_1 is defined as

$$r_1 = \big(\xi \cdot R_1 \cdot (i+1)\big)^{\frac{1}{4}} \tag{7}$$

where ξ is a parameter that will be discussed in Section 5. The radii of the other edges are then obtained recursively in a top-down fashion from Eqs. (1–6).

3.1.3 Iterative computation and computational cost

Let us suppose that an intermediate tree $\mathfrak{T}$ has been built at iteration $i-1 \in [\![1, k-1]\!]$ of Algorithm 1 and that the values $R_\star$, $L_\star$, $\alpha_\star$, $\beta_\star$ and $\rho_\star$ have been computed for this tree $\mathfrak{T}$.

Now, let us suppose that at the iteration i, we incrementally update this intermediate tree $\mathfrak{T}$. Since $R_\star$, $L_\star$, $\alpha_\star$ and $\beta_\star$ are defined in a bottom-up fashion, they only need to be updated for the new two vertices v_{2i} and v_{2i+1} added at iteration i, the two vertices v_j and v_k incident to the edge e_j that is split at this iteration, and all the vertices located in the unique path between v_{2i} and the starting vertex v_0. For a tree $\mathfrak{T}$ correctly balanced, these updates present a time complexity $\Theta(\log i)$. However, all the values $\rho_\star$, defined in a top-down fashion, have to be updated, which leads to a time complexity $\Theta(i)$.

As a consequence, the computation of the radii of the final tree $\mathfrak{T}$ ex nihilo present a time complexity $\Theta(k^2)$.

3.2 Segment construction

Let us now discuss the way to define a new segment in the tree being constructed. At step i of Algorithm 1, an intermediate tree $(\mathcal{V}, \mathcal{E})$ has been built and we now want to define and add two new points p_{2i} and p_{2i+1} in $\mathbb{R}^d$, and the induced segment S_{2i+1}. One of these points, say p_{2i+1}, is chosen randomly (but under certain constraints to be fulfilled, see Section 3.3). The other point, say p_{2i}, is defined from the knowledge of p_{2i+1} and from the position of two points p_j and p_k that form the segment S_j in the current tree.

In practice, we have as input three points, namely p_{2i+1}, p_j, p_k such that the last two points already form a segment of the current tree. We aim to compute a fourth point p_{2i} that will allow us to define three segments: one segment between p_{2i} and p_{2i+1} (i.e. the "new" segment) and two segments between p_{2i} and p_j (resp. p_k). These segments will derive from the splitting of the existing segment S_j between p_j and p_k and its "bending" to allow the connection with the "new" segment, see Figure 1 for an illustration.

The point p_{2i} must lie in the triangle formed by p_{2i+1}, p_j and p_k. Its position is defined as the result of an optimization process based on the Kamyia algorithm [6], summarized as follows.

For the sake of concision, in the sequel, we note $x_0 = p_k$, $x_1 = p_j$, $x_2 = p_{2i+1}$ and $x = p_{2i}$. The expected solution x lies in the plane defined by the three points x_i ($i \in [\![0, 2]\!]$), assumed non-colinear. We can then consider this optimization problem as a 2-dimensional problem expressed in this plane isomorphic to $\mathbb{R}^2$. From now on, we then assume without loss of generality that we have $x_i \in \mathbb{R}^2$.

In the sequel, we describe one step of the (iterative) Kamyia optimization algorithm.

At the beginning of the step, we know the current position of x, obtained from the previous step, or initialized at the first step as

$$x = \frac{x_0 + x_1}{2} \tag{8}$$

For each $i \in [\![1, 2]\!]$, we define the length l_i of the segment between x_i and x as $l_i = \|x_i - x\|_2$ and we set

$$\Delta_i = \frac{f_0 \cdot l_0}{r_0^4} + \frac{f_i \cdot l_i}{r_i^4} \tag{9}$$

where f_i corresponds to the flow[3] in the segment between x_i and x, and r_i is the radius of this segment (defined in the previous sections, see Equation (1)). This formula derives from Hagen-Poiseuille's law; in particular, Δ_i represents a drop of pressure. When computing the Δ_i, both the flows f_i and the lengths l_i are known (the l_i are derived from the current position of the points x_i). The radii r_i are obtained from the previous step (see below), except at the initial step of the process, where $r_0 = r_1 = r_2$ are set from the current tree.

We assume [15] that

$$f_i \propto r_i^3 \tag{10}$$

and [6]

$$r_0^6 = f_0 \cdot \left(\frac{r_1^6}{f_1} + \frac{r_2^6}{f_2}\right) \tag{11}$$

Equations (9) and (11) lead to a two-equation non-linear system

$$\begin{cases} \Delta_1 r_1^4 \left(f_0\left(\frac{r_1^6}{f_1} + \frac{r_2^6}{f_2}\right)\right)^{\frac{2}{3}} - f_0 l_0 r_1^4 - f_1 l_1 \left(f_0\left(\frac{r_1^6}{f_1} + \frac{r_2^6}{f_2}\right)\right)^{\frac{2}{3}} = 0 \\ \Delta_2 r_2^4 \left(f_0\left(\frac{r_1^6}{f_1} + \frac{r_2^6}{f_2}\right)\right)^{\frac{2}{3}} - f_0 l_0 r_2^4 - f_1 l_1 \left(f_0\left(\frac{r_1^6}{f_1} + \frac{r_2^6}{f_2}\right)\right)^{\frac{2}{3}} = 0 \end{cases} \tag{12}$$

where the two unknowns are r_1^2 and r_2^2 (the system is then of degree 4). To solve this system, we can use a non-linear solver such as the C++ library Ceres Solver [1], considered in our implementation.

One may then derive the coordinates of x [6] as

$$x = \frac{\sum_{i=0}^{2} \frac{r_i^2}{l_i} \cdot x_i}{\sum_{i=0}^{2} \frac{r_i^2}{l_i}} \tag{13}$$

[3] In this algorithm, we assume that the flow at each ending points is the same. Then, the values of f_i considered here are proportional to the number of ending points in the subtrees starting at x_i. In particular, we have $f_0 = f_1 + f_2$, and $\frac{f_1}{f_2} = L_j$.

This process allows to compute, at each step, the values l_i from the current position of x, then r_i (Eq. (12)) and finally the new position of x (Eq. (13)). This iterative procedure terminates when the convergence tolerance of gradient is reached. (In our implementation this tolerance is set to 10^{-10}.)

3.3 Constraints

3.3.1 Preliminary remark: evolution of the size of the domain

The domain $\Omega \subset \mathbb{R}^d$ has a given size $|\Omega|$ (area for $d = 2$ and volume for $d = 3$). If we consider that the tree has k ending points, then the influence area of each ending point is defined as $A = \frac{|\Omega|}{k}$. Since these ending points are added incrementally (one per iteration), we assume that the domain grows at the same rate, i.e. at iteration $i \in [\![0, k-1]\!]$ the domain Ω_i is defined as

$$\Omega_i = \left\{ \frac{i+1}{k} \cdot q \mid q \in \Omega \right\} \tag{14}$$

where $\frac{i+1}{k}$ is a scale factor between Ω_i and the targeted final domain Ω. Note that the iteration 0 corresponds to the initialisation step whereas the last iteration $i = k-1$ corresponds to the end of the process, with $\Omega_{k-1} = \Omega$. In this context, the influence area of an ending point p can be seen as a Euclidean ball of size A. If $d = 2$ (resp. $d = 3$), this ball has a radius equal to $(\frac{A}{\pi})^{\frac{1}{2}}$ (resp. $(\frac{3A}{4\pi})^{\frac{1}{3}}$). It is important to note that when going from iteration i to iteration $i+1$, this radius remains constant; however, the positions of the points evolve from p at iteration i to $(1 + \frac{1}{i+1}) \cdot p$ at iteration $i+1$, at the same rate as the evolution of Ω_i to Ω_{i+1}. This growing rate impacts the length ℓ_i of the segments S_i, but also their radius r_i.

3.3.2 Distance constraints

At each iteration i of the process, two new points p_{2i}, p_{2i+1} are defined. One of them, say p_{2i+1}, is an ending point and is defined quasi randomly. More precisely, it is chosen randomly but under various constraints:

- p_{2i+1} must be chosen within the domain Ω;
- p_{2i+1} must be sufficiently far from the current tree (and a fortiori not in the tree).

In order to satisfy the first constraint, it is sufficient to check that $p_{2i+1} \in \Omega$. If Ω is a convex domain defined by m hyperplanes, this can be checked in $\Theta(m)$. If Ω is an arbitrary (convex or non-convex) domain, then a sufficient condition is to check that p_{2i+1} belongs to a cover of convex sets that lies inside Ω. It is of course relevant to choose a cover that fits as much as possible Ω. These convex sets can be e.g. a cover of spheres or a partition of cubes (e.g. voxels). (This last strategy was chosen in our implementation.)

In order to satisfy the second constraint, we compute the distance δ of p_{2i+1} to the tree, i.e. to each segment S of the tree (considering its radius $r(S)$). This distance δ must be greater that $\delta_{\max}$ with

$$\delta_{\max} = \left(\frac{1}{i+1} \frac{|\Omega|}{k} \right)^{\frac{1}{d}} \tag{15}$$

Note that, at step i, this test requires to compute $i+1$ point-to-segment distances (to this end, we will use the same constant-time algorithm as for segment-to-segment distance [11], see below), leading to a time cost $\Theta(i)$. If no random choice fulfills this constraint after a certain number of attempts (set to 100 in our implementation), then the distance $\delta_{\max}$ is decreased as $\alpha.\delta_{\max}$ with $0 < \alpha < 1$ ($\alpha = 0.9$ in our implementation), and this process is then iterated as many times as required.

3.3.3 Non-intersection constraints

Once a new ending point p_{2i+1} has been defined in Ω, three segments have to be computed: one in order to connect p_{2i+1} to the remainder of the tree, and two by splitting a segment already present in the tree (see Section 3.2). The optimization process leading to the determination of these three segments does not check their validity with respect to the domain Ω and to the current tree $\mathfrak{T}$.

To tackle the issue of the domain, a solution is to build a distance map of the border $\partial\Omega$ into the domain Ω. Such a distance map can be built in a discrete way on the Cartesian grid $\mathbb{Z}^d$ associated to the part of the Euclidean space $\mathbb{R}^d$ where Ω lies. Once computed, it is then possible to determine the (finite) set of pixels / voxels where the considered segment lies. If the distance map value at each of these pixels / voxels is greater than the radius of the segment plus $\sqrt{d}$, then one can guarantee that the segment does not intersect the border $\partial\Omega$ of the domain, i.e. it is actually inside Ω.

It is also mandatory that a new (or updated) segment S do not intersect the other segments, i.e. that there is no self-intersection within the tree. This can be done by computing, for each segment S' in the tree (except the segments incident to S) the distance D between S and S', namely

$$D(S, S') = \min\{\|x - x'\|_2 \mid (x, x') \in S \times S'\} \tag{16}$$

Such a distance can be computed in constant time $\Theta(1)$ [11], leading to a checking in linear time cost $\Theta(2i)$ with respect to the number of segments within the tree. In particular, if for all the segments S', the distance $D(S, S')$ is greater that the sum of the radii of S and S', then we can guarantee that the new (or updated) segment S does not intersect the remainder of the tree.

Figure 2 shows the whole process of vascular tree construction.

4 Global view of one iteration of the algorithm

In this section, we describe the whole running of one iteration (say iteration i) of the tree construction algorithm. At this stage, we assume that an intermediate version of the tree $\mathfrak{T}$ has been built, and we aim to add a new segment to this tree with all the induced side effects.

4.1 Evolution of the domain size

Before adding a new branch and a new ending point to the tree, the size of the domain is increased, as discussed in Section 3.3.1. This step has a time complexity $\Theta(i)$ if the coordinates of all the points are updated (i.e. the size of the domain is increased), or $\Theta(1)$ if the "size of the space" is decreased, which means that the physical positions of the points remain unchanged, but a scale factor is applied, that impacts the computation of the length. This last choice has been chosen in our implementation.

4.2 New ending point

A new ending point p_{2i+1} is defined. As discussed in Section 3.3.2, this is a random choice under various constraints. Many attempts may then be carried out, and the time complexity of this step is then $\Theta(i \cdot \nu_p)$ where ν_p is equal to this number of attempts.

4.3 Choice of the candidate segments for connection

The creation of a new segment is carried out by connecting p_{2i+1} to an existing segment S of the tree (that will be split into two segments). In theory, all the segments of the current tree may be considered. However, such an exhaustive strategy would lead to a prohibitive computational cost. In

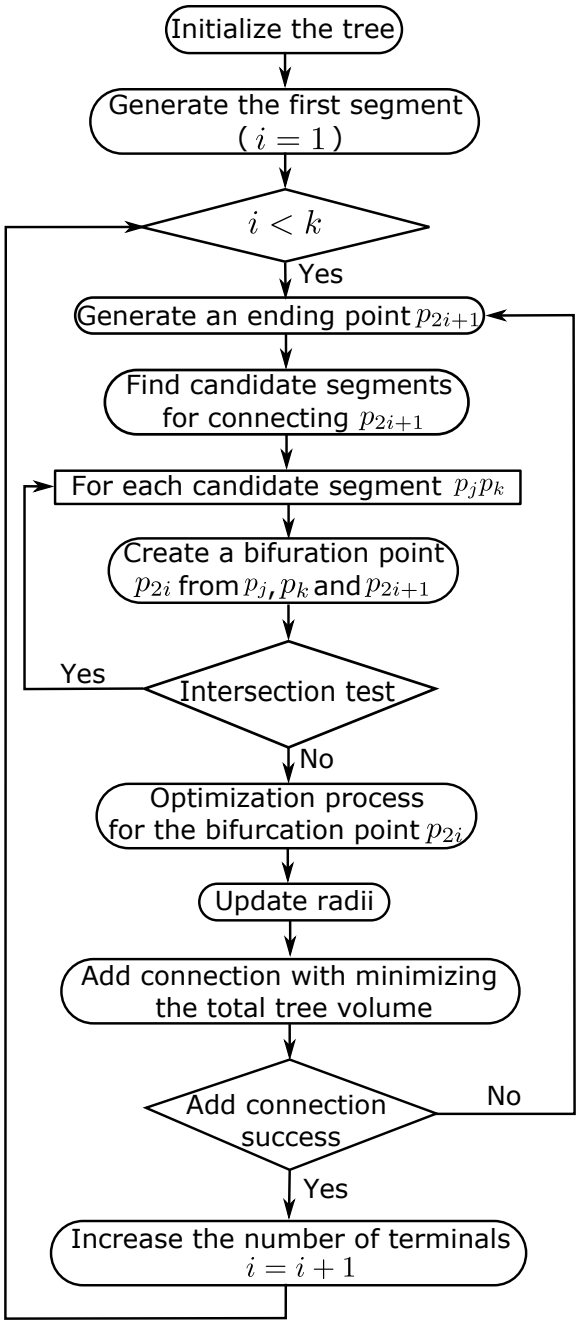


Figure 2: Workflow of the proposed method of vascular tree construction.

practice, only the nearest segments are investigated. The definition of these ν_S segments is carried out in linear time $\mathcal{O}(i \cdot \log \nu_S)$ via point-to-segment distance computation.

4.4 Segment generation and connection

For each segment S within the ν_S candidate segments, a connection between the point p_{2i+1} and the segment S is carried out. This process, documented in Section 3.2, leads to the proposal of a new segment and the update of S as two segments. This proposal is valid if and only if:

- these three segments satisfy the non-intersection requirements discussed in Section 3.3.3. This test of intersection is performed once for a new terminal point (before the optimisation process of bifurcation), and during the optimisation process of bifurcation;

- a ratio constraint that imposes that for each segment, the ratio length / radius of the segment be greater than 2 (i.e., the segment is "long enough"): $2r_i \leq l_i$.

Remark: If the optimization process does not converge or if the obtained bifurcation is not valid, then the candidate segment is discarded. If all the ν_S candidate segments are discarded, then the number ν_S is doubled as many times as necessary. Note that the overall complexity of this step also depends on the complexity of the non-linear solver computation.

4.5 Radius update, volume computation and final choice

For each successful connection, an updated tree is obtained. The radii of the segments of this tree are updated, based on the procedure documented in Section 3.1. It is then possible to compute the whole volume of the tree as the sum of the volumes of each segment. The tree, which minimizes this volume, is chosen as the new current tree at the end of the iteration.

5 Parameters

The proposed algorithm involves various (hyper)parameters introduced in the previous sections. In practice, the purpose is to generate trees that present realistic geometry and morphology with regard to physiological properties. In this context, the tree is seen as a vascular network where blood flows from one unique input (the starting point) towards the k outputs (the ending points).

5.1 Hyperparameters

The proposed algorithm relies in the following hyperparameters:

- $\kappa = \frac{8\mu}{\pi}$, where μ is a physiological parameter defined in the next subsection;
- $\xi = \frac{Q_{\text{out}}}{P_{\text{in}} - P_{\text{out}}}$, where Q_{out} and P_{in}, P_{out} are flow and pressure values defined in the next subsection.

5.2 Physiological parameters

The algorithm also relies in the following physiological parameters:

- μ is the viscosity of the fluid in the structure (in the case of the blood, μ is set to 3.6 cP);
- γ is the Murray bifurcation exponent that controls the geometry of the bifurcations (in general, γ is set to 3);
- Q_{out} is the flow (assumed the same) at each of the k ending points (in our experiments, we set Q_{out} as 0.125 mL/min);
- P_{out} is the pressure (assumed the same) at each of the k ending points (in our experiments, we set P_{out} as 60 mmHg);
- P_{in} is the pressure at the starting point (in our experiments, we set P_{in} as 100 mmHg).

Note that, without loss of generality, the pressure, flow and viscosity parameters are expressed in Pa, mm^3/s and Pa.s in our implementation, respectively.

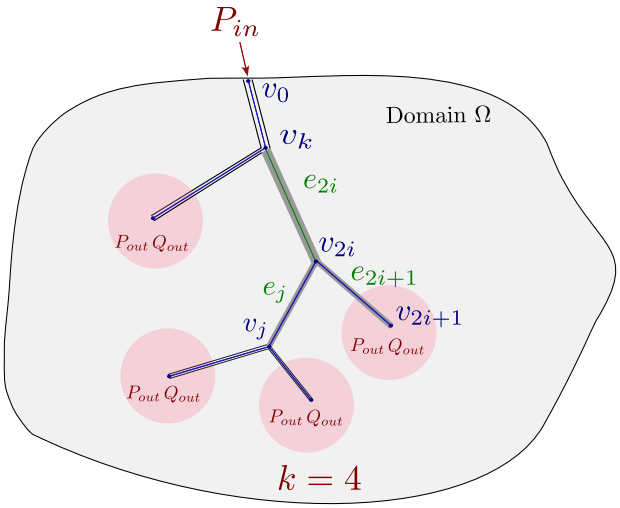


Figure 3: Illustration of the structure used in the CCO algorithm. The red disk area associated to each terminal point represents the neighbouring region receiving Q_{out} flow at pressure P_{out}.

6 Code and implementation details

In this section, we provide information about the principal data structure considered in our implementation of the algorithm (Subsection 6.1), some pseudo-codes related to the most important parts of the algorithm (Subsection 6.2) and a brief source-code description (Subsection 6.3).

6.1 Data structure of a vascular tree

The data structure of a vascular tree is given in the class `CoronaryArteryTree`. Each segment of the vascular tree is defined by the structure `Segment`. It contains the following attributes:

- Distal point of the segment (x_j): `myCoordinate`;
- Index of the segment (j): `myIndex`;
- Radius of the tubular section (r_j): `myRadius`;
- Total number of terminal segments in its children segment (L_j): `myKTerm`;
- Hydrodynamic resistance (R_j): `myResistance`;
- Flow (f_j): `myFlow`;
- Relative radii (radius ratio between the segment and its parent, ρ_j): `myBeta`.

A vascular tree is then composed of:

- A vector of terminal segments: `myVectTerminals`;
- A vector of all the segments of the tree ($\mathcal{E}$): `myVectSegments`;
- A vector of the left (first) and right (second) segments of each indexed segment (Φ): `myVectChildren`;
- A vector of the parent of each indexed segment (φ): `myVectParent`.

Global biological attributes are defined by the structure `BioAlgoParameter`:

- Number of terminal segments (k): `my_NTerm`;
- Perfusion area / volume ($|\Omega|$): `my_aPerf`.
- Final radius of circular area ($|\Omega|/k$): `my_rPerf`;
- Flow (Q_{out}): `my_qPerf`;
- Perfusion pressure (P_{in}): `my_pPerf`;
- Pressure of terminal segment (P_{out}): `my_pTerm`;
- Bifurcation exponent (γ): `my_gamma`;
- Viscosity (μ): `my_mu`;

(Note that 7 of these 8 attributes are parameters, wheres the 8th one, namely `my_rPerf`, is calculated from the others.)

Internal algorithm parameters are defined by the structure `InternAlgoParameter`:

- Current number of terminal segments: `myKTerm`;
- Number of nearest neighbours to be tested: `myNumNeighbor`;
- Average radius of blackboxes: `myRsupp`;
- Scale factor: `myLengthFactor`.

6.2 Pseudo-codes

In this subsection, we provide some pseudo-codes related to the following tasks:

- Generation of the new points in the tree (Algorithms 2 and 3);
- Intersection checking of the branches in the tree (Algorithms 4 and 5);
- Whole process for creating a new bifurcation (Algorithm 6).

6.3 Source codes and dependencies

The code of OpenCCO is written in C++. It is composed of the following classes in the `src` directory:

- `CoronaryArteryTree` contains the structure of a vascular tree and functions to create a bifurcation;
- `DomainController` contains the construction of the domain of a vascular tree;
- `helpers/GeomHelpers` contains functions to generate randomly terminal points, check intersections, Kamyia optimisation, ...;
- `helpers/ExpandTreeHelpers` contains functions to generate a vascular tree;
- `helpers/XMLHelpers` contains functions to export a vascular tree into XLM file.

The program (and the online demo) takes into account the following parameters:

Algorithm 2: Generate randomly a point with distance constraints
→ see C++ code: `generateALocation()` of class `CoronaryArteryTree` in package `src`

Input: A distance threshold $myDThreshold$
Input: A vector of terminal points $myVectTerminals$ of the current tree
Input: A vector of segments $myVectSegments$ of the current tree
Output: A generated point p
Output: A Boolean $isComp$ indicating whether the generation has succeeded

$p \leftarrow$ generate a random point
// Generated point must have a certain distance to all tree terminals
$isComp \leftarrow$ true
$id \leftarrow 1$
while $isComp$ *and* $id < myVectTerminals.size$ **do**
  $isComp \leftarrow distance(myVectTerminals[id].myCoordinate, p) > myDThreshold$
  $id \leftarrow id + 1$
// Generated point must have a certain distance to all tree segments radii
$id \leftarrow 1$
while $isComp$ *and* $id < myVectSegments.size$ **do**
  $isComp \leftarrow distance(myVectSegments[id], p) > myVectSegments[id].myRadius$
  $id \leftarrow id + 1$;
return (p, $isComp$)

Algorithm 3: Generate a terminal point
→ see C++ code: `generateNewLocation()` of class `CoronaryArteryTree` in package `src`

Input: Number of maximum trials $nbTrials$ (= 1000 by default)
Output: A generated point p

$myDThreshold \leftarrow$ compute the distance constraint *// See Section 3.3.2*
$found \leftarrow false$
$n \leftarrow nbTrials$
while $\neg found$ *and* $n \geq 0$ **do**
  $n \leftarrow n - 1$
  $(pt, found) \leftarrow$ generateALocation($myDThreshold$) *// See Algorithm 2*
  if $found$ **then**
    $p \leftarrow pt$
  if $n = 0$ **then**
    $n \leftarrow nbTrials$
    $myDThreshold \leftarrow myDThreshold * 0.9$
return p

- Options for both 2D and 3D cases:
  - number of terminal segments/ending points (`-n` | `--nbTerm` INT, default=1000)
  - perfusion area / volume (`-a` | `--aPerf` FLOAT, default=20000)
  - value of the gamma parameter (`-g` | `-gamma` FLOAT, default=3)
  - minimal distance to border (`-m` | `--minDistanceToBorder` FLOAT, default=5)
  - organ domain using a mask image (`-d` | `--organDomain` TEXT)
  - output the resulting graph as xml file (`-x` | `--exportXML` TEXT)
  - use a squared implicit domain instead a sphere (`-s` | `--squaredDom`, defaut=sphere)
- Specific options in 2D:

Algorithm 4: Check if there is an intersection between two thick segments
→ see C++ code: `isIntersecting()` of class `GeomHelpers` in package `src/helpers`

Input: Two thick segments given by their endpoints and thickness $(segA, segB, rAB)$ and $(segC, segD, rCD)$
Output: A Boolean indicating whether they are intersecting

$cAB \leftarrow (segA + segB)/2$
$cCD \leftarrow (segC + segD)/2$
$lAB \leftarrow (segA - segB).norm$
$lCD \leftarrow (segC - segD).norm$
$dC \leftarrow (cAB - cCD).norm$
$d \leftarrow \frac{(lAB+lCD)}{2} + rAB + rCD$
if $dC > d$ **then**
    return *false*
distanceSeg ← distance between the two segments $[segA, segB]$ and $[segC, segD]$
if *distanceSeg > (rAB + rCD)* **then**
    return *false*
return *true*

Algorithm 5: Check if there is an intersection between a thick segment and all segments of the current tree, except certain segments
→ see C++ code: `isIntersectingTree()` of class `CoronaryArteryTree` in package `src`

Input: A thick segment given by two points and a radius (ptA, ptB, r)
Input: The vector of segments *myVectSegments* of the current tree
Input: A triplet *idExcept* containing the 3 segments indices not tested for intersection
Output: A Boolean indicating whether there is an intersection
// If they are root segment, then ignore the test
if *(idExcept*$[0] = 0$ *or idExcept*$[1] = 0$ *or idExcept*$[2] = 0$*)* **then**
    return *false*
foreach *s : myVectSegments* **do**
    id ← *s.myIndex*
    if $id \neq 0$ *and* $id \neq idExcept[0]$ *and* $id \neq idExcept[1]$ *and* $id \neq idExcept[2]$ **then**
        ptC ← *s.myCoordinate*
        ptD ← *myVectSegments*[*myVectParent*[*id*]].*myCoordinate*
        if *isIntersecting(ptA, ptB, r, ptC, ptD, s.myRadius)* **then** //see Algorithm 4
            return *true*
return *false*

  - initial position of root (`-p` | `--posInit` INT INT, default=image center)
  - output the result into EPS format (`-o` | `--outputEPS` TEXT, default=result.eps)
  - output the result into SVG format (`-e` | `--exportSVG` TEXT, default=result.svg)

- Specific options in 3D:
  - initial position of root (`-p` | `--posInit` INT INT INT, default=image center)
  - output the 3D mesh into OFF file (`-o` | `--outputName` TEXT, default=result.off)
  - output the 3D mesh into text file (`-e` | `--export` TEXT)
  - display 3D view using QGLViewer (`--view`)

The program requires the DGtal library (1.3 or later): `https://github.com/DGtal-team/DGtal`.

Algorithm 6: Create a new bifurcation from a terminal point to a near segment of the tree
→ see C++ code: `isAddable()` of class `CoronaryArteryTree` in package `src`

Input: A new terminal point p
Input: The index $segIndex$ of a segment close enough to p
Input: The maximal number of iterations, $nbIter$
Input: The tolerance to assess the convergence of tree volume, $tolerance$
Output: A Boolean indicating whether p can be added to the tree

$pCurrent \leftarrow$ calculate a first bifurcation from p, the segment $segIndex$ and the image domain
Let $sParent$ be the segment parent of $segIndex$
$pParent \leftarrow sParent.myCoordinate$
Let $sNewLeft$, $sNewRight$, $sCurrent$ be the new left, right and middle segments *// see Section 2.4*
$r0, r1, r2 \leftarrow sCurrent.myRadius$
$vol, volCurr \leftarrow -1$
$isDone \leftarrow false$
$i \leftarrow 0$
while $i < nbIter$ *and* $\neg isDone$ **do**
  $res1 \leftarrow kamyiaOptimization(pCurrent, pParent, sCurrent.myRadius, sNewLeft, sNewRight, 1, pOpt, r0, r1, r2)$ *// see Section 3.2 (Eqs. (9)–(13))*
  if $\neg res1$ **then** // Kamyia does not lead to a solution
    return $false$
  // Check intersection with the current tree for the middle segment
  $idSegPar \leftarrow myVectParent[segIndex]$
  $(idL, idR) \leftarrow (myVectChildren[idSegPar].first, myVectChildren[idSegPar].second)$
  $res2 \leftarrow isIntersectingTree(pOpt, pParent, r0, [idL, idR, idSegPar])$ *// see Algorithm 5*
  if $res2$ **then** // There is an intersection with the middle segment
    return $false$
  if *pCurrent ∉ the image domain or the middle segment ∉ the image domain* **then**
    return $false$
  Do similar test of intersection for the left and right segments
  // Iterate for the optimisation process of the new bifurcation
  Let $cTreeCurr$ and $cTree1$ be a copy of the current tree
  $(sNewLeft.myRadius, sNewRight.myRadius) \leftarrow (r1, r2)$ *// Update values for the segments*
  $cTree1.add(sNewLeft)$
  $cTree1.add(sNewRight)$ *// Add the left and right segments to cTree1*
  $cTree1.update()$ *// Update parameters of cTree1: new segments, physiological parameters...*
  $vol \leftarrow cTree1.computeTotalVolume()$ *// Compute volume of the current tree*
  if $i = 0$ **then** *// The first computation of volume*
    $volCurr \leftarrow vol$
    $cTreeCurr \leftarrow cTree1$
  else
    if $|volCurr - vol| < tolerance$ **then**
      // Check that the resulting segments are long enough (see Section 4.4)
      Let $l0$, $l1$, $l2$ be the length of the segments $[pOpt, pParent]$, $[pOpt, sNewLeft.myCoordinate]$, $[pOpt, sNewRight.myCoordinate]$
      $res3 \leftarrow (2 * r0) \leq l0$ *and* $(2 * r1) \leq l1$ *and* $(2 * r2) \leq l2$
      if $res3$ **then** *// If there is a solution, then save the result*
        $isDone \leftarrow true$
        save $cTree1$
    else
      $volCurr \leftarrow vol$
      $cTreeCurr \leftarrow cTree1$
  Update tree parameters: new position and physiological parameters (see Equations (1–6))
  $i \leftarrow i + 1$
return $isDone$

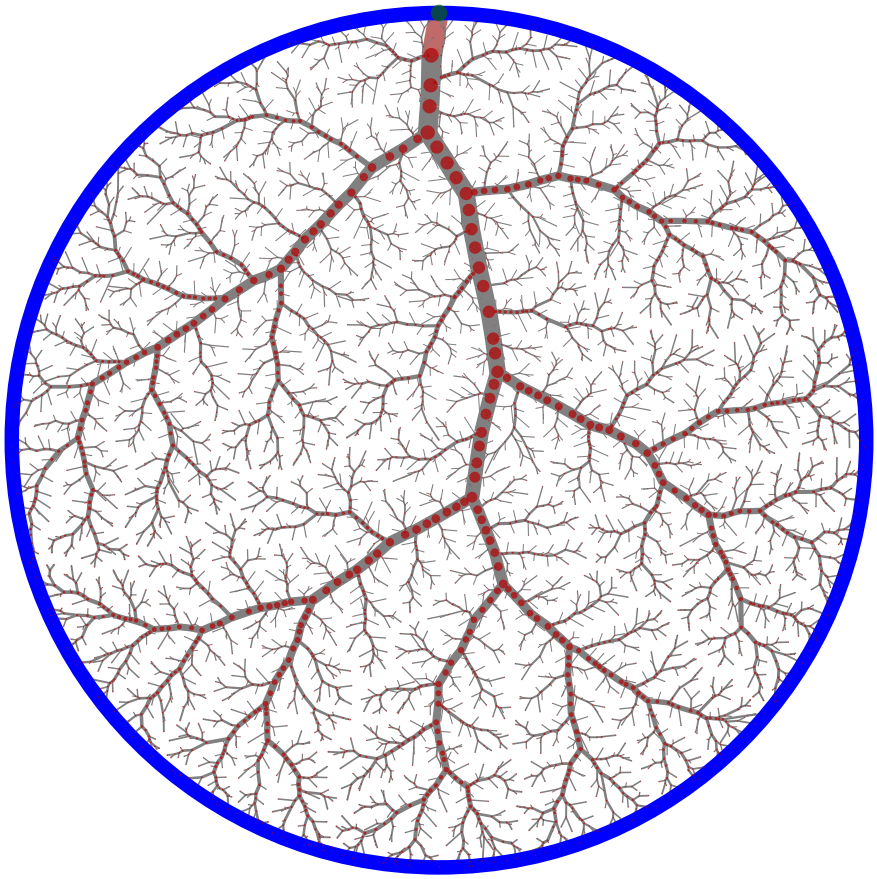


Figure 4: 2-dimensional network generated in a disk domain. The number of terminal points is set to 4000. This result can be compared to the one depicted in [17, Figure 3, p. 486].

7 Experimental results and comparison

In this section, we propose some experiments and results dedicated to compare the behaviour of our implementation with the state-of-the-art version of the 2-dimensional and 3-dimensional algorithm, and variants proposed in the literature.

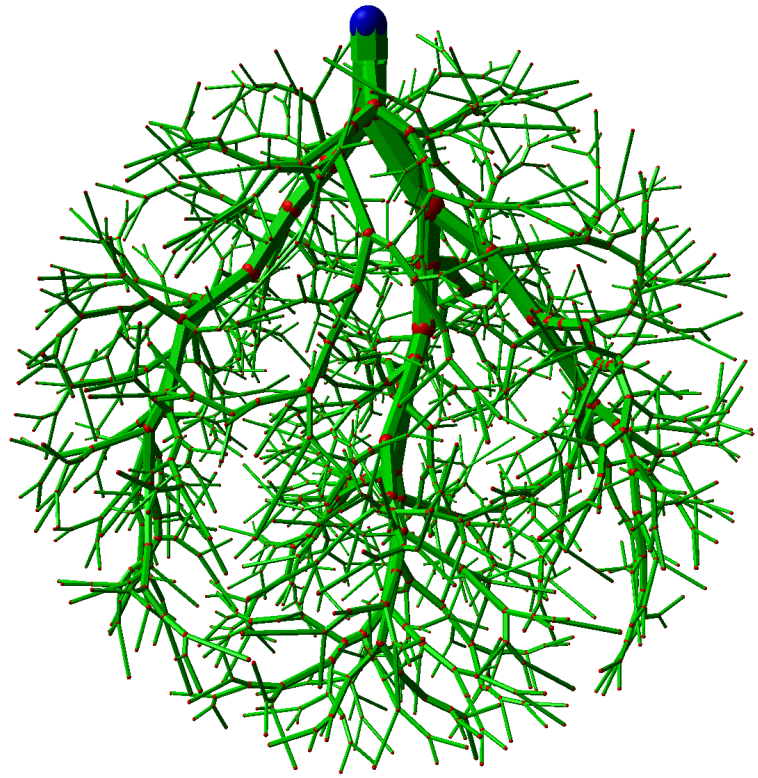
From a qualitative point of view, we first illustrate, in Figures 4 and 5, some 2- and 3-dimensional vascular trees generated in a disk / sphere domain, by the initial algorithms and our implementation. We also provide, in Figure 6, 3-dimensional vascular tree generated in a complex arbitrary domain.

We also illustrate, in Figure 7 the difference between a 3-dimensional vascular tree generated by VascuSynth [3] and with our proposed implementation of CCO with the same input parameters, plus a plot of the diameters of the segments depending on the bifurcation level.

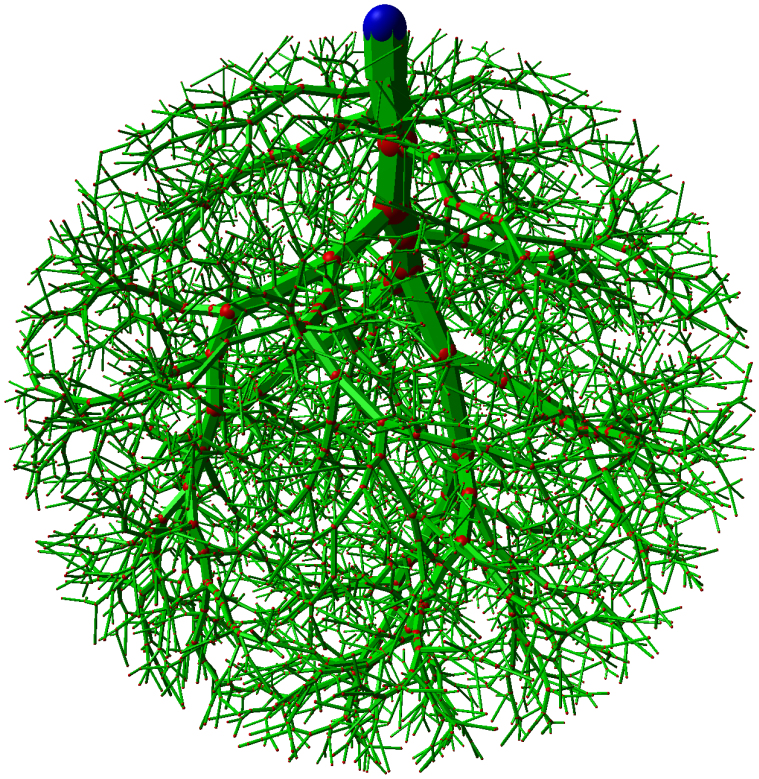
A quantitative assessment of the diameters of the segments depending on the bifurcation level in a non-open implementation proposed in [4], and our implementation is given in Figure 8.

Finally, an analysis of the computation cost of our 2-dimensional and 3-dimensional implementation of CCO is illustrated w.r.t implicit (i.e. spherical) / masked (i.e. arbitrary) domains and vs. VascuSynth [3], in Figure 9. These measurements were performed on a processor Apple M1 Max.

All these experimental results argue in favour of the homogeneity and the correctness of the behaviour of our implementation with the native algorithms proposed in [7] and [17].

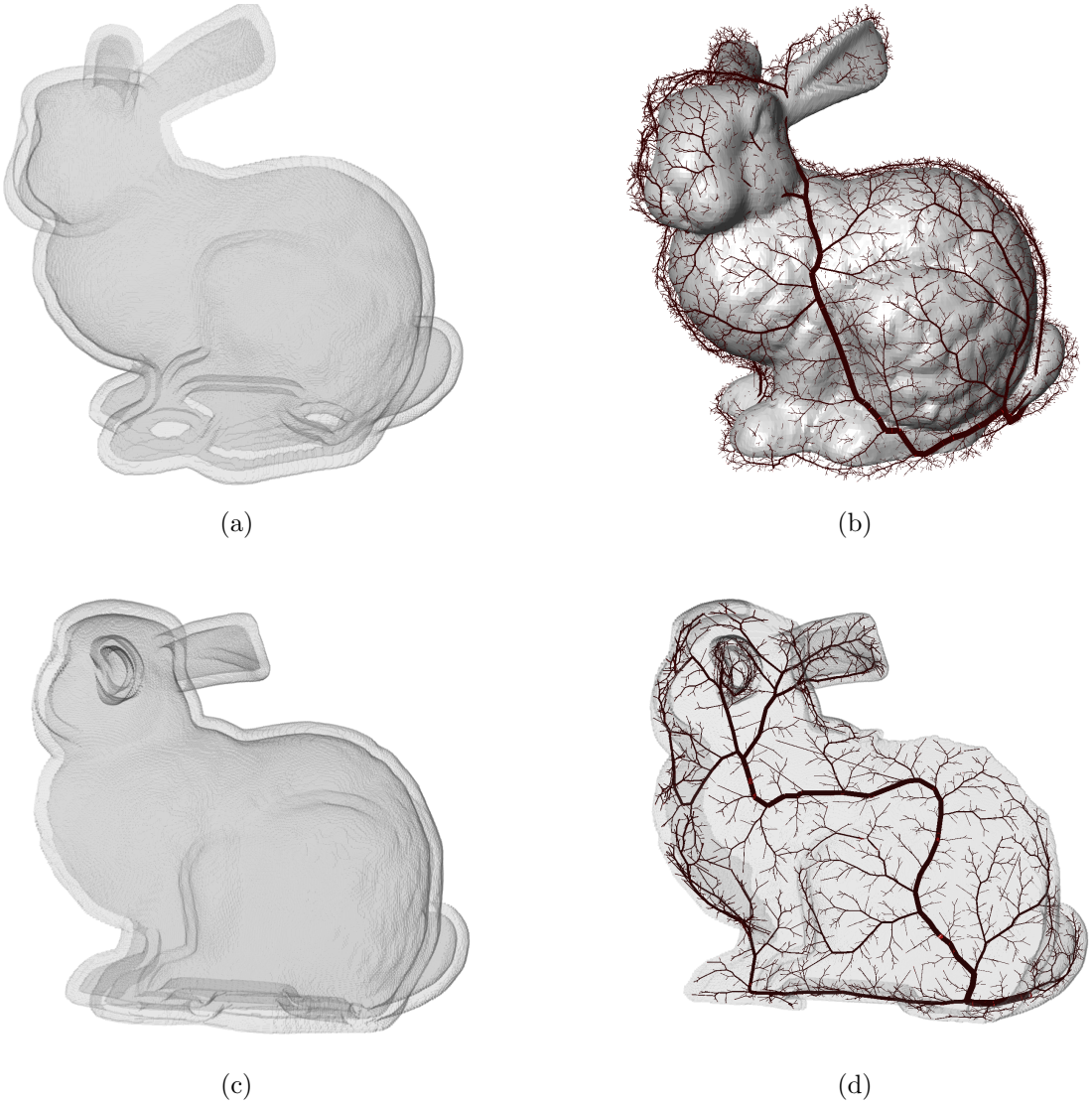


(a)



(b)

Figure 5: 3-dimensional networks generated in a sphere domain. The number of terminal points is set to 1000 (a) and 4000 (b). These results can be compared to those depicted in [7, Figure 5, p. 31].



(a) (b)

(c) (d)

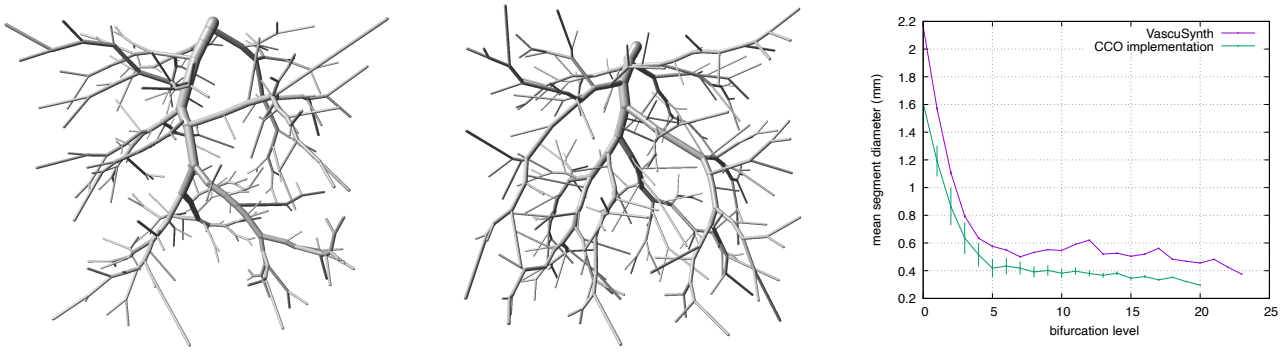
Figure 6: 3-dimensional network generated in an arbitrary shape domain. Here, the domain (a) is composed by the non-convex thick boundary of the Stanford bunny (a topological hollow sphere with holes). The domain (c) is defined by the same construction but by cutting the front mesh to increase visibility. The number of terminal points in the network is set to 20000 in (b) and 3000 in (d).

8 Conclusion and perspectives

In this article we proposed an implementation of the CCO algorithm, initially designed in [17] for the 2-dimensional case and in [7] for the 3-dimensional case. We did our best to provide an implementation that is as close as possible to these original algorithms.

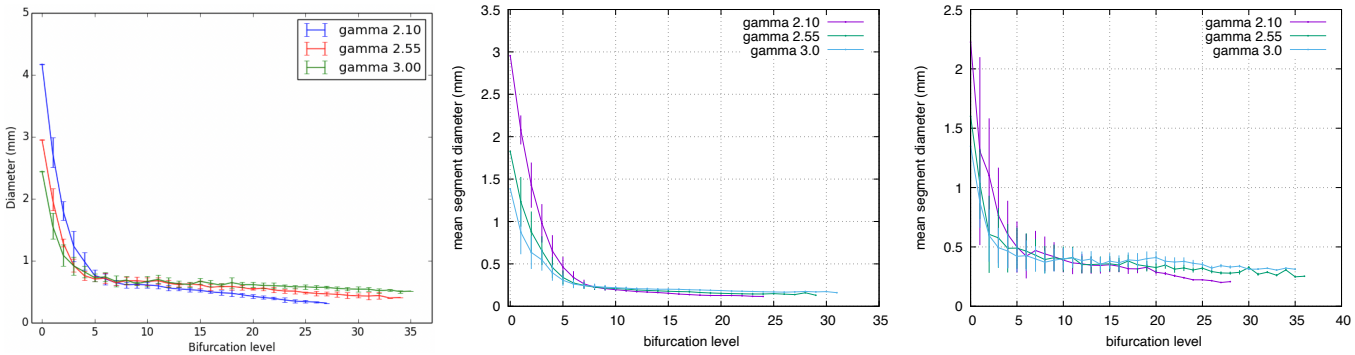
We believe that our implementation could be of precious use for any researcher interested in obtaining realistic vascular networks (or similar tree-based objects). In particular, it may constitute a useful alternative to existing softwares, such as VascuSynth [3].

It could be possible to build upon this framework in order to propose complementary functionalities that could tackle some recurrent issues that occur in vascular modeling. First, the approach could be extended to generate complementary networks (e.g. arterial and venous trees), which would require the handling of distinct, independent trees in a forest structure, whereas considering the constraints that each tree imposes to the other(s). Another important perspective would concern the ability to design a partial network composed of many branches that constitute the extension of end-



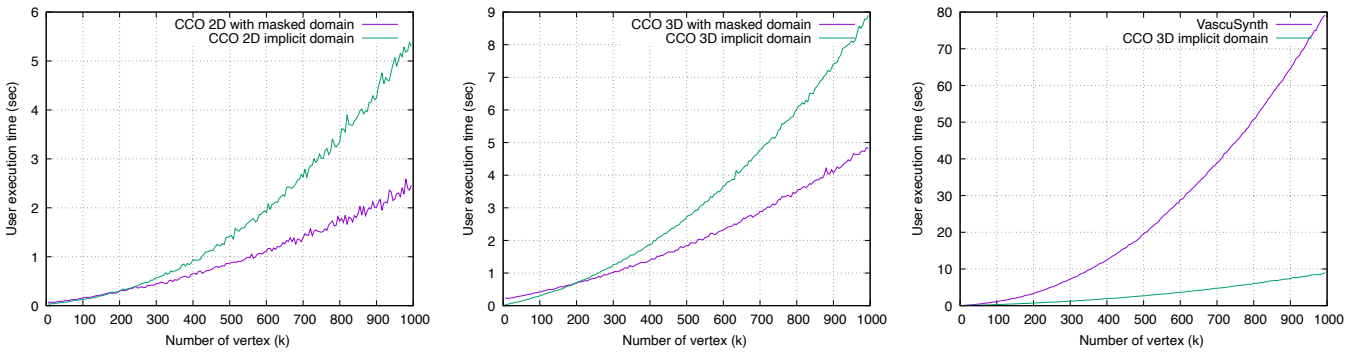
(a) $k = 200$, $\gamma = 3$, $t = 5,61$ s (b) $k = 200$, $\gamma = 3$, $t = 0,24$ s (c)

Figure 7: 3-dimensional networks generated in a cubic domain: VascuSynth (a) and our implementation of CCO (b). (c) Comparison of the segment diameters variation according to the bifurcation level between the VascuSynth reconstruction and our implementation of CCO.



(a) (b) (c)

Figure 8: Averaged segment diameters at respective bifurcation levels. (a) Results from [4]: *"Radius study along bifurcation levels: mean and standard deviation at each bifurcation level on 10 simulated trees"* (text from [4]). (b–c) Results from our implementation in 3-dimensions (b) and 2-dimensions (c). These results can also be compared to those depicted in [17, Figure 4, p. 487]. The behaviours of the curves are comparable, up to the different parameters which may induce distinct quantitative values.



(a) CCO 2D (b) CCO 3D (c) CCO 3D

Figure 9: Comparison of computation time: (a) CCO in 2D and (b) 3D with implicit and masked domains, (c) CCO vs. VascuSynth in 3D with implicit domain.

ing points of the vascular structures at a given scale. Such approach would allow to build some vascular models obtained from segmentation at the lowest scales (e.g. from patient-specific analysis

of medical images), and from modeling at the highest scales, where the images do no longer provide useful information, but where physiological behaviour from realistic models could be statistically inferred. Such optimizations could find inspiration in recent works dedicated to extend / build upon CCO or similar algorithms, e.g. [12, 8].

In the current version of the algorithm, the output flow is the same for each terminal point of the tree, whereas a uniform probability is considered to determine the putative position of these ending points. More versatile strategies could be developed to allow non-homogeneous output flows and biases in the positioning of the ending points, e.g. for better modeling phenomena such as angiogenesis.

From a more technical point of view, some octree-like hierarchical representations of the tree and of the vascular domain may also be considered for optimizing the computational burden induced by the assessment of the geometrical hypotheses related to the tree-tree and tree-domain constraints.

Acknowledgements

This work was supported by the French *Agence Nationale de la Recherche* (grants ANR-18-CE45-0018, ANR-18-CE45-0014, ANR-20-CE45-0011).

References

[1] S. Agarwal, K. Mierle, and The Ceres Solver Team, *Ceres Solver*, 3 2022.

[2] A. Fortin, S. Salmon, J. Baruthio, M. Delbany, and E. Durand, *Flow MRI simulation in complex 3D geometries: Application to the cerebral venous network*, Magnetic Resonance in Medicine, 80 (2018), pp. 1655–1665.

[3] G. Hamarneh and P. Jassi, *VascuSynth: Simulating vascular trees for generating volumetric image data with ground-truth segmentation and tree analysis*, Computerized Medical Imaging and Graphics, 34 (2010), pp. 605–616.

[4] C. Jaquet, *Vers la simulation de perfusion du myocarde à partir d'image tomographique scanner. (Toward simulation of myocardial perfusion based on a single CTA scan)*, PhD thesis, University of Paris-Est, France, 2018.

[5] C. Jaquet, L. Najman, H. Talbot, L. J. Grady, M. Schaap, B. Spain, H. J. Kim, I. E. Vignon-Clementel, and C. A. Taylor, *Generation of patient-specific cardiac vascular networks: A hybrid image-based and synthetic geometric model*, IEEE Transactions on Biomedical Engineering, 66 (2019), pp. 946–955.

[6] A. Kamiya and T. Togawa, *Optimal branching structure of the vascular tree.*, The Bulletin of Mathematical Biophysics, 34 (1972), pp. 431–438.

[7] R. Karch, F. Neumann, M. Neumann, and W. Schreiner, *A three-dimensional model for arterial tree representation, generated by constrained constructive optimization*, Computers in Biology and Medicine, 29 (1999), pp. 19–38.

[8] H. J. Kim, H. C. Rundfeldt, I. Lee, and S. Lee, *Tissue-growth-based synthetic tree generation and perfusion simulation*, Biomechanics and Modeling in Mechanobiology, (In press).

[9] J. Lamy, O. Merveille, B. Kerautret, and Passat. N, *A benchmark framework for multiregion analysis of vesselness filters*, IEEE Transactions on Medical Imaging, 41 (2022), pp. 3649–3662.

[10] D. Lesage, E. D. Angelini, I. Bloch, and G. Funka-Lea, *A review of 3D vessel lumen segmentation techniques: Models, features and extraction schemes*, Medical Image Analysis, 13 (2009), pp. 819–845.

[11] V. J. Lumelsky, *On fast computation of distance between line segments*, Information Processing Letters, 21 (1985), pp. 55–61.

[12] G. D. Maso Talou, S. Safaei, P. J. Hunter, and P. J. Blanco, *Adaptive constrained constructive optimisation for complex vascularisation processes*, Scientific Reports, 11 (2021), p. 6180.

[13] O. Miraucourt, S. Salmon, M. Szopos, and M. Thiriet, *Blood flow in the cerebral venous system: Modeling and simulation*, Computer Methods in Biomechanics and Biomedical Engineering, 20 (2017), pp. 471–482.

[14] S. Moccia, E. De Momi, S. El Hadji, and L. S. Mattos, *Blood vessel segmentation algorithms — Review of methods, datasets and evaluation metrics*, Computer Methods and Programs in Biomedicine, 158 (2018), pp. 71–91.

[15] C. D. Murray, *The physiological principle of minimum work: I. The vascular system and the cost of blood volume.*, Proceedings of the National Academy of Sciences, 12 (1926), pp. 207–214.

[16] M. Schneider, J. Reichold, B. Weber, G. Székely, and S. Hirsch, *Tissue metabolism driven arterial tree generation*, Medical Image Analysis, 16 (2012), pp. 1397–1414.

[17] W. Schreiner and P.F. Buxbaum, *Computer-optimization of vascular trees*, IEEE Transactions on Biomedical Engineering, 40 (1993), pp. 482–491.