Solving General Auxin Transport Models with a Numerical Continuation Toolbox in Python: PyNCT

Delphine Draelants(✉), Przemysław Kłosiewicz, Jan Broeckhove, and Wim Vanroose

Department of Mathematics and Computer Science, Universiteit Antwerpen, Middelheimlaan 1, 2020 Antwerpen, Belgium
{Delphine.Draelants,Przemyslaw.Klosiewicz,Jan.Broeckhove, Wim.Vanroose}@uantwerpen.be
http://www.uantwerpen.be/applied-mathematics

Abstract. Many biological processes are described with coupled non-linear systems of ordinary differential equations that contain a plethora of parameters. The goal is to understand these systems and to predict the effect of different influences. This asks for a dynamical systems approach where numerical continuation methods and bifurcation analysis are used to detect the solutions and their stability as a function of the parameters. We developed PyNCT – Python Numerical Continuation Toolbox – an open source Python package that implements numerical continuation methods and can perform bifurcation analysis based on sparse linear algebra. The software gives the user the choice of different solvers (direct and iterative) and allows the use of preconditioners to reduce the number of iterations and guarantee the convergence when working with complex non-linear models.

In this paper we demonstrate the usefulness of the toolbox with a class of models pertaining to auxin transport between cells in plant organs.We show how easy it is to compute the steady state solutions for different parameter values, to calculate how they depend on each other and to map parts of the solution landscape.

An interactive model development and discovery cycle is key in bio-systems research. It allows one to investigate and compare different model parameter settings and even different models and gauge the model's usefulness. Our toolbox allows for such quick experimentation and has a low entry barrier for non-technical users.

Although PyNCT was developed particularly for the study of transport models in biology, its implementation is generic and extensible, and can be used in many other dynamical system applications.

Keywords: Continuation methods · Bifurcation analysis · Transport models · PyNCT

1 Introduction

In recent years, many molecular-genetic experiments have been performed to increase our insights in how molecular processes in plants work. Due to the restricted set of experiments that can be performed today, the amount of available experimental data is still limited. For this reason researchers try to expand their knowledge of biological processes by means of an interaction between experimental research and mathematical modelling.

Formerly, biologists used the available data to mathematically describe the biological processes (of plants). The incredible amount of uncertainties about how these systems work gave birth to different hypotheses and, as a result, different kinds of models with lots of parameters. These models were initially solved with simple numerical methods for a limited set of parameters.

Today the biological models become very large and complex. Moreover biologists are interested in understanding the whole solution landscape instead of one single particular solution. They want to understand the effect of different influences (parameters) and compare current models based on different hypotheses. In order to solve, analyse and compare these models, state-of-the-art numerical methods are necessary. Unfortunately, these methods, solvers and algorithms are often not easily accessible outside of their application niche. This makes it difficult for systems biologists to directly apply state-of-the-art numerical mathematics to their specific problems. The end result is a growing demand for software packages that combine biological models with numerical tools.

We developed a toolbox, PyNCT, that solves and analyses the non-linear coupled systems of equations that appear in a wide range of models for the transport of chemicals through networks of cells. PyNCT contains numerical continuation methods and can perform bifurcation analysis in order to find parts of the solution landscape as a function of the different parameters. The toolbox is based on sparse linear algebra which enables its users to solve very large systems. With the resulting simulation tools biologists can now explore, analyse and compare various models, test new hypotheses, ...without the need to understand the inner details of mathematics behind the numerical methods. Although we developed PyNCT specifically to study transport models, it works well for all dynamical systems.

The paper is organized as follows. First, in Sect. 2 we give more information about the application domain. We describe a tissue of cells mathematically, we present a general class of transport models already implemented in PyNCT, and we discuss the type of solutions biologists are interested in. We also discuss the state-of-the-art tools that are currently available. Then, in Sect. 3 we present the numerical algorithms in PyNCT in detail. Readers who are not interested in the mathematical details can skip this section. A motivation for the choice of Python and certain libraries as a basis for the implementation of the toolbox can be found in Sect. 4. In Sect. 4.2 we explain how to apply the software for the different types of models and in Sect. 5 we demonstrate it for a specific example, the model of Smith et al. [18]. Finally in Sect. 6 we conclude and give an outlook.

2 The Application Domain

The PyNCT toolbox has been developed to investigate the response of stationary solutions of (large) dynamical systems to changes of control parameters. It can be used for any model consisting of a system of non-linear equations that are smooth and continuously differentiable.

In this paper we show the usefulness of our toolbox, using the auxin transport equations of a cell-based plant organ model as a demonstration vehicle. In Sect. 2.1 we present the mathematical description of a plant tissue of irregular cells. Based on this we describe a class of concentration-based auxin transport equations and look at the typical solutions that are of interest. PyNCT is able to generate solution branches automatically for any model that fits this framework. In Sect. 5 we demonstrate how to do so for a specific auxin transport model described by Smith et al. [18]. At the end of this section we discuss the current state-of the-art tools for this and compare them with our tool.

2.1 Auxin Transport Models

Network of Cells. In biology, cell tissues are represented by a graph with edges and vertices. The edges represent the cell walls of the plant organ and a cell is then a face in this graph. As a consequence a cell is a vertex in the weak dual graph G. See Fig. 1 for an example. This dual graph helps us to describe the tissue mathematically:

- The set of vertices V represents all cells in the tissue and we identify them with an index $i \in \{1, ..., n\}$.
- The set of edges E represents the connections between neighbouring cells. As a consequence the neighbouring cells of a cell i can be identified as all cells up to distance 1 from cell i. This subset of cells is denoted with $\mathcal{N}_i \subset V$.
- Every edge represents the connection between two neighbouring cells and thus we can uniquely associate the information about a cell wall with an edge. By labelling each edge with relevant information about the cell wall (for example the permeability of the cell wall), we get a weighted graph G.
- In every cell we can define various properties of the cell, such as the concentration of a specific hormone, protein,.... These are the variables of the models. We denote m as the number of the state variables per cell.

With the help of this representation of a tissue we can now describe easily how substances in cells are transported throughout the tissue with a system of equations.

The System of Equations. The toolbox contains software to easily calculate solutions for transport models that can be written as follows

$$\dot{\mathbf{y}}_i = \boldsymbol{\pi}(\mathbf{y}_i) - \boldsymbol{\delta}(\mathbf{y}_i) + \boldsymbol{D} \sum_{j \in \mathcal{N}_i} (\mathbf{y}_j - \mathbf{y}_i) + \boldsymbol{T} \sum_{j \in \mathcal{N}_i} (\boldsymbol{\nu}_{ji}(\mathbf{y}_1, ..., \mathbf{y}_n) - \boldsymbol{\nu}_{ij}(\mathbf{y}_1, ..., \mathbf{y}_n)) . \quad (1)$$

The vector $\mathbf{y}_i$ contains the m time-dependent state variables in cell i. For instance, $\mathbf{y}_i$ may contain the auxin concentration ($m = 1$) or both auxin and PIN-FORMED1 concentrations ($m = 2$) in cell i. Further, the model consists of

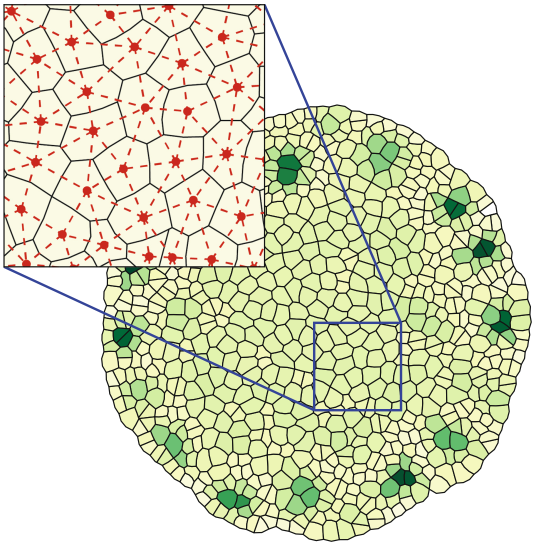

Fig. 1. The large picture shows a typical tissue of cells. The zoomed-in portion details the cell graph (full black) and its weak dual graph G of cellular connections (dashed red) (Color figure online).

$\boldsymbol{\pi}, \boldsymbol{\delta}$, the production and decay functions, respectively, $\boldsymbol{D}$, a matrix with diffusion coefficients, T a matrix with the active transport parameters and $\boldsymbol{\nu}_{ij}$ the active transport functions. In our example, we assume active transport functions that can be expressed as

$$(\boldsymbol{\nu}_{ij})_l\,(\mathbf{y}_1,...,\mathbf{y}_n) = \psi_l(\mathbf{y}_i,\mathbf{y}_j)\frac{\varphi_l(\mathbf{y}_j)}{\sum_{k\in\mathcal{N}_i}\varphi_l(\mathbf{y}_k)}, \quad \text{for } l = 1,...,m, \tag{2}$$

where the functions ψ_l, φ_l depend on the model choices. Many concentration-based auxin transport models can be written in this form. More information about this class of models can be found in [7] and in Sect. 5 we demonstrate how the toolbox works for a specific example.

The Numerical Solutions. The models described above possess an inherent time-scale separation: the growth hormone dynamics involve short time scales (of the order of seconds) [4], while changes in cellular shapes and proliferation of new cells occur on much slower time scales (hours or days) [3]. In order to determine the distribution of auxin in the plant, it is sufficient to concentrate on the fast time scale of the hormone transport. Therefore we can assume a static cell structure and study the plant tissue as a dynamical system where we are only interested in the steady state solutions of the system and their dependence on the model parameters. The PyNCT toolbox is designed with sufficient functionality to calculate those steady state solutions immediately.

2.2 Current State-of-the-Art Tools

We can divide the current toolboxes that calculate these stationary solutions of dynamical systems in function of model parameters in two groups: tools based on dense linear algebra (limited to small tissues) and tools based on sparse linear algebra routines, which scale to large tissues.

The first group is the largest and well known tools like AUTO [6] and MATCONT [5] belong to this group. Also current new tools, especially developed for system biologist to investigate and analyse their new biological systems like Systems biology toolbox for Matlab [15] and Facile [17], heavily rely on dense linear algebra routines since they are all based on AUTO. The disadvantage of these tools is that the routines are not scalable to very large systems with many cells. The number of equations is typically limited to about 500.

An example of the second group is LOCA [14]. LOCA is developed around sparse linear algebra but it is designed for extremely large systems that need to be run on HPC infrastructure. LOCA is not easy to use and requires expert knowledge in C++ and HPC hardware.

Our toolbox is also designed to take advantage of sparse linear algebra but avoids C++ or HPC knowledge. The solutions of the typical large biological cell-based systems can be calculated very fast in contrast with existing system biology tools. Another big advantage of PyNCT compared to LOCA is the usability of the software. As explained in Sect. 4.2 in more detail, it is very easy to use the tool for a large class of transport models and even for many other models, only a routine with the equations must be provided.

3 Currently Available Functionalities

In this section we will explain briefly the main numerical algorithms implemented in PyNCT. Section 3.1 explains the numerical continuation methods. We also discuss the related functionalities available in PyNCT. Section 3.2 explains the principles and applications of bifurcation analysis.

If the reader is not interested in the mathematics under the hood of this toolbox and only wants to use it as a black box, he/she can skip Sect. 3.1 and jump to Sects. 4.2 and 5 where we explain how to use the toolbox.

3.1 Numerical Continuation Methods

The idea of continuation methods is to find a curve of approximate solutions $\mathbf{y}$ of a system of non-linear equations

$$F\left(\mathbf{y}, \boldsymbol{\lambda}\right) = \mathbf{0}, \tag{3}$$

as a function of the parameter vector $\boldsymbol{\lambda}$ with

$$F : \mathbb{R}^{v+w} \to \mathbb{R}^{v} : \left(\mathbf{y}, \boldsymbol{\lambda}\right) \mapsto F\left(\mathbf{y}, \boldsymbol{\lambda}\right). \tag{4}$$

Following the implicit function theorem we know that for a non-singular point $\left(\mathbf{y}^{(0)}, \boldsymbol{\lambda}^{(0)}\right)$ that satisfies $F\left(\mathbf{y}^{(0)}, \boldsymbol{\lambda}^{(0)}\right) = \mathbf{0}$, the solution set $F^{(-1)}(\mathbf{0})$ can be locally parametrized about $\left(\mathbf{y}^{(0)}, \boldsymbol{\lambda}^{(0)}\right)$ with respect to a parameter of $\boldsymbol{\lambda}$. This means that the system of equations $F(\mathbf{y}, \boldsymbol{\lambda}) = 0$ defines an implicit curve $\mathbf{y}(\boldsymbol{\lambda}(s))$ for any parametric curve $\boldsymbol{\lambda}(s) : \mathbb{R} \to \mathbb{R}^w$ in $\mathbb{R}^w$ [1]. To construct such a curve of subsequent solution points $\left(\mathbf{y}^{(i)}, \boldsymbol{\lambda}^{(i)}\right) = \left(\mathbf{y}^{(i)}, \boldsymbol{\lambda}^{(i)}(s)\right)$, continuation methods use a starting point $\left(\mathbf{y}^{(0)}, \boldsymbol{\lambda}^{(0)}\right)$, a solution of system (3), along with an initial continuation direction [12]. This starting point is typically a known trivial solution. An important family of continuation methods are predictor-corrector schemes. The idea of these algorithms is to predict a new solution point first. Then, in the corrector step, this predicted point is used as the initial guess for an iterative method that will converge to the solution up to a given tolerance. In our toolbox, the predictor step uses the secant method and a given step size to predict a guess for the next solution point on the curve. The corrector step improves the guess with Newton iterations.

Newton's Method. When applying the above continuation method, we improve the guess $\left(\tilde{\mathbf{y}}^{(i+1)}, \tilde{\boldsymbol{\lambda}}^{(i+1)}\right)$, found in the predictor step with Newton iterations [10]

$$\mathbf{y}^{(i+1)} = \tilde{\mathbf{y}}^{(i+1)} - \frac{F\left(\tilde{\mathbf{y}}^{(i+1)}, \tilde{\boldsymbol{\lambda}}^{(i+1)}\right)}{F'\left(\tilde{\mathbf{y}}^{(i+1)}, \tilde{\boldsymbol{\lambda}}^{(i+1)}\right)}, \tag{5}$$

until a sufficiently accurate new solution point $\left(\mathbf{y}^{(i+1)}, \boldsymbol{\lambda}^{(i+1)}\right)$ of F is reached.

In every iteration step, the system

$$J(\mathbf{y}, \boldsymbol{\lambda})\, \boldsymbol{x} = -F(\mathbf{y}, \boldsymbol{\lambda}) \tag{6}$$

is solved, with Jacobian matrix $J(\mathbf{y}, \boldsymbol{\lambda})$ defined by

$$J(\mathbf{y}, \boldsymbol{\lambda})_{ij} = \frac{\partial (F)_i}{\partial (\mathbf{y})_j} (\mathbf{y}, \boldsymbol{\lambda}) \tag{7}$$

By default in PyNCT we can use a direct or an iterative solver for (6).

- *Direct sparse linear solver:* The direct linear solver from SciPy (`scipy.sparse.linalg.spsolve`) provides excellent performance for moderately sized systems. At the time of this writing, the solver is a wrapper around either SuperLU or UMFPack; both mature and widely used sparse direct solver libraries [8].
- *Iterative solver:* If the size of the system requires the use of an iterative solver, PyNCT enables the use of Generalized Minimal RESidual (GMRES), a Krylov based solver, implemented in SciPy. We chose this method because it is very robust and applicable on all types of linear problems. This is necessary because continuation methods calculate both the stable and unstable solutions. In many cases the latter degrades or even destroys convergence of most iterative solvers.

Although we only suggest these two linear solvers in PyNCT, SciPy provides a typical array of iterative solvers based on Generalized Minimal RESidual (GMRES), Conjugate Gradient (CG), and derived methods. All these methods can be used easily. More information about such Krylov subspace solvers can be found in [9] and up-to-date information about the linear solvers available in SciPy can be found in the SciPy documentation pages[1].

Jacobian. A continuation method requires the Jacobian matrix J for the calculations of the Newton corrections. The Jacobian of cell-based biological systems with local interactions is a very large sparse matrix. Therefore it is important to exploit our knowledge about the structure of the Jacobian. By ordering the variables in the right way it can be divided in different building blocks where every block represents the derivative of an equation of the model to a variable representing a substance in each cell. For instance, consider a system of m transport equations for every cell, with n the number of cells and m the number of unknowns (the different substances in a cell) as described in Eq. (1). The Jacobian then consists of m^2 blocks of size $n \times n$ if the vector of unknowns is grouped per substance type. The example available in PyNCT uses this ordering, but it is possible to order the variables in any way. All these blocks have a sparse structure because in every equation the changes over time only depend on the variables in the cell itself and the neighbours up to distance 2.

In the PyNCT toolbox it is possible to choose between using the exact Jacobian or an approximation:

- The Jacobian is calculated exactly by determining the derivatives of the system with the use of SymPy, a Python library for symbolic mathematics [19].
- The approximated Jacobian is calculated numerically by using finite differences. The jth column of the Jacobian matrix is found by a forward difference scheme
$$J\left(\mathbf{y}^{(i)}, \boldsymbol{\lambda}^{(i)}\right)_j = \frac{F\left(\mathbf{y}^{(i)} + \epsilon \boldsymbol{e}_j, \boldsymbol{\lambda}^{(i)}\right) - F\left(\mathbf{y}^{(i)}, \boldsymbol{\lambda}^{(i)}\right)}{\epsilon}, \tag{8}$$
where $\boldsymbol{e}_j$ is the jth unit vector and $\left(\mathbf{y}^{(i)}, \boldsymbol{\lambda}^{(i)}\right)$ is the ith calculated solution point on the branch as before.

We chose for forward finite differences because it is a very easy algorithm and do not need many calculations per iteration. For instance the value of $F\left(\mathbf{y}^{(i)}, \boldsymbol{\lambda}^{(i)}\right)$ is already calculated and saved. The default value for ϵ is $\epsilon = 10^{-10}$ but the user can customize it if desired.

Preconditioning. When using iterative methods to solve each Newton step, we can use a preconditioner to reduce the number of iterations or to guarantee convergence of the iterative method when working with complex systems [9].

1 http://docs.scipy.org/doc/scipy/reference/sparse.linalg.html.

Instead of solving the original linear system $J(\mathbf{y}, \boldsymbol{\lambda})\,\boldsymbol{x} = -F(\mathbf{y}, \boldsymbol{\lambda})$ we solve the preconditioned system

$$P^{-1}J(\mathbf{y}, \boldsymbol{\lambda})\,\boldsymbol{x} = -P^{-1}F(\mathbf{y}, \boldsymbol{\lambda}), \tag{9}$$

which is a better conditioned problem, leading to faster convergence of the Jacobian solve. By choosing the right preconditioner, preconditioned iterative solvers perform better then direct solvers. For problems where the diffusion between the cells dominates traditional preconditioners that approximately invert the Poisson equation such as incomplete factorizations or multigrid can be effective. However, when the active transport dominates different preconditioners need to be developed. This is still an open topic of research.

In PyNCT it is possible to use a preconditioner. Since a good preconditioner asks specific knowledge about the model, we did not provide any general preconditioners but it can be specified by the user. How this should be implemented is explained in Sect. 4.2

3.2 Bifurcation Analysis

The study of the relation between the stability of a solution and the parameters of the corresponding dynamical system is known as (local) bifurcation analysis [16]. Such an analysis identifies the stable and unstable solutions and the bifurcation points that mark the transitions between them. This is biologically relevant since it will allow us to predict the patterns that emerge in the time evolution as the parameters of the model are changed. A bifurcation point is a solution $\left(\mathbf{y}^{(i)}, \boldsymbol{\lambda}^{(i)}\right)$ of system (3) where the number of solutions changes when $\boldsymbol{\lambda}$ passes $\boldsymbol{\lambda}^{(i)}$. For a complete review of the different types of bifurcation points and their properties we refer to [16]. The analysis usually leads to a bifurcation diagram that highlights the connections between stable and unstable branches as the parameters change. It is useful to track all these solution branches that emerge, split or end in a bifurcation point which can be done with the help of numerical continuation methods explained in Sect. 3.1.

Our toolbox contains methods to calculate the stability of a solution point directly after each point or after calculating the whole solution branch. For the transport models, we chose to calculate the stability of the solutions as part of the post-processing since even without the stability information, the continuation data can be very useful. A great advantage of this choice is that the continuation data is much faster to compute because calculating the eigenvalues for every solution point on a branch is very time-consuming.

To calculate the eigenvalues of the Jacobian, we use the `'eig'` routine in `scipy.linalg` based on dense linear algebra, although the Jacobian of transport models is a sparse matrix (see Sect. 3.1). Typically in transport models, around a bifurcation point, many eigenvalues cross the imaginary axis. As a consequence, the sparse routines of `scipy.sparse.linalg` for calculating eigenvalues fail to converge when searching for eigenvalues around zero. Although we are using dense linear algebra, calculating the stability for moderate system sizes can be performed in an acceptable time frame by parallelizing calculations with MPI (using `mpi4py`[2]).

Note that if interested in sparse routines, SciPy provides a sparse routine `scipy.sparse.linalg.eigs` that can be used easily in PyNCT.

After calculating the eigenvalues of the solution points, the bifurcation points must be indicated manually. It is then possible to start the continuation again from these bifurcation points in a new direction to find the branches that emerge. However, for now PyNCT does not contain methods for automatic branch switching.

4 Overview of Software Structure

4.1 Choice of Language and Libraries

Language. The PyNCT toolbox is implemented in Python. This choice is motivated by a number of factors:

- Python is a flexible language and is well-suited for rapid development. Adapting model code is straightforward and does not require an edit-compile-link cycle as does, for instance, C++.
- Python has a low entry barrier. It is easy to learn and to use and thus an ideal language for less technical users.
- Python has a large standard library with good documentation and a huge amount of contributed, community-maintained packages. PyNCT uses several existing libraries that include for example numerical methods so we don't have to 'reinvent the wheel'. More information about the packages included in PyNCT can be found below.
- Python is an open source programming language and also our software is freely available.

Numerical Libraries. The numerical part of our toolbox relies substantially on NumPy [2] and SciPy [8]. The former provides a foundation of linear algebra primitives in Python. The latter extends it by providing a huge variety of algorithms, solvers and support methods for "all things scientific" in Python. Both are high-quality, popular and well-documented open source libraries.

To enhance both speed and accuracy of the calculations we use symbolic expressions for the specification of the equations in the biological model and automatic differentiation to obtain the exact Jacobian expression. SymPy[19], a Python library for symbolic mathematics, is an excellent tool for these purposes in our case. The use of symbolic expressions, however, depends on the biological model under investigation and is not universally feasible for all applications

[2] https://pypi.python.org/pypi/mpi4py.

(Remark that as mentioned in Sect. 3.1 also the approximated Jacobian can be used if the user can't or don't want to use SymPy).

Other Libraries. The infrastructure for loading and storing virtual tissue representations and generating tissue geometries is provided by the Python Plant Tissue Simulation toolbox PyPTS [11]; an open source library. PyPTS uses a HDF5 based file format to store simulation results which makes pre/post-processing, visualisation and exchanging results with other tools such as VirtualLeaf [13] easy. It also provides an easy API for accessing and modifying tissue entities and attributes.

4.2 The Executable

When using the toolbox for a specific model, the system of equations must be specified.

For a specific class of transport models, the toolbox can be used by just providing the equations and parameters in configuration files (see Sect. 5 for an example).

For all other models a new class must be constructed. The class must contain an initialize method and a method that applies your system of equations. Additionally we also need a configuration file similar to the ones constructed for the transport models and explained in Sect. 5. It contains the parameter values of the model and the specifications of the numerical methods. At last an executable script, similar to the biology demo is necessary to start up the continuation. The PyNCT package already includes a basic template for this class, the executable and the configuration file which makes it very easy to start implementing your own model.

To extend this basic template, you can define an extra method that constructs the Jacobian in a given point. You can define an exact or an approximate Jacobian that differs from the standard approximation method described in Sect. 3.1. Then you can choose between the different Jacobian implementations to solve the Newton iterations. It is also possible to specify a preconditioner in this class to speed up the convergence to a solution point.

5 A Look at the Toolbox via an Example

In this section we show how easy it is to use the toolbox and find parts of the solution space of the model of Smith et al. [18]. The model satisfies Eqs. (1) and (2) and features 2 state variables per cell, namely the indole-3-acetic acid (IAA) concentration, $a_i(t)$, and the PIN-FORMED1 (PIN1) amount, $p_i(t)$. The model features IAA production, decay, active and passive transport terms, whereas for PIN1 only production and decay are included. This results in the following set of coupled non-linear ordinary differential equations (ODEs)

$$\begin{aligned}\frac{da_i}{dt} =& \frac{\rho_{\text{IAA}}}{1+\kappa_{\text{IAA}} a_i} - \mu_{\text{IAA}} a_i + \frac{D}{V_i} \sum_{j \in \mathcal{N}_i} l_{ij} \left(a_j - a_i\right) \\ &+ \frac{T}{V_i} \sum_{j \in \mathcal{N}_i} \left[P_{ji}(\boldsymbol{a}, \boldsymbol{p}) \frac{a_j^2}{1+\kappa_T a_i^2} - P_{ij}(\boldsymbol{a}, \boldsymbol{p}) \frac{a_i^2}{1+\kappa_T a_j^2} \right], \end{aligned} \tag{10}$$

$$\frac{dp_i}{dt} = \frac{\rho_{\text{PIN}_0} + \rho_{\text{PIN}} a_i}{1+\kappa_{\text{PIN}} p_i} - \mu_{\text{PIN}} p_i, \tag{11}$$

for $i = 1, ..., n$ with n the number of cells. In this model D is a diffusion coefficient, V_i is the cellular volume, $l_{ij} = S_{ij}/(W_i + W_j)$ is the ratio between the contact area S_{ij} of the adjacent cells i and j, and the sum of the corresponding cellular wall thicknesses W_i and W_j. In addition, T is the active transport coefficient and P_{ij} is the number of PIN1 proteins on the cellular membrane of cell i facing cell j,

$$P_{ij}(\boldsymbol{a}, \boldsymbol{p}) = p_i \frac{l_{ij} \exp\left(c_1 a_j\right)}{\sum_{k \in \mathcal{N}_i} l_{ik} \exp\left(c_1 a_k\right)}. \tag{12}$$

More details on the model and the parameters can be found in [18].

The rest of the section is divided in three parts, the preparation, the actual calculations and the post-processing. In these sections we explain step by step how to find the steady state solutions starting from the above model.

5.1 Preparing for Continuation

Before we can calculate the solutions, we need to specify the model and choose from several solution methods implemented in PyNCT. Therefore we fill in a model file and a parameter file respectively.

The Model File. In the model file each part of the system (production, decay, diffusion, ...) is listed. For example for the model of Smith et al. this file becomes

```
{
  "decayPIN": "muPIN*p",
  "productionPIN": "(rhoPIN0 + rhoPIN * a) / (1.0 + kPIN * p)",
  "decayIAA": "muIAA*a",
  "productionIAA": "rhoIAA / (1.0 + kIAA * a)",
  "passive_transport": "D * wall_length * (a_j - a_i)",
  "phi": "wall_length*exp(c1*a_j)",
  "psi": "p*a_i**2/(1.0+kT*a_j**2)"
}
```

The Parameter File. In the parameter file we specify all parameters that are necessary to perform the continuation. This includes the model parameters, information about the tissue, the solvers, the continuation and the saving process.

In this example, we consider a tissue with 742 irregular prismic cells that cover an almost-circular domain (geometry extracted from [13]) with free boundary conditions [7]. We choose as continuation parameter the model parameter T, and the trivial solution of this model in $T = 0$ as the starting point. We also specify a directory and file name to save the continuation data. A part of the parameter file reads

```
{
  "input": "./location/of/cells_742.h5",
  "output": "./location/of/continuation.h5",
  "rhoIAA": 1.500,
  "D": 1.000,

  ...

  "T": 0.0,
  "startpoint": "value",
  "startpoint_a": "(-1.0 + sqrt(1.0 + 4.0*kIAA*rhoIAA/muIAA))/(2.0*kIAA)",
  "startpoint_p": "(-1.0 + sqrt(1.0 + 4.0*kPIN*(rhoPIN0 + rhoPIN*a) /muPIN
      ))/(2.0*kPIN)",

  ...
}
```

More information and examples of both the model file and the parameter file, can be found in the demos directory of the PyNCT toolbox.

5.2 Executing the Continuation

After specifying all parameters in the correct files, we can start the continuation by calling the '`doContinuation`' method.

```
from pynct.biology import doContinuation
doContinuation.doContinuation('/location/of/parameterFile.json',
                              '/location/of/modelFile.json')
```

Every solution point is saved immediately after it is calculated in the specified output file. This method has the advantage that we can already start with the post processing before all points are calculated.

5.3 Post-processing

In the PyNCT biology demo, we already included two functionalities necessary for post-processing the calculated data. We can determine the stability properties of the solutions and we can visualize the solutions.

The Stability. The stability of the solutions on a continuation branch is determined with the function calculateEigenvaluesMpi in PyNCT.

```
from pynct.biology import calculateEigenvaluesMpi
calculateEigenvaluesMpi.main()
```

This function calculates the eigenvalues and saves them in a specified file. More information can be found in Sect. 3.2.

Plotting Tools. In order to process and interpret the calculated data in plant biology, it is very useful to visualize it. Although many tools already exist for plotting data, we added a number of basic functions in the PyNCT demo specifically aimed at visualizing continuation data from biological systems.

All plotting tools are implemented in the file `'plottools.py'` which therefore needs to be imported. We can plot bifurcation diagrams with or without the stability of the calculated solutions and all the solution patterns. The functions work by just specifying the right data files. For example, the following code gives an interactive plot with the bifurcation diagram and a corresponding solution pattern.

```
from pynct.biology import plottools
plottools.bifDiagramInteractive('/location/of/continuation.h5')
```

We can change the highlighted solution point and thus the solution pattern interactively. Figure 2 displays such a plot where also the stability propertiees are shown.

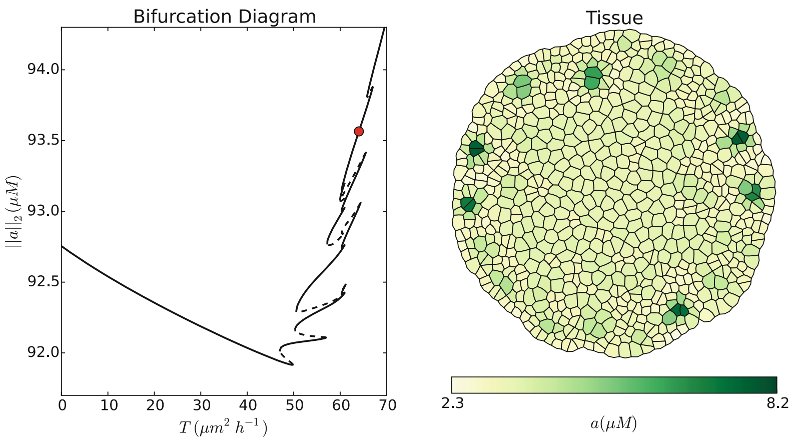

Fig. 2. Bifurcation diagram and corresponding solution pattern for the Smith et al. model for an almost-circular domain of 742 irregular cells (geometry taken from [13]). Left: An example of a bifurcation diagram that depicts the 2-norm of auxin concentration versus the continuation parameter T. Stable solutions are drawn with a full line and unstable solutions are dashed. Right: The solution pattern corresponding with the red dot on the left figure (Color figure online).

The bifurcation diagram shows the norm over all cells of IAA versus the continuation parameter T. As in the figure, we can highlight one solution point on the continuation branch. The distribution of IAA in the tissue in this solution point is then automatically displayed at the right in the figure. The darker the cell is coloured, the higher the concentration of the unknown (IAA).

More information about the plotting tools and how to use them can be found in the PyNCT biology demo.

6 Conclusion and Future Directions

We presented the Python Numerical Continuation Toolbox PyNCT, an open source library. The toolbox contains different state-of-the-art numerical algorithms for numerical continuation and is able to calculate the stability properties of the solutions.

The methods can be applied on coupled non-linear (smooth and continuously differentiable) equations and specifically on models describing the transport throughout a network of cells. For general models the system must be implemented in a new class but for a subset of concentration-based transport models (those models that satisfy Eqs. (1) and (2)) only a specification of the model parts in a configuration file is necessary. In the future we want to extend the class of models that can be analysed automatically.

The numerical methods implemented in PyNCT are based on sparse linear algebra since biological processes can be described often by just describing what happens in the direct neighbourhood. Therefore solutions can be calculated efficiently. Further there is no limitation on the number of unknowns or the size and shape of the tissue. These are the main advantages and differences of our toolbox in comparison with existing tools for system biologists.

PyNCT helps us to explore parts of the solution space. We can calculate a branch of solutions and determine the stability of each solution. Based on this information, we identify bifurcation points and calculate new branches that emerge. However, in the future we will include a method that detects the bifurcation points and performs automatic branch-switching.

Finally our toolbox allows quick experimentation and has a low entry barrier for less technical users. Biologists can now compare various transport models and explore different hypotheses very easily.

Acknowledgements. DD acknowledges financial support from the Department of Mathematics and Computer Science of the University of Antwerp. This work is part of the Geconcerteerde Onderzoeksactie (G.O.A.) research grant “A System Biology Approach of Leaf Morphogenesis” granted by the research council of the University of Antwerp. We acknowledge Giovanni Samaey for sharing basic version of a continuation code.

References

1. Allgower, E., Georg, K.: Numerical Path Following. Springer, Berlin (1994)
2. Ascher, D.: Numpy, Numerical Python (2001). http://www.numpy.org/. Accessed June 2015
3. Beemster, G., Baskin, T.: Analysis of cell division and elongation underlying the developmental acceleration of root growth in arabidopsis thaliana. Plant Physiol. **116**, 515–526 (1998)
4. Brunoud, G., Wells, D., Oliva, M., Larrieu, A., Mirabet, V., Burrow, A., Beeckman, T., Kepinski, S., Traas, J., Bennett, M., et al.: A novel sensor to map auxin response and distribution at high spatio-temporal resolution. Nature **482**, 103–106 (2012)

5. Dhooge, A., Govaerts, W., Kuznetsov, Y.A.: Matcont: a matlab package for numerical bifurcation analysis of odes. ACM Trans. Math. Softw. (TOMS) **29**(2), 141–164 (2003)
6. Doedel, E.J.: Auto: A program for the automatic bifurcation analysis of autonomous systems. Congr. Numer. **30**, 265–284 (1981)
7. Draelants, D., Avitabile, D., Vanroose, W.: Localized auxin peaks in concentration-based transport models of the shoot apical meristem. J. R. Soc. Interface 12(106) (2015). http://rsif.royalsocietypublishing.org/content/12/106/20141407.full
8. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: open source scientific tools for Python (2001). http://www.scipy.org/. Accessed June 2015
9. Kelley, C.T.: Iterative methods for linear and nonlinear equations. SIAM: Frontiers in Applied Mathematics 16 (1995)
10. Kelley, C.T.: Solving Nonlinear Equations with Newton's Method, vol. 1. Society for Industrial and Applied Mathematics, Philadelphia (2003)
11. Kłosiewicz, P.: Pypts, python plant tissue simulations (2015). https://pypi.python.org/pypi/PyPTS/. Accessed June 2015
12. Krauskopf, B., Osinga, H., Galán-Vioque, J.: Numerical Continuation methods for Dynamical Systems. Springer, Netherlands (2007)
13. Merks, R., Guravage, M., Inzé, D., Beemster, G.: Virtualleaf: an open-source framework for cell-based modeling of plant tissue growth and development. Plant Physiol. **155**(2), 656–666 (2011)
14. Salinger, A.G., Bou-Rabee, N.M., Pawlowski, R.P., Wilkes, E.D., Burroughs, E.A., Lehoucq, R.B., Romero, L.A.: Loca 1.0 library of continuation algorithms: theory and implementation manual. Sandia National Laboratories, Albuquerque, NM, Technical Report No. SAND2002-0396 (2002)
15. Schmidt, H., Jirstrand, M.: Systems biology toolbox for matlab: A computational platform for reasearch in systems biology. Bioinformatics Advance Access (2005)
16. Seydel, R.: Practical Bifurcation and Stability Analysis: From Equilibrium to Chaos, vol. 5. Springer, New York (1994)
17. Siso-Nadal, F., Ollivier, J.F., Swain, P.S.: Facile: a command-line network compiler for systems biology. BMC Syst. Biol. **1**(1), 36 (2007)
18. Smith, R., Guyomarc'h, S., Mandel, T., Reinhardt, D., Kuhlemeier, C., Prusinkiewicz, P.: A plausible model of phyllotaxis. PNAS **103**(5), 1301–1306 (2006)
19. SymPy Development Team: SymPy: Python library for symbolic mathematics (2014). http://www.sympy.org