Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

T. DIERKES, M. WADE, U. NOWAK, S. RÖBLITZ

BioPARKIN

—

Biology-related Parameter Identification in Large Kinetic Networks

ZIB-Report 11-15 (December 2011)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

BioPARKIN

—

Biology-related Parameter Identification in Large Kinetic Networks

T. Dierkes M. Wade U. Nowak S. Röblitz*

19th December 2011

Abstract

Modelling, parameter identification, and simulation play an important rôle in systems biology. In recent years, various software packages have been established for scientific use in both licencing types, open source as well as commercial. Many of these codes are based on inefficient and mathematically outdated algorithms. By introducing the package BioPARKIN recently developed at ZIB, we want to improve this situation significantly. The development of the software BioPARKIN involves long standing mathematical ideas that, however, have not yet entered the field of systems biology, as well as new ideas and tools that are particularly important for the analysis of the dynamics of biological networks. BioPARKIN originates from the package PARKIN, written by P.Deuflhard and U.Nowak, that has been applied successfully for parameter identification in chemical physics for many years. This report is addressed to scientists who want to get to know the mathematical background of BioPARKIN, and its actual implementation.

Keywords: systems biology, ordinary differential equations, sensitivity analysis, parameter identification, affine invariant Gauss-Newton algorithm, numerical library, graphical user interface

Contents

*Corresponding author

Introduction

BioPARKIN is a software package to aid researchers in the field of systems biology. It facilitates the simulation of biology-related models, and is specialised in the computation of parameter sensitivities and the identification of parameter values (based on experimental data). Its numerical core is built upon long-standing mathematical ideas that have not yet been applied in systems biology as well as new ideas that are particularly useful for analysing dynamical biological networks. BioPARKIN is available free of charge as an open-source project. It can be used in academic as well as commercial settings.

Systems biology aims at describing biological processes using mathematical models that permit biologically sound quantitative predictions. To arrive at statements of this kind, the time courses of biological processes are modelled by differential equations that describe the concentration or amount of the involved substances over time. For the time being, we consider only systems of ordinary differential equations (ODE systems). In BioPARKIN, the ODE systems are solved numerically by making use of LIMEX [8], a linearly implicit extrapolation method that is especially suited for stiff systems.

From a mathematical point of view the main difficulty is not to simulate such systems, i.e. to solve the differential equations, but to determine the unknown parameters in such a way that the simulation results agree with experimental measurement values. This is an inverse problem which can be formulated as a nonlinear least squares problem. In biological models, there are usually dependencies between parameters that typically lead to rank-deficient problems. For the solution of such problems, efficient and reliable numerical algorithms based on affine invariant Newton methods have been developed over the past decades [6]. An error-oriented algorithm with adaptive trust region and rank strategies has been implemented in the code NLSCON (Nonlinear Least Squares with CONstraints) [6]. A precursor version of NLSCON had already been part of the software package PARKIN [22, 7, 21], a single shooting method for parameter identification

in large chemical reaction kinetic networks. The two methods differ in the damping strategy of the Newton steps. NLSCON allows the user to control the algorithmic setting, whereas PARKIN works much more in an automated way, which might not always be suitable for the problem at hand. Both methods have been integrated into BioPARKIN and form the numerical core library of the software. Thus, BioPARKIN can be considered as a successor of PARKIN especially designed for applications in systems biology. In particular, we aimed at a comprehensive and flexible implementation using object-oriented design. In fact, BioPARKIN uses a stand-alone library written in C++, PARKINcpp, that deals most efficiently with the following numerical tasks:

- fast expression evaluation,
- solver LIMEX for stiff differential-algebraic systems,
- sensitivity analysis,
- Gauss-Newton methods NLSCON and PARKIN for parameter identification.

The graphical user interface of BioPARKIN is written in the highly flexible scripting language Python. The graphical and the numerical part work together using the wrapper library SWIG. In this way, BioPARKIN combines the best of both worlds: highly efficient C++ and very flexible Python.
The current release of BioPARKIN is not able to detect if a network of reactions is too large, i.e. if there are redundant reaction links that, eventually, could be eliminated without any change in the results. However, BioPARKIN is designed to indicate and to correctly deal with the situation where the parameter space contains a non-trivial subspace of solutions to the inverse problem at hand. Similarly, if a network is too small, i.e. if important compartments, species, and/or reactions are missing in order to explain the given data, this eventually shows up in poor convergence of the Gauss-Newton method during parameter identification.
This report is addressed to all scientists and researchers who want to learn more about the mathematical background of BioPARKIN, and its actual implementation. The outline of the report is as follows. Section 1 briefly introduces in the SBML standard used by already existing systems biology-related software. The mathematical problem is more formally presented in Section 2 while the iterative approach to its solution is put forward in Section 3. The next section, Section 4, describes some of the challenges in cross-platform development. State-of-the-art numerical routines are implemented in the C++ library PARKINcpp which is the focus of Section 5. This library is combined with a graphical user interface that allows intuitive handling of models including species, parameters, and mechanisms. The compatibility of BioPARKIN with the model standard SBML, an overview of the user interface and some details concerning software architecture and implementation are given in Section 6. Section 7 presents some results obtained using BioPARKIN. Finally, the report ends with a small outlook in Section 8 and a conclusion given in Section 9.

1 Existing Standards and Software

Having introduced the problem at hand and some aspects about our approach to solve it, the following parts will shed light on existing software, their advantages and their possible shortcomings. Before assessing existing software, however, the file format used by most software in systems biology is briefly explained.

1.1 SBML Standard

During the last decade, it became clear that well-defined standards would benefit the systems biology community. Independent from each other, several projects started to create a standard for modelling biological systems and sharing them with the community.
The most prominent projects are CellML [13, 14], BioPAX [5], and SBML [4]. Retrospectively, SBML is the most-widely used standard. While CellML, BioPAX, and other formats are still maintained and progressed, SBML is the de-facto standard—especially in simulation-centric workgroups and projects [25].

Regardless of which standard is used the researcher has to know the syntax and terminology of the format. SBML defines important entities like Species, Reactions, Compartments, Kinetic Laws, Rate Rules, Assignment Rules, Events, and others. This, of course, implies a steep learning curve to understand the format, its possibilities and its pitfalls. On the upside, almost all SBML-compatible tools share the same terminology. This makes it easier to use new tools that comply with SBML. It also facilitates the communication between researchers who might use different SBML tools.
For all these reasons, BioPARKIN supports SBML—currently in its most-wide used variant Level 2 Version 4 (see Section 6.1.1 for details).
The SBML standard is supported by a lively community of researchers and software engineers. There is an official software library, libSBML, to access SBML files and interact with them. This library is thoroughly maintained by a core team of developers since, obviously, the library is vital to almost all other software projects that comply with SBML.

1.2 Existing Software

Systems Biology Workbench. The Systems Biology Workbench (SBW) is both a set of small-to-medium sized SBML software tools and a system for SBML-centric tools to communicate with each other (via the so-called SBW Broker) [4].
Among others, the SBW contains tools for designing (JDesigner), simulating (Simulation Tool), creating layouts of (Network Layout), SBML models. Some tools exist only for Windows, others are available for nearly all platforms.
No tool within SBW was found that could serve as a suitable basis for a cross-platform solution like BioPARKIN.

COPASI. COPASI ("COmplex PAthway SImulator") is a general-purpose tool for modelling and simulation that has been published even before the SBML standard itself [15]. It uses its own file format but can import and export SBML files. Based on ideas from its predecessor GEPASI [20], the development of COPASI began in 2006 and is progressed by a multi-national team based in Virgina in the USA, Heidelberg in Germany, and Manchester in the UK.
COPASI is structurally well designed, has lots of features and a steep learning curve. Because of its flexibility, the user interface is not very streamlined. The feature list includes computing the sensitivities of parameters and the estimation of parameter values from experimental data.
While the software is suited for diverse applications, it is not easily extensible. There is no plugin architecture that allows the user to build on existing functionality. The source code of COPASI is available but may not be altered to create other software—it is allowed to use the source code within own projects but not to change it.
In general, the software is well documented. Detailed questions, however, often remain unanswered. Having access to the source code may seem to alleviate the problem. In practice, though, finding any particular code and following the program flow is too time-consuming to understand a feature—e.g. the details of computing relative sensitivities.

CellDesigner. CellDesigner is one of the most-widely used SBML tools to edit and display SBML models [10]. It can also be used to simulate models using the SOSlib library [18]. It is developed by several institutes in Japan and has progressed steadily over the past years. It adheres to the SBGN standard that defines the visual representation of SBML entities [17].
Although it supports the simulation of models it cannot compete with the plenty of features that, for example, COPASI offers. It is more suited to get a quick impression of a model's simulation results.
CellDesigner features a plugin interface. This feature is not widely adopted. Most of the existing plugins serve simple tasks and it is doubtful that the functionality of the BioPARKIN project could be implemented within such a plugin.

POEM. The development of BioPARKIN was inspired by its predecessor, an in-house tool called POEM ("Parameter Optimisation and Estimation Method") which was developed and maintained by U. Nowak with meticulous precision. POEM combines a numerical core written in FORTRAN, and a graphical user interface using Matlab.

Models have to be coded by hand in FORTRAN. Every change in the model requires the user to recompile the code into an executable file which, in turn, is used by the Matlab frontend. Thus, although POEM is sometimes really cumbersome to use, its rich feature list, and the high quality of its computational results, drove the development of BioPARKIN forward.
BioPARKIN tries to provide POEM's feature set in a more usable and accessible way.

2 Problem Description

2.1 Large Kinetic Networks in Systems Biology

A major topic in systems biology is the study of the dynamical evolution of bio-chemical mechanisms within a well-defined, biology-related context. The bio-chemical mechanisms in such a compound under consideration are typically given as a, possibly huge, set of chemical reactions between numerous species. Thus, the set of chemical reactions are the building blocks of the system model forming a large kinetic network. Assuming the general principle of mass action kinectics, this large network can readily be transformed to a system of n ordinary differential equations (ODEs) leading to the initial value problem (IVP)

$$y' = f(y\,;\,p), \qquad y(t_0) = y_0 \tag{2.1}$$

where the right-hand side, f, denotes the dependence of the change in the species vector, y', on both the species, $y \in \mathbb{R}^n$, and the parameter vector, $p \in \mathbb{R}^q$, of dimension q. The initial condition vector, y_0, has the same dimension as the species vector,y. In BioPARKIN, the ODE systems are solved numerically with LIMEX, a linear implicit Euler method with extrapolation [8, 9].
The first step in systems biology often involves developing a suitable model description, f, that will not be dealt with in detail here. Instead, it is assumed that some discrete experimental data (in form of species concentration versus time),

$$(\tau_1, z_1), \ldots, (\tau_M, z_M), \tag{2.2}$$

is available. Note that freqently, only a certain amount of the n species concentration are measurable observables, if at all. The task at hand now reduces to quantify the q unknown components of the parameter vector, p, by comparison between model values and measured data.
A complete data set, of course, must include prescribed statistical tolerances, $\delta z_j\,(j = 1, \ldots, M)$, for each measurement as well. The mathematically correct handling of these will be described in greater detail in Section 2.6 below.

2.2 Multiple Experiments

The design of experiments almost always includes different conditions such that the effects of these different conditions on the system under investigation can be observed and studied. In the simplest case, calibration measurements might be necessary, for example, or data related to different initial conditions, $y_{0,1}, y_{0,2}, \ldots, y_{0,\nu}, \ldots$, are given. Numerically, these situations can be handled by the concatenation of several IVPs,

$$y'_\nu = f_\nu(y_\nu\,;\,p), \qquad y_\nu(t_{0,\nu}) = y_{0,\nu}, \quad \nu = 1, 2, \ldots, \tag{2.3}$$

very similar to the management of breakpoints/events (see below). If required, the solution y_ν corresponding to the (virtual) initial timepoint, $t_{0,\nu}$, can readily be shifted to the (original) initial time, t_0, for comparison or plotting purpose.

2.3 Breakpoint Handling

A sudden event (maybe from outside the biological system) is handled by introducing a breakpoint, $t_b > t_0$, and subsequently, splitting the ODE system into a y^--part for $t_0 < t \le t_b$, and a y^+-part for $t_b < t$,

$$(y^-)' = f(y^-\,;\,p), \qquad y^-(t_0) = y_0 \tag{2.4}$$
$$(y^+)' = f(y^+\,;\,p), \qquad y^+(t_b) = g(y^-\,;\,p) \tag{2.5}$$

where $g : \mathbb{R}^n \times \mathbb{R}^q \longrightarrow \mathbb{R}^n$ is a mapping of the initial conditions; possibly dependent on the parameter vector, p. The approach of splitting the ODE system with respect to time also applies in the case of multiple experiments, at least for the cases described above.

2.4 Parameter Constraints

In order to enforce constraints such as positivity or upper and lower bounds on the unkown parameters to be determined in the model, a (differentiable) transformation, $\varphi : \mathbb{R}^q \longrightarrow \mathbb{R}^q$, can be applied resulting in a different parametrisation, u, of the model ODE system,

$$p = \varphi(u), \quad y' = f(y\,;\,\varphi(u)) = \tilde{f}(y\,;\,u) \tag{2.6}$$

A global positivity constraint on the parameter vector, p, can be achieved for example by the (componentwise) exponential transformation

$$p_i = \exp(u_i), \qquad i = 1, \ldots, q \tag{2.7}$$

To impose an upper and a lower bound, A and B, respectively, a sinusoidal transformation

$$p_i = A + \frac{B - A}{2}\,(1 + \sin u_i)\,, \qquad i = 1, \ldots, q \tag{2.8}$$

can be used. For a single bound, C, as last example in this section, a root square transformation

$$p_i = C \pm \left(1 - \sqrt{1 + u_i^2}\right), \qquad i = 1, \ldots, q \tag{2.9}$$

(with the upper sign for an upper bound and the lower sign for a lower bound) is possible.
The last two transformation formulae are especially eligible since, at least for small perturbations $\mathrm{d}p_i \approx \varphi'\,\mathrm{d}u_i$, the differentials are bounded and, most importantly, are essentially independent of the new parametrisation, u.
Note that the application of any transformation of the parameters obviously change the sensitivities of the parameters to the dynamical evolution of the ODE system, as will be shown next. Therefore, it is strongly recommended that parameter constraints should only be applied in order to prevent the parameter vector components, p_i, from taking on physically meaningless values.
Currently, only the global positivity constraint case is implemented in BioPARKIN, i.e. the user can only select whether to impose the positivity constraint by the exponential transformation on *all* parameters in a given model, or not.

2.5 Sensitivity Matrix

The dependence of the solution, $y(t\,;\,p)$, on the parameters, p, is characterised by the sensitivity (n, q)-matrix, $S = S(t)$, defined by

$$S_{ij}(t) = \frac{\partial y_i}{\partial p_j}(t). \tag{2.10}$$

The matrix can readily be computed as solution to the variational equation

$$S' = f_y(y\,;\,p)\,S + f_p(y\,;\,p), \qquad S(t_0) = 0 \tag{2.11}$$

Consequently, the variational equation associated with a transformed system reads

$$\begin{aligned} \tilde{S}' &= \widetilde{f}_y(y\,;\,u)\,\tilde{S} + \tilde{f}_u(y\,;\,u) \\ &= f_y(y\,;\,\varphi(u))\,\tilde{S} + f_p(y\,;\,\varphi(u))\,\varphi'(u), \qquad \tilde{S}(t_0) = 0 \end{aligned} \tag{2.12}$$

clearly showing the influence of the transformation, φ, upon the sensitivity matrix, $\tilde{S}$.

In the case that breakpoints/events are specified, the computation of the sensitivity matrix also splits into the computation of S^- and S^+,

$$\begin{aligned} (S^-)' &= f_y(y^-;\, p)\, S^- + f_p(y^-;\, p) && (2.13)\\ S^-(t_0) &= 0, && (2.14) \end{aligned}$$

$$\begin{aligned} (S^+)' &= f_y(y^+;\, p)\, S^+ + f_p(y^+;\, p) && (2.15)\\ S^+(t_b) &= g_y(y^-;\, p)\, S^-(t_b) + g_p(y^-;\, p) && (2.16) \end{aligned}$$

respectively.
Often, model species and model parameters cover a broad range of physical units and their values can vary over orders of magnitude. To achieve comparability, the absolute sensitivities have to be normalized by the absolute values of species and parameters to obtain relative sensitivities,

$$S_{ij}(t) = \left(\frac{\partial y_i}{\partial p_j}\right)(t) \cdot \frac{\max\{|p_j|\,,\, \mathrm{thres}(p_j)\}}{\max\{\max\limits_{t\in I}|y_i(t)|\,,\, \mathrm{thres}(y_i)\}} \tag{2.17}$$

where thres$(\cdot)$ are user-specified threshold values for parameters and species, respectively, and the integration time interval of the ODE system, I, is used. In the graphical user interface, i.e. in the Results Window, these relative sensitivities are displayed.
Note that, since the variational equation is an inhomogeneous *linear* equation, the qualitative behaviour of its solution trajectories is essentially determined by the distribution of the eigenvalues, i.e. the spectrum, of the factor of the linear term, $f_y(y\,;\,p)$, of the right-hand side. Accordingly, typical stability patterns of the trajectories such as convergence towards an equilibrium or, more likely, convergence towards a limit cycle, or even divergence, are possible. Of cource, for biology-related systems the latter behaviour is more natural, and thus more often observable.

2.6 Parameter Identification

Following the fundamental idea of Gauss, parameter identification is, as implemented in BioPARKIN, equivalent to solving the *weighted* least squares minimisation problem,

$$\frac{1}{M}\sum_{j=1}^{M} \|D_j^{-1}\big(y(\tau_j\,;\,p) - z_j\big)\|_2^2 = \min, \tag{2.18}$$

with diagonal weighting (n,n)-matrices,

$$D_j := \mathrm{diag}\Big((\delta z_j)_1,\, \ldots,\, (\delta z_j)_n\Big), \qquad j = 1, \ldots, M. \tag{2.19}$$

Note that, if not all components of a datum, $z_j \in \mathbb{R}^n$, are available for a specific measurement time point, τ_j, the missing data in the least squares formulation is simply replaced by the computable model value, therefore effectively neglecting the corresponding contribution in the sum (2.18). The corresponding entry in D_j is set to one (w.l.o.g.).
If, on the other hand, a component of given error tolerance, δz_j, or even the whole vector, is put to zero, this contribution to the sum (2.18) is also taken out, and considered as a (nonlinear) equality constraint to the least squares formulation instead.[1]

Remark. In the (hopefully rare) case of missing error tolerances, the following estimation approaches are used:
In PARKIN, the measurement tolerances are computed as

$$\big(\delta z_j\big)_k = \max\left\{\eta \cdot \max_i \big(z_i\big)_k,\, \big(z_j\big)_k\right\},$$

with some small safety factor η.

[1] This mechanism, however, is still to be made available to the BioPARKIN user.

In NLSCON, the measurement tolerances are computed as

$$\left(\delta z_j\right)_k = \max\left\{ \max_i \left(z_i\right)_k, \, \mathrm{thres}(z_j)_k \right\},$$

with some user specified threshold mapping, thres$(\cdot)$.

The least squares problem (2.18) may be written even more compactly as

$$\|F(p)\|_2^2 \equiv F(p)^T F(p) = \min, \tag{2.20}$$

where $F : \mathbb{R}^q \to \mathbb{R}^{\mathbb{L}}$ is a nonlinear mapping and structured as a stacked vector of length $L = nM$,

$$F(p) = \begin{bmatrix} D_1^{-1}\left(y(\tau_1\,;\,p) - z_1\right) \\ \vdots \\ D_M^{-1}\left(y(\tau_M\,;\,p) - z_M\right) \end{bmatrix}. \tag{2.21}$$

If *not all* components of a measurement, z_j, are given, the number L is accordingly made smaller, $L < nM$.
In all cases the goal is to minimise the relative deviation between model and data at the measurement time points, τ_i. The above problem, which is usually highly nonlinear in the unknown parameter vector, p, can be solved by affine covariant Gauss-Newton iteration [6] where each iteration step, k, calls for the solution of a *linear* least squares problem,

$$\|F'(p^k)\Delta p^k + F(p^k)\|_2^2 = \min, \tag{2.22}$$

$$p^{k+1} = p^k + \lambda_k \Delta p^k, \quad k = 0, 1, 2, \ldots \tag{2.23}$$

The update step uses a customised QR decomposition for solving the linear least squares problem, especially in the rank-deficient case.

3 Iterative Solution Approach

Having a set of measurement points as described above,

$$(\tau_1, z_1, \delta z_1)\,, \ldots, \, (\tau_M, z_M, \delta z_M), \tag{3.1}$$

with δz_j additionally denoting a statistical tolerance of the j-th measurement (i.e. standard deviation of z_j), the inverse problem of finding unknown parameters, p, can be stated most conveniently by the formalism as introduced in Section 2.6: Defining the pointwise deviations between model and data,

$$\delta y(\tau_j\,;\,p) := z_j - y(\tau_j\,;\,p), \qquad j = 1, \ldots, M \tag{3.2}$$

and the M diagonal weighting (n, n)-matrices,

$$D_j := \mathrm{diag}(\delta z_{j1}, \ldots, \delta z_{jn})\,, \qquad j = 1, \ldots, M$$

formed by the prescribed statistical tolerances, δz_j, a discrete (weighted) l_2-product is introduced by

$$(\delta y,\, \delta y) := \frac{1}{M} \sum_{j=1}^{M} \|D_j^{-1} \delta y(\tau_j\,;\,p)\|_2^2.$$

Formally, if some of the components of the j-th measurement, z_j, are not available, this would mean to set the corresponding components in the measurement tolerance, δz_j, to infinity, i.e. to take the contribution of these components out of the discrete l_2-product, just as described in Section 2.6. The given discrete data gives rise to a vector mapping, $F : \mathbb{R}^q \longrightarrow \mathbb{R}^L$, with $L = nM$ if all

compoments at every data point have been measured (really rare, not only in Systems Biology!), or otherwise some $L < nM$,

$$F(p) := \begin{bmatrix} D_1^{-1}\delta y(\tau_1\,;\,p) \\ \vdots \\ D_M^{-1}\delta y(\tau_M\,;\,p) \end{bmatrix}. \tag{3.3}$$

In the *compatible* case, it is assumed that there is at least one parameter vector, $p^* \in \mathbb{R}^q$, such that the positive cost functional, $c : \mathbb{R}^q \longrightarrow \mathbb{R}$,

$$c(p) := \|F(p)\|_2^2 \equiv (\delta y,\, \delta y), \tag{3.4}$$

vanishes for $p = p^*$, i.e. $F(p*) = 0$ holds.
Otherwise, if the last condition can not be achieved, any parameter vector, $p^* \in \mathbb{R}^q$, that minimises the cost functional,

$$c(p^*) = \min_{p\in\mathbb{R}^q} \|F(p)\|_2^2 \tag{3.5}$$

is referred to as a solution to the weighted least squares problem as well. In general, the task at hand is highly non-linear, ill-posed, and possibly underdetermined. Therefore, BioPARKIN tries to compute a solution to it by a successive, iterative approximation: the well-known, but yet refined Gauss-Newton method.

3.1 Parameter Scaling

Before dealing with the implemented Gauss-Newton algorithm in detail, a proper internal scaling has to be addressed. In general, a scaling-invariant algorithm, i.e. an algorithm that is invariant under the choice of units in a given problem, is (almost) indispensable to guarantee any reliable results. Therefore, the following scaling strategy within the course of the Gauss-Newton iteration has been implemented: an internal weighting vector, $pw \in \mathbb{R}^q$, is used to define the local scaling matrices, W_k , by

$$W_k = \mathrm{diag}(pw_1, \ldots, pw_q) \tag{3.6}$$

with locally given

$$pw_i := \max\left\{|p_i^k|,\, \mathrm{thresh}_i\right\}, \qquad i = 1, \ldots, q \tag{3.7}$$

where $\mathrm{thresh}_i > 0$ are a suitable threshold values for scaling chosen by the user. Consequently, any relative precision of parameter values below these prescribed threshold values will be insensible and meaningless.

3.1.1 Scaling Update Procedure

Listing 1: Internal Parameter Scaling

Input :

pw^{user} := user − supplied weighting vector

Initial check :

if $(|pw_i^{\mathrm{user}}| = 0)$:

$$pw_i^{\mathrm{user}} := \begin{cases} \mathrm{eps}, & \text{if problem highly nonlinear} \\ 1, & \text{if problem mildly nonlinear} \end{cases}$$

endif

Initial update:

$$pw_i^0 := \max\left\{ |pw_i^{\text{user}}|\,,\, |p_i^0| \right\}$$

Iteration update:

$$pw_i^k := \max\left\{ |pw_i^{\text{user}}|\,,\, \tfrac{1}{2}\left(|p_i^{k-1}| + |p_i^k|\right) \right\}$$

Summarising, the internal scaling matrix, W_k, and the error norm, $\|\cdot\|$ (weighted root mean square), used in the codes are given by

$$W_k = \operatorname{diag}\left(pw_1^k, \ldots, pw_q^k\right) \tag{3.8}$$

and

$$\|v\| := \|v\|_w := \sqrt{\frac{1}{q}\sum_{i=1}^{q}\left(\frac{v_i}{pw_i}\right)^2}, \qquad v \in \mathbb{R}^q. \tag{3.9}$$

3.2 Gauss-Newton Method

Starting with an initial guess, $p^0 \in \mathbb{R}^q$, the (damped) Gauss-Newton method is given as

$$p^{k+1} = p^k + \lambda_k \Delta p^k, \qquad k = 0, 1, \ldots \tag{3.10}$$

Here, the steplength, $0 < \lambda_k \leq 1$, is recomputed successively in each iteration (see below). The update, Δp^k, is the minimum norm solution to the *linear* least squares problem,

$$\|F'(p^k)\,\Delta p^k + F(p^k)\| \stackrel{!}{=} \min, \tag{3.11}$$

where the $(L \times q)$-Jacobian matrix, $F'(\cdot)$, can be approximated either by computing the difference quotient, for $\ell = 1, \ldots, L$ and $i = 1, \ldots, q$,

$$J_{\ell,i} = \frac{1}{h}\left(F_\ell(p + e_i h) - F_\ell(p)\right), \qquad h = \mathcal{O}\left(|p_i|\,\sqrt{\text{eps}}\right), \tag{3.12}$$

or by stacking rows of the sensitivity matrices, $S(\tau_j)$, corresponding to the measurement points $(\tau_j,\, z_j)$,

$$J = \begin{bmatrix} S(\tau_1) \\ \vdots \\ S(\tau_M) \end{bmatrix} \tag{3.13}$$

Either approach to compute the Jacobian matrix is possible in BioPARKIN, to make sure that, at each current parameter estimation, p^k, the approximation $J \approx F'(p^k)$ is valid. Note that the computation of the difference quotient avoids the costly integration of the variational equation, an ODE system of $((n+1) \times q)$ equations in total.

For later use, the notation of the so-called *simplified Gauss-Newton correction*, $\overline{\Delta p}^{k+1}$, as the minimum norm solution to

$$\|J(p^k)\,\overline{\Delta p}^{k+1} + F(p^{k+1})\| \stackrel{!}{=} \min, \tag{3.14}$$

may also be introduced here *en passant*.

3.2.1 Setplength Strategy

The determination of an optimal damping factor, $0 < \lambda_k \leq 1$, is based on the assumption that a global Jacobian Lipschitz constant, $\omega_* < \infty$, exists such that

$$\|F'(p^*)^+[F'(y) - F'(x)](y - x)\| \leq \omega_* \|y - x\|^2, \qquad x, y \in D \subset \mathbb{R}^q, \tag{3.15}$$

where $p^* \in \mathbb{R}^q$ dentoes the unknown solution of the minimisation problem as introduced before. Additionally, some local constants, $\omega_k \leq \bar{\omega}_k < \infty$, should exist satisfying

$$\|F'(p^k)^+[F'(y) - F'(x)](y - x)\| \leq \omega_k \|y - x\|^2, \tag{3.16}$$

$$\|F'(p^k)^+[F'(y) - F'(x)]\, v\,\| \leq \bar{\omega}_k \|y - x\| \cdot \|v\|, \qquad v, x, y \in D \subset \mathbb{R}^q. \tag{3.17}$$

Introducing, further, the quantity

$$\overline{\Delta}^k := -F'(p^k)^+ \left[F(p^k) + F'(p^{k-1})\overline{\Delta p}^k\right], \tag{3.18}$$

and using the bigger local Lipschitz constant, $\bar{\omega}_k < \infty$, it is seen that for the simplified Gauss-Newton corrections, $\overline{\Delta p}^k$, the inequality

$$\begin{aligned}
\|\overline{\Delta p}^k - \Delta p^k + \overline{\Delta}^k\| &= \|\left[I - F'(p^k)^+F'(p^{k-1})\right]\, \overline{\Delta p}^k\| && (3.19)\\
&= \|F'(p^k)^+\, F'(p^k)\, \left[I - F'(p^k)^+F'(p^{k-1})\right]\, \overline{\Delta p}^k\| && (3.20)\\
&= \|F'(p^k)^+\left[F'(p^k) - F'(p^{k-1})\right]\, \overline{\Delta p}^k\| && (3.21)\\
&\leq \bar{\omega}_k\, \lambda_{k-1}\, \|\Delta p^{k-1}\| \cdot \|\overline{\Delta p}^k\| && (3.22)
\end{aligned}$$

where maximal rank of $F'(x)$, at $x = p^k$, is assumed (thus, for $x = p^k$, the orthogonal projection fulfils $P(x, x) \equiv F'(x)^+F'(x) = I$) in the second line, and the Moore-Penrose Axiom $F'(x)^+\, F'(x)\, F'(x)^+ = F'(x)^+$ for a generalised inverse, in order to derive the third line. Hence, one obtains the *a priori* estimate

$$[\bar{\omega}_k] := \frac{\|\overline{\Delta p}^k - \Delta p^k + \overline{\Delta}^k\|}{\lambda_{k-1}\, \|\Delta p^{k-1}\|\, \|\overline{\Delta p}^k\|} \leq \bar{\omega}_k. \tag{3.23}$$

Now, considering the *global level function*

$$T_*(p) := T(p|F'(p^*)^+) := \frac{1}{2}\|F'(p^*)^+\, F(p)\|^2, \tag{3.24}$$

it can be shown that

$$T_*(p^k + \lambda\Delta p^k) \leq t_k^2\, T_*(p^k) \tag{3.25}$$

with

$$t_k := t_k(\lambda) = 1 - \lambda + \frac{1}{2}\lambda^2\, h_k^*, \tag{3.26}$$

and

$$h_k^* := \omega_*\, \|(F'(p^*)^+F'(p^k))^{-1}\| \cdot \|\Delta p^k\|. \tag{3.27}$$

Therefore, the descent measured in terms of the natural level function is maximised by minimising $t_k(\lambda)$, resulting in

$$\lambda_*^{\text{opt}} \quad = \quad \min\left\{1,\, \tfrac{1}{h_k^*}\right\}. \tag{3.28}$$

One can verify that

$$h_k := \omega_k\|\Delta p^k\| \leq h_k^*, \quad \text{and} \quad \omega_k \leq \omega_*\, \|(F'(p^*)^+F'(p^k))^{-1}\|. \tag{3.29}$$

Inserting the local estimate $[h_k^*] := [\bar{\omega}_k]\,\|\Delta p^k\|$ with $[\bar{\omega}_k]$ from above leads to the readily computable *a priori* estimate

$$\lambda_k^{(0)} = \min\{1,\, \mu_k\}\,, \quad \mu_k := \frac{1}{[\bar{\omega}_k]\,\|\Delta p^k\|}. \tag{3.30}$$

Note that this *prediction strategy* needs an initial estimate, $\lambda_0^{(0)}$, given by the user as additional input for the very first iteration as the prediction strategy requires information from the previous iteration step.
If, however, the damping factor, $\lambda_k^{(0)}$, still does not satisfy the *natural monotonicity test* in terms of the *local level function*, $T(p|F'(p^k)^+)$,

$$\left\|-\nabla T(p|F'(p^k)^+)|_{p=p^{k+1}}\right\| = \left\|\overline{\Delta p}^{k+1}\right\| < \left\|\Delta p^k\right\| = \left\|-\nabla T(p|F'(p^k)^+)|_{p=p^k}\right\| \tag{3.31}$$

an additional *correction strategy* is invoked to compute the *a posteriori* estimates,

$$\begin{array}{rcll} \lambda_k^{(\nu)} & = & \min\left\{1\,,\, \frac{1}{2}\lambda_k^{(\nu-1)},\, \mu_k^{(\nu-1)}\right\}, & \\ \mu_k^{(\nu-1)} & = & \frac{1}{\left[h_k(\lambda_k^{(\nu-1)})\right]}, & \nu = 1, 2, \ldots \end{array} \tag{3.32}$$

where

$$\left[h_k(\lambda_k^{(\nu-1)})\right] := \frac{2}{(\lambda_k^{(\nu-1)})^2}\,\frac{\|\overline{\Delta p}^{k+1,\nu-1} - (1-\lambda_k^{(\nu-1)})\,\Delta p^k\|}{\|\Delta p^k\|} \le h_k. \tag{3.33}$$

Usually, this *a posteriori* loop is rarely activated, as experience shows. To avoid an infinite loop, however, it is ensured that both estimates, $\lambda_k^{(0)}$ and $\lambda_k^{(\nu)}$, $\nu = 1, 2, \ldots$, always satisfy the condition

$$\lambda_k^{(\nu)} \ge \lambda_{\min}, \qquad \nu = 0, 1, 2, \ldots \tag{3.34}$$

with a minimal permitted damping factor, $\lambda_{\min}$, provided by the user. In case $\lambda_k^{(\nu)} < \lambda_{\min}$ deliberate rank reduction is invoked, which usually leads to larger damping factors. Otherwise, the Gauss-Newton iteration will be stopped.

3.2.2 Subcondition Monitor

For the solution of the linear least squares problem in each iteration step, a QR-decomposition of the associated Jacobian (L, q)-matrix, $J = F'(p)$,

$$Q\,J\,\Pi = \begin{pmatrix} R \\ 0 \end{pmatrix} \tag{3.35}$$

by applying Householder reflections with additional column pivot maximisation, is realised in BioPARKIN. Here, for simplicity, the full rank case is assumed where $q \le L$ and R is an upper triangular (q, q)-matrix, $R = (r_{ij})$. The permutation, Π, is determined such that

$$|r_{11}| \ge |r_{22}| \ge \ldots \ge |r_{qq}|. \tag{3.36}$$

For some required accuracy, $\delta > 0$, given by the user, the *numerical rank*, $\ell := \mathrm{rnk}(J)$, indispensible to the solution of ill-posed problems, is then defined by the inequality

$$|r_{\ell+1,\ell+1}| < \delta\,|r_{11}| \le |r_{\ell\ell}|. \tag{3.37}$$

Note that this definition is highly biased by both row and column scaling of the Jacobian. Introducing, nevertheless, the so-called subcondition number, for $\ell = q$, by

$$\mathrm{sc}(J) := \frac{|r_{11}|}{|r_{qq}|} \le \mathrm{cond}_2(J), \tag{3.38}$$

it follows that, if $\delta \cdot \mathrm{sc}(J) \ge 1$, the Jacobian will certainly be rank-deficient. In this case, a rank-deficient pseudo-inverse is realised, either by a QR-Cholesky variant or by a QR-Moore-Penrose variant. Both cases of pseudo-inverses of the Jacobian, J, will be denoted by $\left(J^\ell\right)^+$.

3.2.3 Rank Optimisation

A deliberate rank reduction may additionally help avoid an iteration towards an attractive point, $\hat{p}$, where the associated Jacobian matrix, $J(\hat{p})$, becomes singular. The general idea of this device is to reduce the maximal permitted rank in the QR decomposition until the natural monotonicity will be fulfilled again or, of course, no further rank reduction is possible. The subroutine to do so is as follows.
To start with, let q denote the current rank. The ordinary Newton correction, Δp^k, is then recomputed with a prescribed maximum allowed rank, $\ell = q - 1$. With the new (trial) correction, $\Delta p^{k,\ell}$, a new *a priori* damping factor, a new trail iterate, and a new simplified correction,

$$\lambda_k^{(0,\ell)} = \min\left\{1,\, \mu_k^{(0,\ell)}\right\}, \tag{3.39}$$

$$p^{(0,\ell)} = p^k + \lambda_k^{(0,\ell)} \Delta p^{k,\ell}, \tag{3.40}$$

$$\Delta p^{k,\ell} = -J^\ell(p^k)^+ F(p^k), \tag{3.41}$$

$$\overline{\Delta p}^{(0,\ell)} = -J^\ell(p^k)^+ F(p^{(0,\ell)}), \tag{3.42}$$

are computed, respectively.
If now the monotonicity check is successfully passed, the Gauss-Newton iteration proceeds as usual. Otherwise, the damping factors, $\lambda_k^{(\nu,\ell)} (\nu = 1, 2, \ldots)$, are calculated using the *a posteriori* estimates as given above. If in the *a posteriori* loop, in turn, $\lambda_k^{(\nu,\ell)} < \lambda_{\min}$ occurs, the maximum allowed rank is further lowered by one and, again, the repetition of the rank reduction step starts once more.
This rank reduction procedure is carried out until natural monotonicity, $\|\overline{\Delta p}^{(\nu,\ell)}\| \leq \|\Delta p^{k,\ell}\|$, holds true or, alternatively, a final termination criterion, $\ell < \ell_{\min}$ $(0 < \ell_{\min} < q)$, is reached.
Note that the application of a rank-reduced Newton step means to perform an intermediate Gauss-Newton step. Although, in principle, both methods are algorithmically very similar, there are some essential theoretical differences. As a consequence, the local Lipschitz constant estimates, $[\bar{\omega}_k^{(\nu,\ell)}]$, have to be modified accordingly,

$$[\bar{\omega}_k^{(\nu,\ell)}] := \frac{\left(\|\overline{\Delta p}^{(\nu,\ell)} - \Delta p^{k,\ell} + \overline{\Delta}^k\|^2 - \Delta^2\right)^{\frac{1}{2}}}{\lambda_{k-1}\, \|\Delta p^{k-1}\|\, \|\overline{\Delta p}^k\|} \leq [\bar{\omega}_k], \tag{3.43}$$

with the norm, $\Delta \geq 0$, of the projection

$$\Delta = \| \left[I - J^\ell(p^k)^+ J^\ell(p^k)\right] \overline{\Delta p}^k \|, \tag{3.44}$$

see [6] for further details.
Note that an emergency rank reduction can occur in a step where the rank of the Jacobian, $J(p^k)$, has already been reduced because of the sub-condition criterion.

3.3 Pseudo-Code

An informal algorithm, including damping strategy and rank reduction as explained, may be given as follows.

```
1
2  Input:
3  ------
4  p^0          initial guess (of dimension q)
5  λ_0          initial damping factor
6  λ_min        minimal permitted damping factor
7  ℓ_min        minimal permitted rank
8  cond_max     maximal permitted subcondition number
9  ε_req        required accuracy of solution
10 k_max        maximal number of iterations
11 F, J         user-supplied routines describing the problem to solve
```

Start:

$k \leftarrow 0$
$F_k \leftarrow F(p^k)$
(**A**) $J_k \leftarrow J(p^k)$
$\ell_{\max} \leftarrow q$
(**B**) $\Delta p^k \leftarrow -\left(J_k^\ell\right)^+ F_k$
ℓ maximal s.t. $\ell \le \ell_{\max},\ \mathrm{sc}(J_k^\ell) < \mathrm{cond}_{\max}$
(by $QR-$decomposition)
$\lambda_k^{(0,\ell)} \leftarrow \begin{cases} \lambda_0, & k = 0 \\ \min\left\{1,\, \mu_k^{(0,\ell)}\right\}, & k > 0 \end{cases}$
$\nu \leftarrow 0, \quad \lambda_k^{(-1,\ell)} \leftarrow 1$
$\lambda \leftarrow \lambda_k^{(0,\ell)}$
(**C**) **if** $\left((\ell = q \vee \ell = \ell_{\min}) \wedge (\lambda < \lambda_{\min}) \wedge (\lambda_k^{(\nu-1,q)} > \lambda_{\min})\right)$:
$\lambda \leftarrow \lambda_{\min}$
endif
if $(\lambda < \lambda_{\min})$:
$\ell_{\max} \leftarrow \ell_{\max} - 1$
if $(\ell_{\max} < \ell_{\min})$: **exit**("rank strategy failed")
else : **continue** at (**B**)
endif
$p^{k+1,\nu} \leftarrow p^k + \lambda\, \Delta p^k$
$F_{k+1,\nu} \leftarrow F\left(p^{k+1,\nu}\right)$
$\overline{\Delta p}^{k+1,\nu} \leftarrow -\left(J_k^\ell\right)^+ F_{k+1,\nu} \qquad \left(\left(J_k^\ell\right)^+ \text{ from (**B**)}\right)$
if $\left((\|\overline{\Delta p}^{k+1,\nu}\| \le \varepsilon_{\mathrm{req}}) \wedge (\|\Delta p^k\|^2 \le 10 \cdot \varepsilon_{\mathrm{req}}) \wedge (\lambda = 1)\right)$:
$p^* \leftarrow p^{k+1,\nu} + \overline{\Delta p}^{k+1,\nu}$
if $(\ell = q)$: **exit** ("solution", p^*)
if $(\ell < q)$: **exit** ("stationary point", p^*)
endif
$\lambda_k^{(\nu+1,\ell)} \leftarrow \min\left\{1,\, \mu_k^{(\nu+1,\ell)}\right\}$
if $\left(\|\overline{\Delta p}^{k+1,\nu}\| \le \|\Delta p^k\|\right)$:
$p^{k+1} \leftarrow p^{k+1,\nu}$
$F_{k+1} \leftarrow F_{k+1,\nu}$
$k \leftarrow k + 1$
if $(k > k_{\max})$: **exit**("too many iterations")
else : **continue** at (**A**)
else :
$\nu \leftarrow \nu + 1$
if $(\lambda = \lambda_{\min})$: **exit**("damping strategy failed")
$\lambda \leftarrow \min\left\{\lambda_k^{(\nu,\ell)},\, \frac{1}{2}\lambda\right\}$
$\lambda \leftarrow \max\{\lambda,\, \lambda_{\min}\}$
continue at (**C**)
endif

Listing 2: Global Gauss-Newton Scheme with Damping and Rank Strategy

3.4 Further Practical Issues

It is well-known that the numerical realisation of the routines calculating the forward model equation, F, and the Jacobian, J, have to be numerically accurate enough in order to preserve dif-

ferentiability of the underlying model with respect to species and parameters. Moreover, since the biological networks are likely to be stiff, the integrator solving the ODE systems has to be capable to deal with stiff ODE systems. In BioPARKIN, a linearly-implicit extrapolation code is used (LIMEX) for computing the numerical solution of the ODE system, with detailed order and stepsize control enabled: in the local convergence domain of the Gauss-Newton iteration, i.e. $\lambda = 1$, the scaling and the stepsizes for the stiff itegrator are preserved [8]. Thus differentiability is ensured, provided that, near the unknown solution, the integration order stays constant during this iteration phase.

3.4.1 Statistical *a posteriori* Analysis

Provided the iterative Gauss-Newton procedure indeed converges to a solution vector, $p^* \in \mathbb{R}^q$, the linearised model, taken at this solution point, readily enables a statistical *a posteriori* analysis. Assuming that a QR decomposition (with column pivoting) of the unscaled (!) Jacobian, $J(p^*) \simeq F'(p^*)$,

$$J(p^*)\Pi = Q \cdot R, \qquad r := \operatorname{rank}(R) \tag{3.45}$$

is available, the variance, $\sigma^2 = \sigma^2(p^*)$, of the residual with respect to the given data is estimated by

$$\widehat{\sigma}^2 = \frac{1}{M-r} \sum_{j=1}^{M} \|y(\tau_j\,;\,p^*) - z_j\|_2^2 \,. \tag{3.46}$$

Using the splitting of the upper triangular factor,

$$R = \begin{bmatrix} \widetilde{R} \\ 0 \end{bmatrix} \qquad \text{with } (r,r)-\text{matrix } \widetilde{R}, \tag{3.47}$$

the variance-covariance matrix w.r.t. the solution vector, p^*, is then computed by

$$\operatorname{cov}(p^*) = \widehat{\sigma}^2 \left(\widetilde{R}^T \widetilde{R}\right)^{-1} \tag{3.48}$$

and, having set $V := \operatorname{cov}(p^*)$ for abbreviation, correspondingly the correlation matrix, $C \in \mathbb{R}^{q,q}$, by

$$C_{ij} = \frac{1}{\sqrt{V_{ii}}\,\sqrt{V_{jj}}}\, V_{ij}, \qquad i,j = 1,\,\ldots,\,q. \tag{3.49}$$

Note that the last formula tacitly implies that an entry C_{ij} is set to zero if (at least) one of the square roots is zero.

4 Cross-Platform Development

BioPARKIN is made up of two main components. All the numerical functionality resides in a separate library called PARKINcpp. The graphical user interface (GUI) for managing models, data and for interacting with the numerical library—e.g. invoking actual computations—is the other part, called BioPARKIN. Thus, BioPARKIN is both the name of the complete software package as well as of the GUI.

4.1 PARKINcpp & BioPARKIN

PARKINcpp. As the name suggests, PARKINcpp is written in C++. The intent of this library is to make algorithms developed at the ZIB available inside a modern and modular package. A lot of PARKINcpp's functionality previously existed only in legacy FORTRAN code that is not suitable to be used inside other projects.
The development of PARKINcpp alongside the BioPARKIN GUI ensures that the library has a mature API (Application Programming Interface) that can just as easily be used within other software projects.

C++ was chosen because it is both abstract enough to support modern design principles and close enough to the hardware to provide good performance—an important trait for a numerical software package.

BioPARKIN. BioPARKIN is written in Python and uses the popular Qt GUI toolkit (for details, see Section 6).
Python is an interpreted script language that is suitable for rapid prototyping and for quickly adapting the code to changes. Its popularity has been increasing for years, especially in fields like biology and bioinformatics.
The Python interpreter—needed to run Python code—is available for all major operating systems as is the Qt library. BioPARKIN was intended to be a cross-platform tool from the very beginning.

4.2 Supported Operating Systems

While BioPARKIN's GUI is developed in Python and works on several operating systems without any further adjustments, the PARKINcpp library has to be compiled for each platform specifically. Currently, there are versions for Windows and Linux. Support for Mac OS X is possible.

4.3 Extending Existing Software

With the advent of the SBML standard a whole ecosystem of software tools has sprung up (see Section 1.2). Most of the existing software is intended for a single purpose and is not suited to be the basis for a project like BioPARKIN whose goal is of wider intent. However, some software—e.g. CellDesigner (see Section 1.2)—has interfaces dedicated to accept plug-ins of different kinds.
The plug-in architecture of several software tools was evaluated. In all cases, doubts about the feasibility of integrating all of BioPARKIN's intended features into a plug-in remained. Experience shows that the development of plug-ins often hits the boundaries of what the underlying framework allows.
In order to be as independent as possible and retain full control over the software environment, BioPARKIN was realised as a stand-alone software project. Details about the implementation can be found in the following sections.

5 Numerical Library: PARKINcpp

The C++ subpackage of BioPARKIN consists of several classes that have been divided into different subdirectories, or modules, respectively. These modules are briefly described as follows,

`addpkg`
: In this subdirectory all additional packages are collected. Currently, the packages dlib, ExprEval, LIMEX4_3A, newmat11, and Ode (dop853) are included. In the present version of PARKINcpp, however, only the packages dlib (matrix/vector backend), and LIMEX4_3A (a linearly implicit extrapolation code for solving ODE systems) are used.

`common`
: Here, common constants and types (`typedefs`) in so-called header files are listed.

`linalg`
: This subdirectory incorporates typical tasks of Linear Algebra, such as QR decomposition. It serves also as base for the implementation (or class interface) of the fundamental `Matrix` and `Vector` classes. Note that particular emphasis has been put here on providing an almost one-to-one mapping of MATLAB-like methods of matrix-/vector-operations and procedures. It is absolutely clear that the classes of this section are the core building blocks for all other classes in PARKINcpp.

`nonlin`
: In this subdirectory the non-linear solvers are gathered, including the important interface definition of the user function `UserFunc`. Presently, both NLSCON and PARKIN are available here.

`odelib`
: Here, the interface (and adapter) classes to any solvers of ordinary differential equations (and systems thereof) are put together. In future versions of PARKINcpp, this place shall also include (interface) classes for solving delay equations, for example.

`swig`
: This subdirectory contains the so-called interface definition files for SWIG, a wrapper generator especially suited for (non-typed) script languages such as Python. Admittedly, the content of this directory needs a complete review, if not a complete redesign. Done correctly, this would provide a clean interface of PARKINcpp to other languages (e.g. Java, Lua, Perl, Ruby, Tcl, to name a few) as well.

`system`
: In this subdirectory, all classes relevant for the description of—and solutions to—(biological) systems are listed. In particular, the fast evaluation of abstract syntax trees (AST) for the description of the right-hand side (RHS) of an ODE system is enabled by the two classes, Expression and ExprNode.

`tstprg`
: Last but not least, some test routines and examples how to use the classes of PARKINcpp are given here.

The general approach to the development and overall design of the C++ package has been, on the one side, to re-use as many as possible code already avaible in other high level programming languages (FORTRAN77: `limex4_3a.f`, `park11.f`; MATLAB: `nlscon.m`, `deccon.m`, `solcon.m`), and, on the other side, to have a clean object-oriented setup right from the start.

5.1 Functionality

Usage of the C++ package can most succinctly be explained by giving code snippets, demonstrating the basic functionality of the library.

C++

```
// !!!
// This code snippet is for demonstration purpose only. It will NOT run.
// !!!

#include "bioparkin.h"
using namespace BIOPARKIN;

int main()
{
      Real                      tstart = -15.0;
      Real                      tend   = 100.0;
      BioSystem                 biosys(tstart,tend);

      BioRHS::ExpressionMap     exprs;
      BioSystem::Species        species;
      BioSystem::Parameter      parameters;
      BioSystem::Param          parvals,initvals,guessvals,rslts;

      biosys.setODESystem(exprs);
      biosys.setParameters(parameters);
      biosys.setInitialValues(initvals);
      biosys.setParamValues(parvals);

      //

      Vector                          timepts;
      BioSystem::MeasurementList      measvals;
      BioSystem::Param                guessthres,measthres;

      biosys.setMeasurementList(timepts,measvals);

     //

      int             rc   = 0;
      Real            xtol = 1.0e-4;
      IOpt            iopt;
      BioProcessor    proc( &biosys, "nlscon" );

      proc.setIOpt(iopt);
      proc.setCurrentParamValues(guessvals);
      proc.setCurrentParamThres(guessthres);
      proc.setCurrentSpeciesThres(measthres);

      rc = proc.identifyParameters(xtol);

      if ( rc == 0 ) rslts = proc.getIdentificationResults();

      return rc;
}
```

Listing 3: Code snippet showing the typical structure of a PARKINcpp application.

Input/Output Facilities. The first code snippet (Listing 4) given in this paragraph demonstrates how to enter a simple ODE system describing the reaction mechanics $A + B \rightleftharpoons C$ with

reaction rate constants, k_1 and k_2, respectively.

C++

```
//
//  Simple  A + B <=> C  example ; with reaction rates k1, k2, resp.
//
//    (1)       A' = - k1 * A * B + k2 * C
//    (2)       B' = - k1 * A * B + k2 * C
//    (3)       C' = - 2 * k2 * C + 2 * k1 * A * B
//

      BioSystem                 biosys( 0.5, 6.0 );
      BioRHS::ExpressionMap     emap, aux;
      BioSystem::Species        species;


      species.push_back("A");
      species.push_back("B");
      species.push_back("C");


      aux["rct"] = Expression(TIMES, "k1", Expression(TIMES,"A","B"));
      aux["rev"] = Expression(TIMES, "k2", species[2]);


      // define Eqn (1)
      emap["A"]          =  Expression(MINUS, aux["rev"], aux["rct"]);

      // define Eqn (2)
      emap[species[1]]   =  Expression(MINUS, aux["rev"], aux["rct"]);
      // or  emap["B"]   =  emap[species[0]];    alternatively

      // define Eqn (3)
      emap[species[2]]   =  Expression( MINUS,
                               Expression(TIMES, 2.0, aux["rct"]),
                               Expression(TIMES, 2.0, aux["rev"])
                            );


      biosys.setODESystem(emap);

      biosys.setInitialValue(species[0], 1.0);
      biosys.setInitialValue(species[1], 2.0);
      biosys.setInitialValue("C", 0.0);

      // ...

//
```

Listing 4: Formulating reacting kinetics as ODE systems within PARKINcpp.

The following list of operations is available within the Expression interface:

PLUS, MINUS, TIMES, DIVIDE, POWER,
ABS, CEIL, FLOOR, SIGN,
EXP, LN, LOG,
SIN, COS, TAN, SINH, COSH, TANH,
ARCSIN, ARCCOS, ARCTAN, ARSINH, ARCOSH, ARTANH

HILLplus , HILLminus

All listed operators here take either one or two arguments, as usual, with the only exception being the so-called Hill functions,

$$H^{+}(x,y,p) := \frac{c}{1+c}, \qquad \text{with } c := \left(\frac{x}{y}\right)^{p} \tag{5.1}$$

$$H^{-}(x,y,p) := \frac{1}{1+c}, \qquad \text{again with } c = \left(\frac{x}{y}\right)^{p} \tag{5.2}$$

The second example (Listing 5) shows how to assign parameters in an ODE system with values, e.g. with inital start guesses for a first simulation run.

C++

```
//

      BioSystem                 biosys(0.0, 100.0);
      BioRHS::ExpressionMap     emap;
      BioSystem::Parameter      parameters;
      BioSystem::Param          parvals;

      // ...
      biosys.setODESystem(emap);
      // ...

      parameters.push_back("p1");
      parameters.push_back("p2");
      parameters.push_back("p3");
      // ...

      std::string s = parameters[2];   // <-- this is "p3" here

      parvals["p1"]             = 7.32;
      parvals[parameters[1]]    = -3.14;
      parvals[s]                = sqrt(2.0);
      // ...


      biosys.setParameters(parameters);
      biosys.setParamValues(parvals);
      // ...

//
```

Listing 5: Setting parameter values in ODE systems in PARKINcpp.

The core items of each instantiation of the BioSystem class are available by corresponding getter methods, e.g. `getODESystem()`, `getParamValues()`.

Parameter Transformation. The transformation of the parameters can be switched by an input option field/array that must be given in the setup of the BioProcessor performing the Gauss-Newton iteration. This is illustrated in the code snippet given by Listing 6.

C++

```
//

      BioSystem      biosys(-1.0, 1.0);
      // ...

      int            nParameter = 5;
      BioProcessor   proc( &biosys, "nlscon" );
      IOpt           iopt;
      Vector         trans;      // <-- Vector always starts to
                                 //     count from _one_ !

      trans.zeros(nParameter);
      trans(1) = 1;              // apply exponential transformation
      trans(4) = 1;              // to first and fourth parameter _only_

      iopt.transf = 1;           // apply/switch for all p.
      iopt.itrans = trans;       // select transformation for single p.

      proc.setIOpt(iopt);
      // ...

//
```

Listing 6: Mechanism of parameter transformation switching.

The following flags are currently available:

`iopt.transf = 0`: No transformation.

`iopt.transf = 1`: Exponential transformation, $p = \exp(u)$.

`iopt.transf = 2`: Lower bound transformation with $L \in \mathbb{R}$, $p = L - \left(1 - \sqrt{1 + u^2}\right)$.

`iopt.transf = 3`: Upper bound transformation with $U \in \mathbb{R}$, $p = U + \left(1 - \sqrt{1 + u^2}\right)$.

`iopt.transf = 4`: Lower and upper bound transf. $(L < U)$, $p = L + \frac{1}{2}(U - L)(1 + \sin(u))$.

The actual values of the upper and lower bounds are passed by corresponding `iopt.lbnd` and `iopt.ubnd` fields, respectively.

Approximation of Jacobian. In principle, the trajectories of the linearised model—the solution of the variational equation needed to set up the Jacobian—can be computed approximatively in two ways, either as numerical solution of the variational equation, or as difference quotient approximation. A snippet for this mechanism is given in Listing 7.

C++

```
//
    BioSystem           biosys( 0.0, 15.0 );
    BioProcessor        proc( &biosys, "nlscon" );
    TrajectoryMap       trajMap, scaledTrajMap;
    IOpt                iopt;

    iopt.jacgen = 3;        // <- use difference quotient instead
    proc.setIOpt(iopt);     //    of biosys supplied var. eqn
    // ...

    Expression::Param   parVals;
    Expression::Param   parThres, speThres;

    proc.setCurrentParamValues(parVals);     // <- p. values where
                                             //    sens. are taken

    proc.setCurrentParamThres(parThres);     // <- thresholds for
    proc.setCurrentSpeciesThres(speThres);   // <- species and p.

    //

    trajMap       = proc.computeSensitivityTrajectories();
    scaledTrajMap = proc.getScaledSensitivityTrajectories();

    //

    int                 rc = 0;
    Vector              timepoints;
    MatrixList          matList;
    QRconDecompList     qrList;

    timepoints.zeros(2);
    timepoints(1) = 2.34;
    timepoints(2) = 4.56;

    rc = proc.prepareDetailedSensitivities( timepoints );

    if ( rc == 0 )
    {
        matList = proc.getSensitivityMatrices();
        // qrList = proc.getSensitivityDecomps();
    }

    // ...

//
```

Listing 7: Sensitivity computation by evaluating the Jacobian matrices.

Statistical *a posteriori* Analysis. Following a successful Gauss-Newton run, BioPARKIN also offers a statistical *a posteriori* analysis (see Listing 8).

```
//
        BioSystem           biosys(0.0,10.0);

        int                 rc = 0;
        Real                xtol = 1.0e-4;
        IOpt                iopt;
        BioProcessor        proc( &biosys, "nlscon" );
        Expression::Param   idParam;
        Matrix              vcvMat, corMat;

        iopt.qstat = true;
        proc.setIOpt(iopt);
        // ...

        rc = proc.identifyParameters(xtol);

        if ( rc == 0 )
        {
                idParam = proc.getIdentificationResults();
                vcvMat  = proc.getVarianceCovarianceMatrix();
                // corMat  = proc.getCorrelationMatrix();
        }

//
```

Listing 8: Determination of the variance-covariance matrix following a successful identification run.

5.2 Foreign Library Access

A simplified wrapper and interface generator, i.e. a package known as SWIG, provides most conveniently the possibility to use the numerical C++ library PARKINcpp in other programming languages as well. The SWIG package is a mature and popular open-source tool to enable code that is written in a variety of languages to use native C and C++ routines and objects, respectively. Moreover, the design of SWIG supports simultaneously to generate interfaces for a number of scripting (Python, Ruby, Perl) and non-scripting languages (Java, C#, and other) [1]. All it needs to make C/C++ classes and functions accessible to other languages is to write so-called interface files. The complexity of these interfaces depends on the wrapped C/C++ code, e.g. the types of function arguments.
Currently, PARKINcpp provides interfaces especially for the script language Python (found in the `swig` subdirectory, see beginning of Section 5). These interface files define all necessary conversion information, e.g. C++ templates, to generate the interface code.[2]
The generated interface code results in a `parkin.py` file that can simply be imported in Python. This import modules comes together with a special dynamical link library file `_parkin` (`_parkin.pyd` on Windows, and `_parkin.so` on Linux), also compiled by SWIG from the original C++ code, PARKINcpp.
Currently, only Python-specific interface files are available within PARKINcpp. The nature of SWIG, however, allows to support additional programming languages (e.g. Java) if needed. This turns PARKINcpp into an attractive numerical library for a lot of different projects or groups.

[2] A PDF (of a talk by SWIG's inventor) showing one of the earliest applications of SWIG and some details on interfaces can be found here: http://www.dabeaz.com/presentations/SwigPyTalk.pdf

5.3 Design Patterns

This section lists the design patterns that have been identified as extremely useful during the entire PARKINcpp development, sorted by the classes they have been applied to.

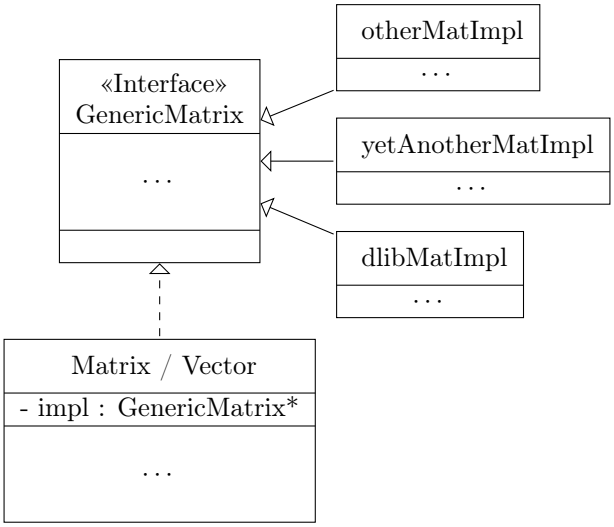


Figure 5.1: Adapter pattern used in BioPARKIN for Matrix/Vector class

`class Matrix / class Vector`
: Uses a type of an *adapter pattern* in order to unify different interfaces of existing matrix packages (such as the dlib C++ package) to the matrix interface used throughout PARKINcpp. The PARKINcpp matrix interface has been developed with the basic aim to imitate a MATLAB-like syntax as much as possible. The usage of the adapter pattern, additionally, decouples the interface definition from the implementation and, consequently, facilitates a hypothetical future change of the underlying matrix implementation enormously.

`class Expression`
: This class is structured as an *abstract syntax tree (AST)* since the objects of this type shall represent arbitrary mathematical formulae, the right-hand sides of an ODE system, for instance. The leaves of this tree are the terminal symbols, i.e. variable names and constant numbers, and the other nodes hold the operators with one, two, or three operands. Moreover, since (symbolic) differentiation is a pure algebraic operation, these objects also know how to take the derivative of themselves, w.r.t. a given variable.

`class BioSystemWrapper` (special wrapper class)
: Provides the call-back interfaces typically required by numerical ODE solvers such as LIMEX. Especially, if the requesting code is written in legacy FORTRAN, this *wrapper idiom* also cares for the correct function type linkage since FORTRAN package libraries often expect function signatures different from the usual ones of C++ compilers. The crucial point for successfully linking with FORTRAN, here, is to use solely *static member functions* as the call-back routines passed to the FORTRAN library.

`class BioProcessor / class BioPAR`
: These classes introduce a further abstraction layer into the PARKINcpp design. They are built, more or less, on *composition* rather than inheritance since they intrinsically carry a "has a"-relationship, but not the relation "is a". In this way, multiple systems could be dealt with simultaneously much easier, in principle.

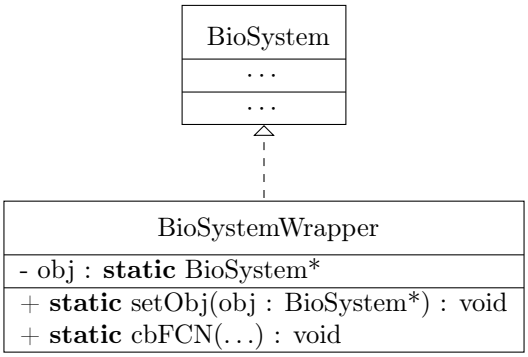


Figure 5.2: Wrapper idiom for call-back routines, as required by static-typed libraries

6 Graphical User Interface: BioPARKIN

This section gives an overview of the graphical user interface (GUI) of BioPARKIN, both from a user-oriented perspective (see Section 6.2.1 and Section 6.2.2) and from a technical developer-oreinted perspective (see Sections 6.3.2, 6.3.3, and 6.3.4).

6.1 Functionality

The GUI makes most of PARKINcpp's features available for easy use. Those features are detailed in Section 5.1. The following paragraphs describe the functionality that is added by the BioPARKIN GUI layer.
Additional guidance on how to use BioPARKIN can be found in the BioPARKIN manual [3].

6.1.1 SBML Compatibility

The SBML standard uses Versions and Levels to distinguish between SBML specifications with different feature sets. Each development cycle of SBML increments the Version, which generally increases the number of modelling cases that can be realized within the SBML syntax. Within a Level, Versions are increased when issuing "bugfixes" or removing contradictions within the standard [4]. The most recent release of SBML is Level 3 Version 1. Level 3, however, can still be considered to be a young standard. The available software mostly supports SBML Level 2 Version 4 which had long been the most recent specification before Level 3 was released in October 2010. When work on BioPARKIN started, SBML Level 2 Version 4 was the current specification. Due to the fact that this Level still has the strongest hold within the field, BioPARKIN focuses on supporting it. Supporting Level 3 will be possible later on.
Currently, BioPARKIN only supports a subset of SBML Level 2 Version 4. SBML's event definitions, for example, support more types of events than BioPARKIN is able to handle at the moment. Such exceptions are usually reported by errors or warnings within the user interface or the log file.

6.1.2 File Input and Output

BioPARKIN uses SBML to handle model files (see previous section). An additional format is needed to store experimental data and simulation results.
A simple custom CSV-like format was chosen to save such data. It uses tabulators to separate fields and has the structure given in Table 1. The first column has to be named "`Timepoint [unit]`" with "`unit`" defining the actual time unit of the measured (or simulated) data. Of the following columns, every column ending with "`[unit]`" is treated as data ("`[]`" is possible, too, for unit-less data). The names of these data columns have to correspond to species IDs within the model so that BioPARKIN can match model entities and data.
If a column called "`SD`" is preceded by a data column it holds the standard deviations of the measurements in the preceding column.

Timepoint [unit]	ID1 [unit]	SD	ID2 [unit]	SD	Arbitrary Column
0	0	1	0	1	...
3.33	0.4	1	0.09	1	...
...	...	...	...	...	...

Table 1: Structure of a BioPARKIN data file. Neigboring cells are separated by a tab character.

Columns with arbitrary names can be appended at the end. As long as they do not contain square brackets, they are effectively ignored by BioPARKIN when doing calculations. This can be helpful to keep track of patient IDs within files, for example.
BioPARKIN not only reads files formatted this way but uses the same format to save simulation results to disk.

6.1.3 Data Browser

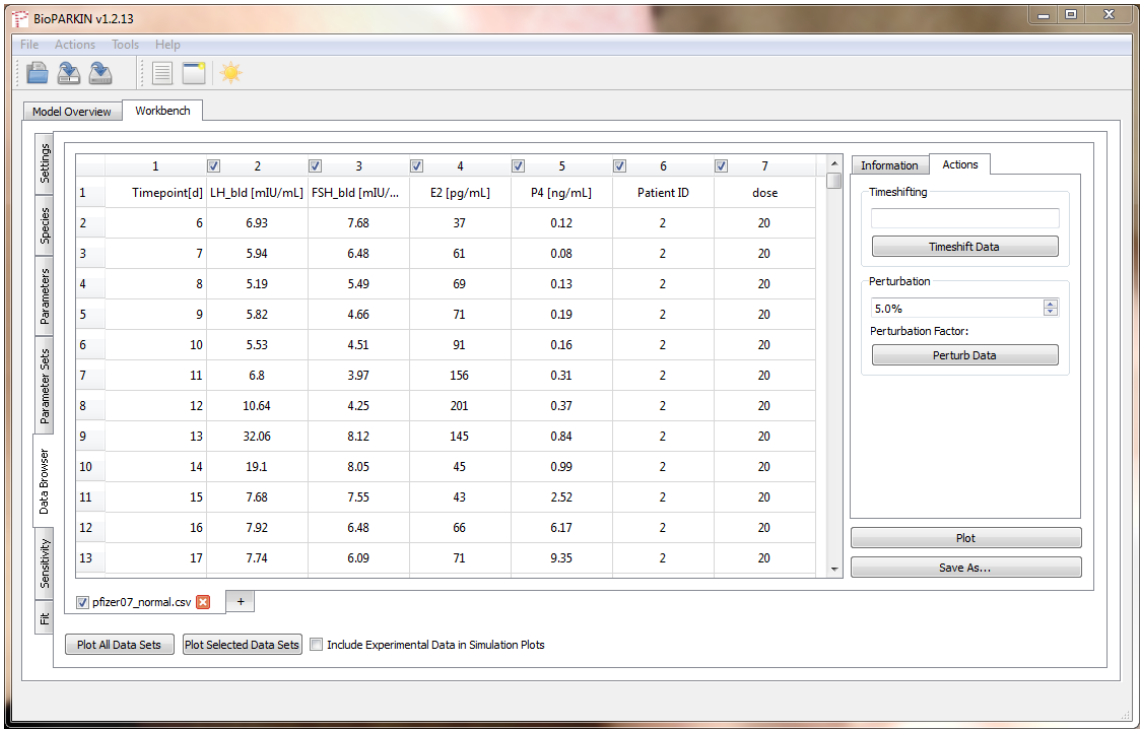


Figure 6.1: BioPARKIN's Data Browser showing a data file.

Data files are opened within the Data Browser interface (see figure 6.1). The data is shown in a table to provide the user with a well-known view of the data. Each loaded file is shown in its own tab at the bottom (hence, the name "Browser"). The "+" tab shows a page where additional files can be selected and opened.
An important feature of the Data Browser is the ability to select or deselect individual data files and individual columns within a file. Files can be selected via a checkbox on their respective tab. Columns can be selected via a checkbox in the their respective table header. Only selected data is used when doing computations. In this way, it is easy to use only a subset of a data set (e.g. only a specific species) when fitting model parameters.
Moreover, a timeshift can be applied to the data (individually per file). This is often needed in order to match the start of the simulation to the data time points.
When BioPARKIN is started with "`--debug`" from the command line, the Data Browser shows an additional option to perturb the data using uniformly randomised noise. This is useful for testing

purposes as it allows the user to take simulated data, perturb it slightly, and then use this data to "re-estimate" the parameters that were used to generate the data in the first place.
Currently, editing the data is not enabled. The Data Browser should be used to view and to select the data, not to edit it. This view-only approach may change, though, once BioPARKIN supports merging data.

6.1.4 Parameter Selection

The PARKINcpp library supports the computation of sensitivities and the identification of parameter values for subsets of parameters. This functionality is exposed in BioPARKIN. The researcher can select the parameters of interest which can speed up computation times.[3]

6.1.5 Parameter Sets

When a researcher is working on a model or exploring its parameter space, parameter values are usually changed quite often. The parameter sets feature of BioPARKIN tries to accommodate for this behaviour.
Each parameter set includes all parameters and their values. Parameter sets are displayed side-by-side in the simulation tab and can be duplicated or removed.
Upon opening a SBML model, an "Original" parameter set is created that allows the user to revert back to the initial parameter values at any time. Another set is created automatically, called "Guess". This is meant to be used in conjunction with the parameter identification. The results of an identification run are also put into a parameter set and, thus, can be plotted, compared with measurements, etc.

6.1.6 ODE System Generation

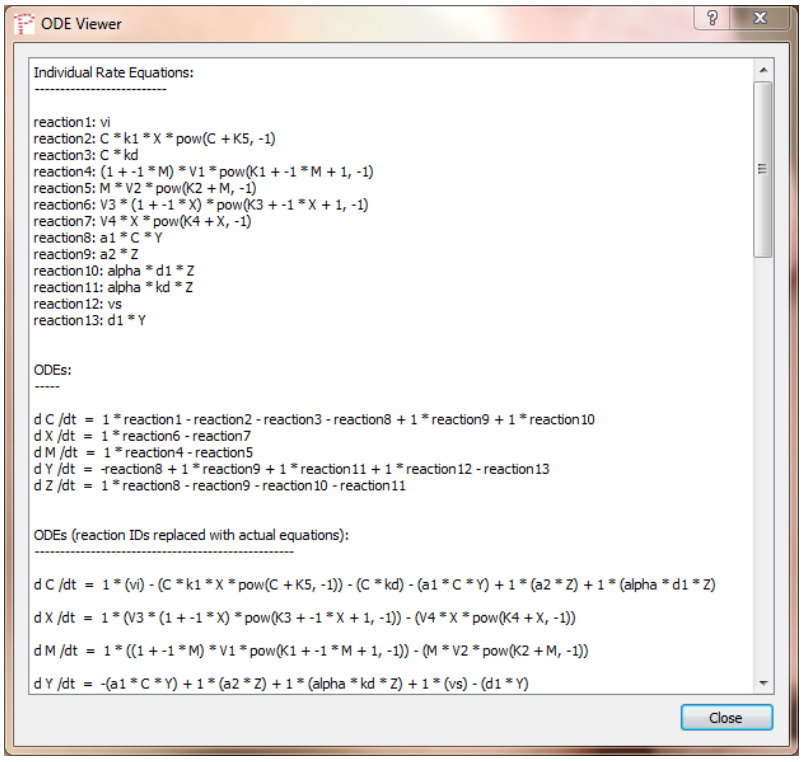


Figure 6.2: ODE overview as generated by BioPARKIN

[3]Selecting fewer parameters speeds up the computation only if Numerical Differentiation (either of the two modes) is selected for Jacobian calculation in the Setting tab of the Workbench.

SBML files do not usually define ordinary differential equations (ODEs) directly. Instead, they define the interactions between Species as Reactions with reaction kinetics. BioPARKIN parses the reaction network and creates one differential equation per Species.
The generated ODE system can be reviewed by the user (see Figure 6.2). This ODE system is used when setting up the PARKINcpp backend for computations. The ODEs shown to the user in the ODE Viewer window are shown in various levels of detail—e.g. a level with parameter IDs, and a level with their names.
For implementation details, see Section 6.3.4.

6.2 Workflow

6.2.1 Window Layout

The user experience (UX) of a software program is usually understood as the sum of usability, look, and feel. It thereby extends the notion of whether a software is merely usable to include the question if a software is "a joy to use" and if it pleases the eye. Instead of frustrating the user, current software often tries to reward the user for using it[19, 12].
While it is too far-fetched to say that BioPARKIN has great looks and feel, the development always had good UX in mind. The arrangement of windows, tabs, and buttons was optimised several times as it became clear how a better usability could be achieved. This section illustrates the (current) design that this approach led to.
Invoking BioPARKIN, the user first encounters a window with menu bar and icons at the top, as usual. The main part of the window is taken up by two tabs that either show the current model or the simulation controls.

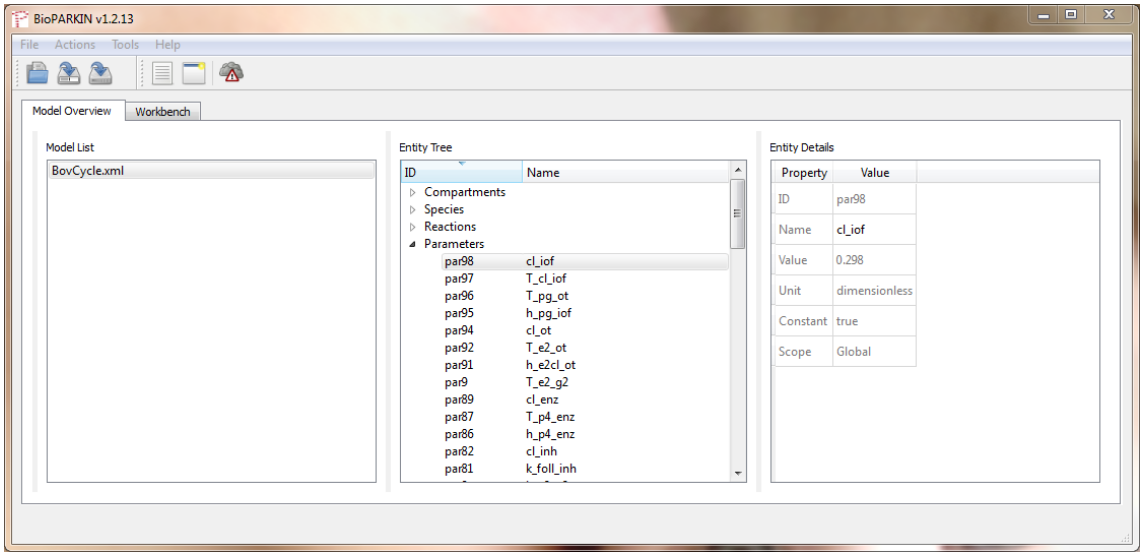


Figure 6.3: BioPARKIN's Model Overview tab showing a model and a selected parameter

Model Overview Tab. Within the Model Overview tab (see figure 6.3), the currently opened model is shown on the left.[4] In the center, there is a tree-like view of all the model's entities—e.g. Compartments, Species, Reactions. A table with details of the currently selected entity is placed on the right.
A future update of BioPARKIN will allow editing the model—e.g. adding and removing species and reactions—from within this tab. Currently, the model tab is only meant to convey an overview of the parts of a model.

[4]BioPARKIN currently supports only one model to be open at the same time. The framework does support more models at once but this has implications on workflow and usability that will be solved in a future version.

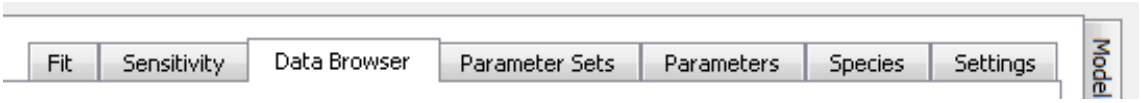


Figure 6.4: Tabs within the Workbench (the screenshot has been rotated 90° clockwise)

Workbench Tab. The Workbench tab hosts all the settings and controls to do the following (see Figure 6.4):

- Start simulations
- Plot simulated and experimental data
  - Manipulate and save the plots
- Show simulated and experimental data in tables
  - Save and recolour (based on a user-defined threshold) data
- Compute parameter sensitivities
  - Generate an overview and plot the sensitivitiy trajectories of all parameter/species pairs
  - Define timepoints and look at sensitivity (Jacobian) matrices and subconditions at these timepoints
- Identify parameter values based on experimental data
- Manage experimental data
  - Select relevant data
  - Apply a timeshift to data
- Manage parameter sets
  - Easily compare and plot variants of parameter sets
- Change global settings with impact on all computations
  - Switch between two Gauss-Newton variants in PARKINcpp (NLSCON and PARKIN, see Section 5)

6.2.2 Workflow Details

This section illustrates a typical workflow within BioPARKIN.
Currently, BioPARKIN is focused on simulating SBML models, providing sensitivity analysis and parameter estimations. The model tab gives a quick impression of the model but is not (yet) meant to build or to edit a model. For the time being, following the principle of interoperatibility between SBML software tools, users are kindly asked to use other SBML programs in order to build and modify SBML models, e.g. to make use of CellDesigner (see Section 1.2).
Actual usage of the software is covered in more detail in the BioPARKIN manual [3].

Loading and viewing a model. The Model Overview tab (as seen in Figure 6.3) shows relevant model entities in the central tree view. There are main nodes for SBML Compartments, Species, Parameters, Reactions, Rate Rules, Assignment Rules and Events. Each of these nodes can be expanded by clicking the plus/triangle sign[5] on the left of it. This reveals all entities of this type within the model.
The table on the right shows the details of the entity selected in the central tree. Most entries in that table cannot be edited and are shown so that the user can get an understanding of the model. The name of the selected entity can always be edited. Depending on the type of entity, different additional properties can be edited as well. These are the properties which can be edited:

All Entities
`Name` (most useful for parameters so that they can be more easily identified in the ODE Viewer).

Reactions
`Math` can be edited to change the actual reaction kinetic of a reaction.

Assignment and Rate Rule
`Variable` can be edited to change the left-hand side of this rule, i.e. the target. `Math` can be changed to define the right-hand side of the rule, e.g. the assignment.

Event
The target of the event can be changed via `Target`. The `Trigger` can be changed as well. It has to be given in the format *"*`eq(time, x)`*"* (where `x` can be any real number that lies strictly within the integration interval). Finally, `Expression` defines what will happen to the target in case the trigger is fired (e.g. if just a parameter ID is given, the value of this parameter is assigned to the target).

Plotting Simulations. Common settings for all computations can be defined in the Settings tab. Regarding (forward) simulation, the most important ones are Start Time and End Time.
Simulate buttons can be found in the lower-right corner of several tabs (Settings, Species, Parameters, Parameter Sets). Another way to start a simulation run is to use the keyboard shortcut `CTRL+ALT+S`.
The species trajectories are computed at timepoints that are chosen automatically by the adaptive stepsize control within the integrator LIMEX (see Section 3.4). Timepoints are not interpolated before being shown in the Results window (see next section). As a result, some parts of the output curves may seem rough (few support points), other parts may be drawn extremely smooth (lots of points). These are the exact data produced by and used within the integrator. In this way, the user can see what happened during the computation, preventing wrong assumptions—based on arbitrary interpolation.
Each simulation run produces two result tabs within the Results window—one showing a plot and another presenting the timecourse data in a table (see Figure 6.5 for an example).

[5]The exact look of the tree widget and its expand signs depends on the operating system used.

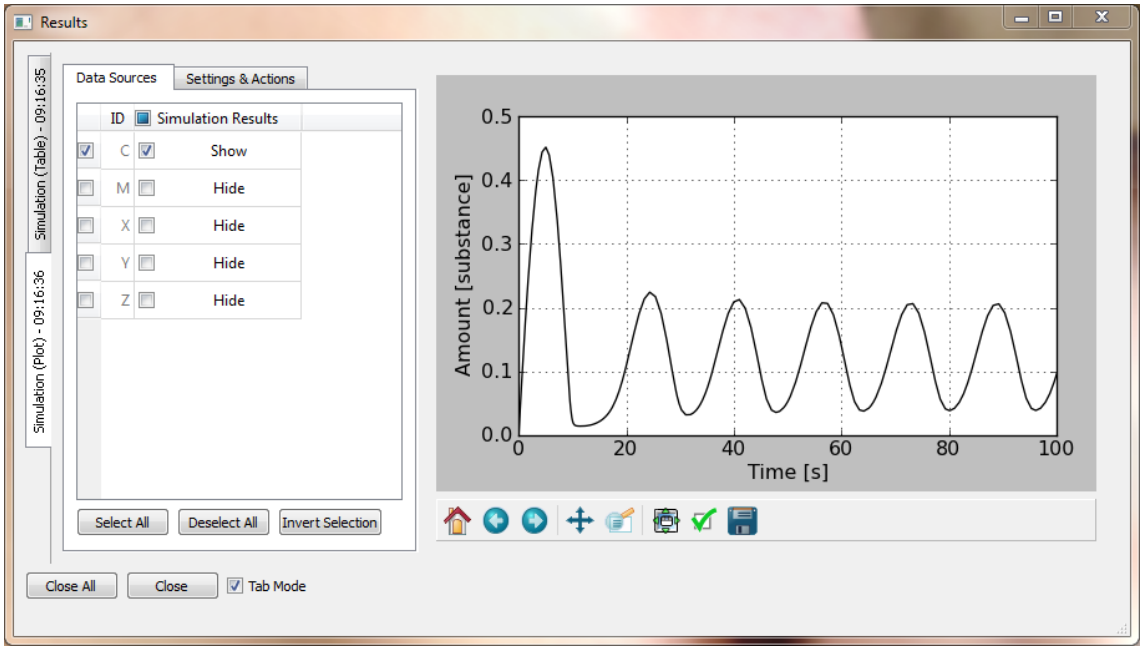


Figure 6.5: Results Window with two tabs. The active tab shows the plot of a simulation run.

Results Window. All results (simulations, as well as sensitivity and parameter identification computations) are shown in the Results window (see Figure 6.5). The window opens the first time anything is computed. If the user closes it, it can be reopened via a button in the menu bar at the top.
Per default, the Results window uses a tabbed interface. Each computation result (let it be a plot or a table) is placed within its own tab in the left-sided tab bar. A checkbox (Tab Mode) at the bottom can be deselected to switch to a windowed mode in which every tab is replaced by a small window (inside the surrounding Results window). If this is done, two buttons will appear (Tile Windows and Cascade Windows) which can be used to automatically rearrange all the results. This can be used, for example, to conveniently arrange plots and view them side-by-side (see Figure 6.6).

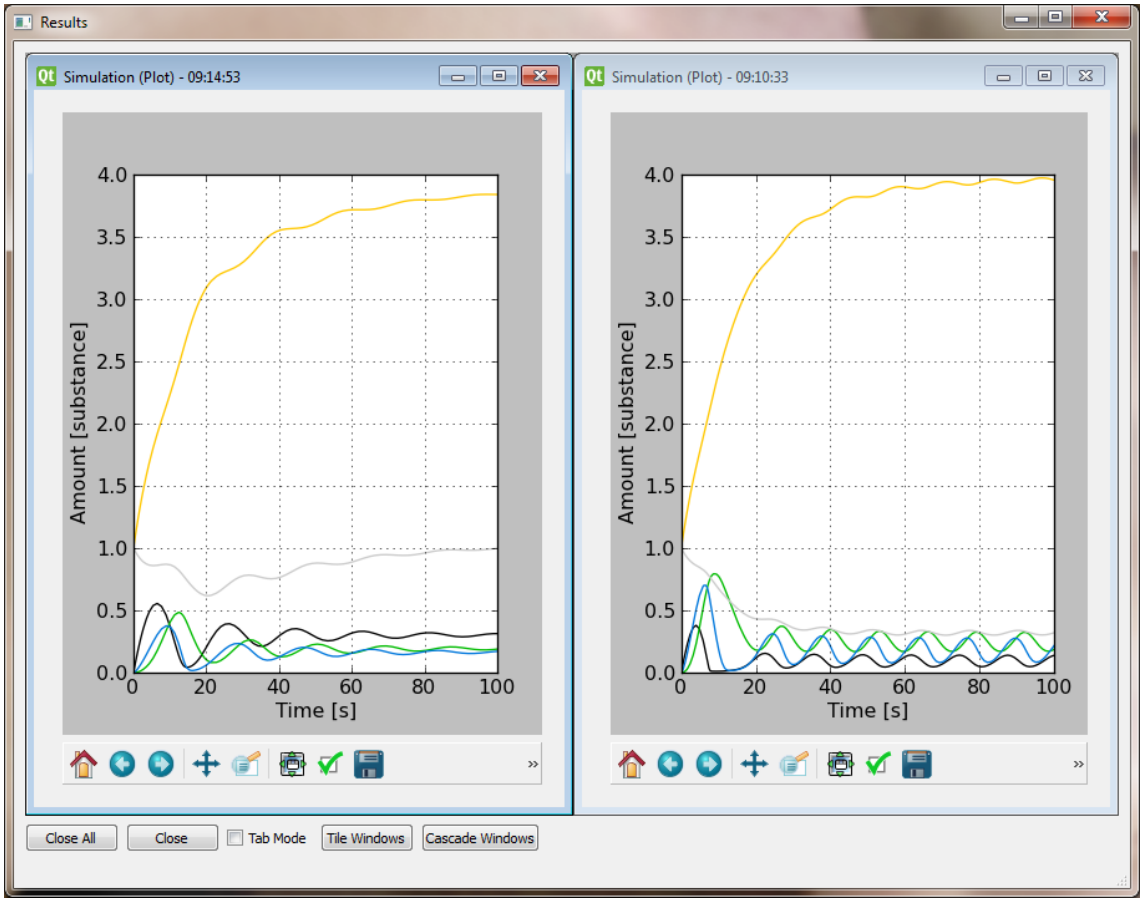


Figure 6.6: Results Window in non-tabbed mode, showing two plots side-by-side.

On the left side of each result (again, plot or table) there is a Data Sources tab. This tab shows all available data sources, e.g. species in the case of a simulation, or species/parameter combinations in the case of a sensitivity computation. It can be used to filter the results. Per default, only the first item is selected. Checkboxes in the row/column headers make it possible to select whole rows/columns at once.
The Settings & Actions tab provides some additional functionality such as saving the table/plot, showing a plot legend, and colouring table cells based on a threshold, see the BioPARKIN manual for more detailed explanations on how to use these features [3].

Computing Sensitivities. The Sensitivity tab shows two tables. The table on the left is used to select parameters. The right table allows to select species. This allows the user to predefine the parameter/species combinations for which sensitivities will be computed.
The number of selected parameters has an impact on computation time if the computation of the Jacobian matrix is set to Numerical Differentiation (either of the two modes) in the Settings tab. Conversely, if it is set to use Variational Equations, the number of selected parameters has no impact on computation times (see Section 3.2).
The number of selected species has no impact on computation times. It is merely a convenient way to filter the output of the computations beforehand. Internally, sensitivities are always computed for all the species (for each selected parameter).
Sensitivities can be shown in two ways as will be explained in the next paragraphs.

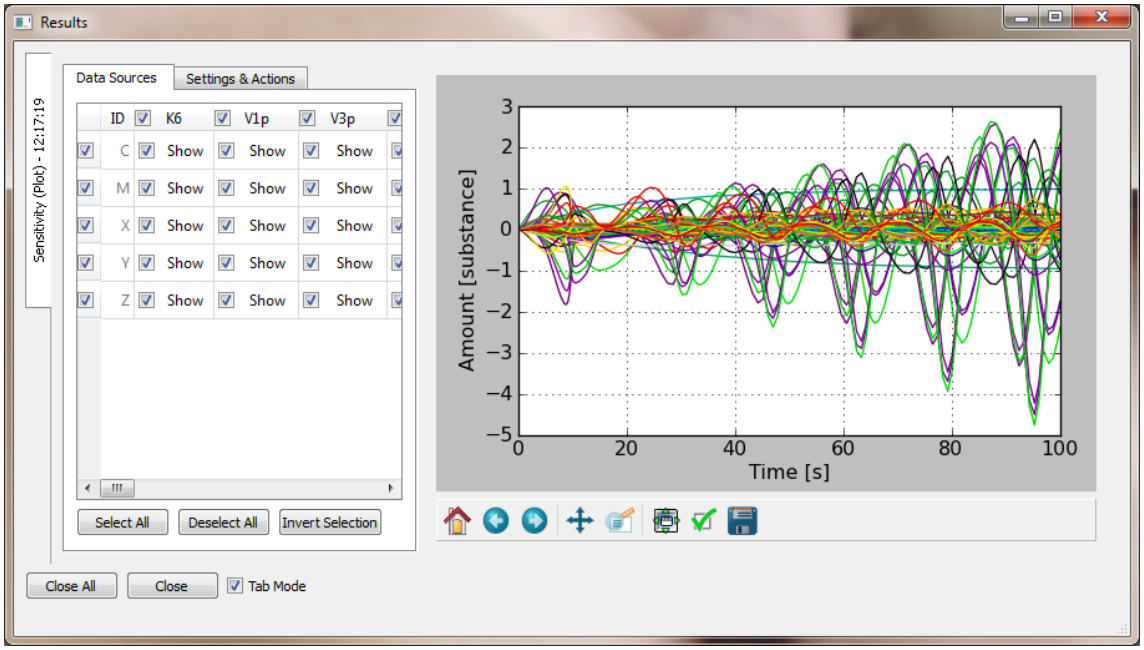


Figure 6.7: Example of a Sensitivity Overview plot with all parameter/species combinations selected

Sensitivity Overview. The sensitivitiy matrix—the Jacobian—is computed at every timepoint that is chosen by the adaptive stepsize control of the integrator (e.g. LIMEX). For each parameter/species combination the values within these Jacobian matrices form a trajectory which will be plotted by activating the Sensitivitiy Overview (see Figure 6.7).
This overview enables the user to assess how the sensitivity of a parameter (with respect to a species *and* the current initial value of the parameter) changes over time.

Detailed Sensitivities. Activating Detailed Sensitivities, the user is prompted to define timepoints for which to compute sensitivities. This is done within the Time Chooser window (see Figure 6.8).
This window offers three ways to construct a list of timepoints:

From Data
: In the top-left, there is a button to extract all timepoints from the data that is currently loaded (see Section 6.1.3 about the Data Browser).

By Interval
: In the top-right, the user can specify a timespan and a number of intervals (or an interval size) to compute any number of timepoints.

By Hand
: All timepoints appear in a large text area at the bottom. The user can just accept the timepoints, determined by one of the previous steps, or he can manually enter arbitrary timepoints. BioPARKIN will only take those timepoints into account that lie strictly inside the simulation time interval defined in the Settings tab. If the user provides incompatible input (e.g. letters), the GUI prompts to rectify the timepoint list before BioPARKIN proceeds.

Detailed sensitivity information will be provided for *all* timepoints specified within the Timepoint Chooser. Having completed all requested computations, two tabs per timepoint are shown within the Results Window:

Sensitivities
: This is the "raw" Jacobian matrix for all parameter/species pairs at a given timepoint. A

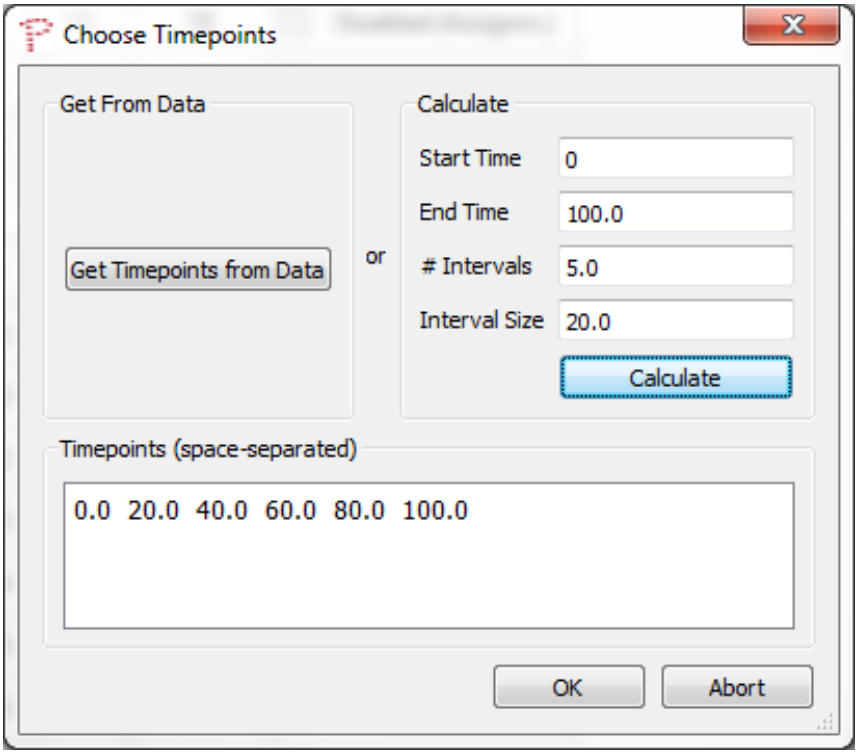


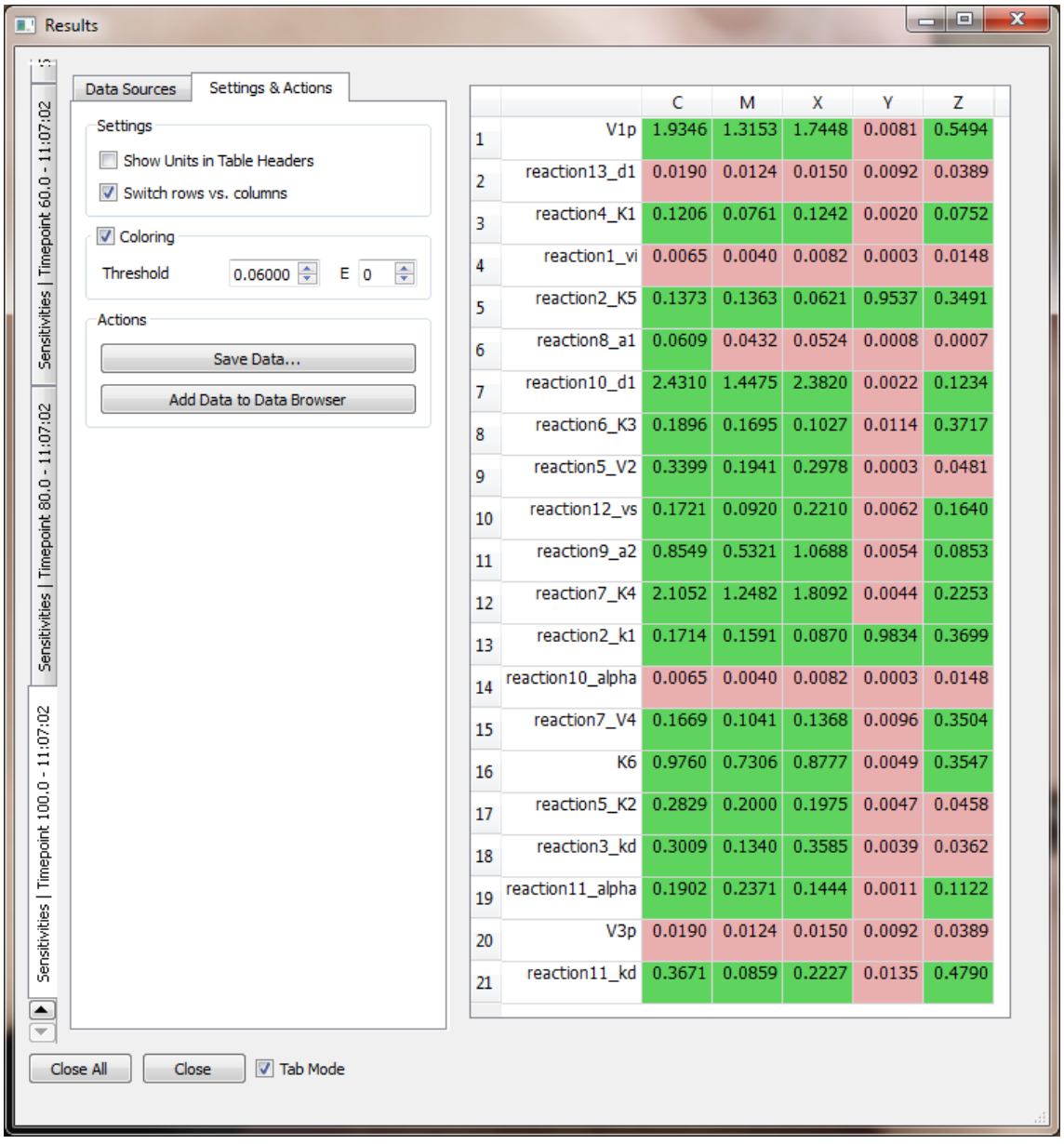
Figure 6.8: The Timepoint Chooser is shown before detailed sensitivities are computed. Sensitivity information will be shown for *all* specified timepoints.

threshold-based coloring is applied to all the cells by default. The threshold can be adjusted within the Settings & Actions tab of the result window (see Figure 6.9).

Subconditions

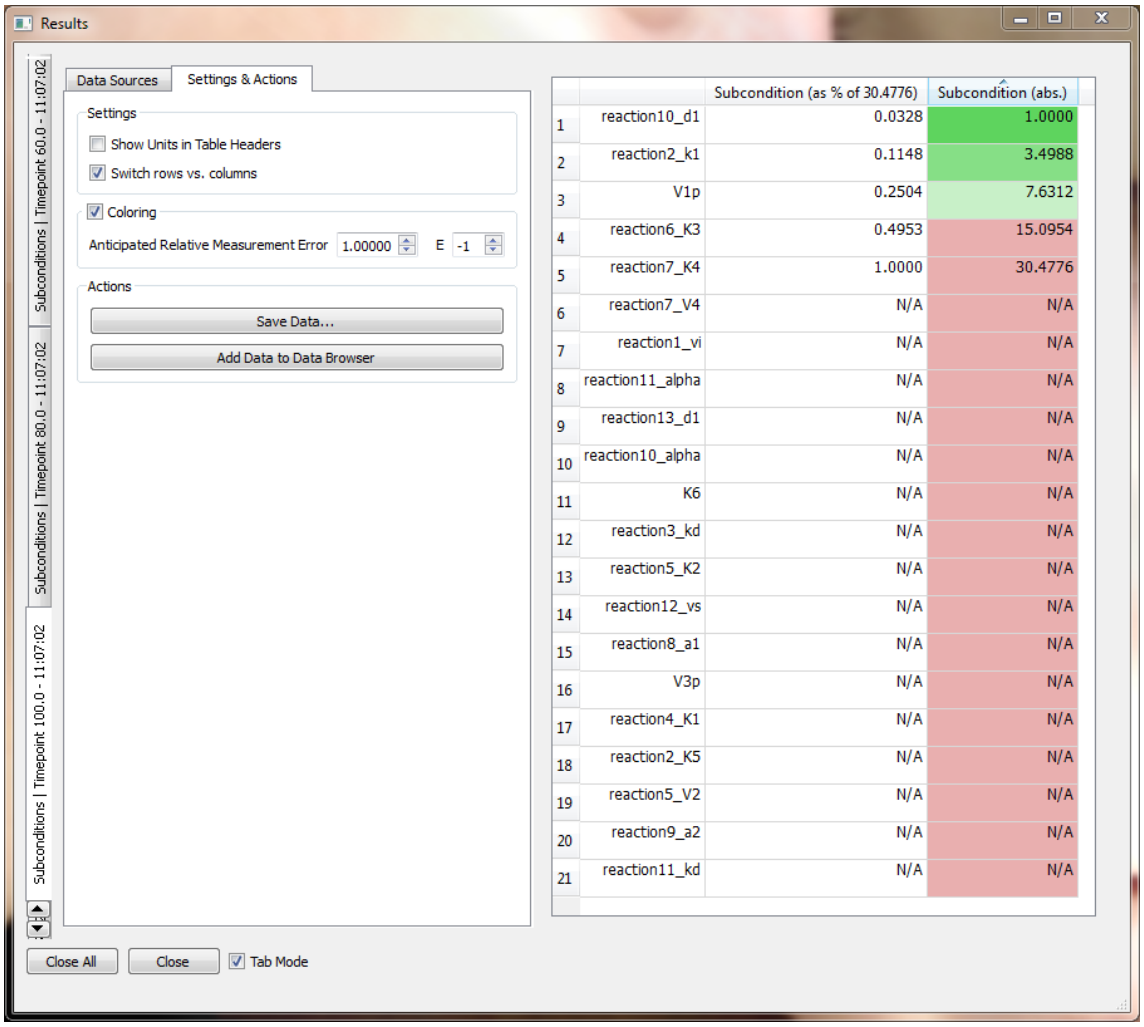
For each given timepoint, the subconditions of all selected parameters are computed (see Section 3.2.2). The subcondition numbers are shown in a simple table with two columns per parameter. The right column shows the absolute subcondition value. The largest valid value is used as a basis to normalize the values as shown in the left column (see Figure 6.10). Subconditions are an indicator for the sensitivity of a given parameter at a given timepoint (with regard to the parameters current value). The lower the subcondition, the higher the sensitivity. Subcondition cells can be coloured based on a user-defined Anticipated Relative Measurement Error. By default, this value is set to the RTOL value defined in the Settings tab. By increasing the value (e.g. from 1E-07 to 1E-03) the user can get an impression (watching the colors in the absolute subcondition column change) of the theoretical identifiability of the parameters if the measurements have this relative measurement error.

Identifying Parameters. The first step for identifying parameters is to set sensible initial values for all parameters. The best way to do this is within the Parameter Sets tab. Here, the user can create and handle multiple sets of parameter values. Because these sets are not overwritten after performing the parameter identification, they are a good starting point to test different combinations of parameter values and the resulting fit (see Section 7.3 for an example).
Inside the Fit tab, the user can select the parameters whose values should be identified. A click on Identify Parameters starts the computation. The results of this estimation run are presented within a a tab in the Results window. More importantly, they are automatically put into their own set in the Parameter Sets tab. Hence, the user can immediately reuse the estimated values, e.g. to create a simulation plot.



		C	M	X	Y	Z
1	V1p	1.9346	1.3153	1.7448	0.0081	0.5494
2	reaction13_d1	0.0190	0.0124	0.0150	0.0092	0.0389
3	reaction4_K1	0.1206	0.0761	0.1242	0.0020	0.0752
4	reaction1_vi	0.0065	0.0040	0.0082	0.0003	0.0148
5	reaction2_K5	0.1373	0.1363	0.0621	0.9537	0.3491
6	reaction8_a1	0.0609	0.0432	0.0524	0.0008	0.0007
7	reaction10_d1	2.4310	1.4475	2.3820	0.0022	0.1234
8	reaction6_K3	0.1896	0.1695	0.1027	0.0114	0.3717
9	reaction5_V2	0.3399	0.1941	0.2978	0.0003	0.0481
10	reaction12_vs	0.1721	0.0920	0.2210	0.0062	0.1640
11	reaction9_a2	0.8549	0.5321	1.0688	0.0054	0.0853
12	reaction7_K4	2.1052	1.2482	1.8092	0.0044	0.2253
13	reaction2_k1	0.1714	0.1591	0.0870	0.9834	0.3699
14	reaction10_alpha	0.0065	0.0040	0.0082	0.0003	0.0148
15	reaction7_V4	0.1669	0.1041	0.1368	0.0096	0.3504
16	K6	0.9760	0.7306	0.8777	0.0049	0.3547
17	reaction5_K2	0.2829	0.2000	0.1975	0.0047	0.0458
18	reaction3_kd	0.3009	0.1340	0.3585	0.0039	0.0362
19	reaction11_alpha	0.1902	0.2371	0.1444	0.0011	0.1122
20	V3p	0.0190	0.0124	0.0150	0.0092	0.0389
21	reaction11_kd	0.3671	0.0859	0.2227	0.0135	0.4790

Figure 6.9: Detailed view of the Jacobian matrix at timepoint 100 with a threshold of 0.06 to emphasise differences (e.g. between species Y and the rest) in sensitivity. There is one row per parameter and one column per species.



		Subcondition (as % of 30.4776)	Subcondition (abs.)
1	reaction10_d1	0.0328	1.0000
2	reaction2_k1	0.1148	3.4988
3	V1p	0.2504	7.6312
4	reaction6_K3	0.4953	15.0954
5	reaction7_K4	1.0000	30.4776
6	reaction7_V4	N/A	N/A
7	reaction1_vi	N/A	N/A
8	reaction11_alpha	N/A	N/A
9	reaction13_d1	N/A	N/A
10	reaction10_alpha	N/A	N/A
11	K6	N/A	N/A
12	reaction3_kd	N/A	N/A
13	reaction5_K2	N/A	N/A
14	reaction12_vs	N/A	N/A
15	reaction8_a1	N/A	N/A
16	V3p	N/A	N/A
17	reaction4_K1	N/A	N/A
18	reaction2_K5	N/A	N/A
19	reaction5_V2	N/A	N/A
20	reaction9_a2	N/A	N/A
21	reaction11_kd	N/A	N/A

Figure 6.10: Subconditions of selected parameters at timepoint 100. Note that this example model has only 5 species, so it is not possible to compute subconditions for more than 5 parameters (see Section 3.2). The Anticipated Relative Measurement Error is set to 1E-01, leading to three "green" subconditions at the right. So, in theory, three parameters remain identifiable even if the measurements have a high relative error of 10^{-1}.

All the relevant options for configurating the identification backend can be found in the Settings tab. The user can choose between the two Gauss-Newton implementations of the PARKINcpp library (NLSCON & PARKIN) at the top of this tab, and configure options at the bottom.

6.3 Implementation

6.3.1 Underlying Framework

The next two sections briefly sum up the software foundation of BioPARKIN. For details, refer to the developer documentation[6].

Python. Python[7] is an interpreted programming language. It can be used as a scripting language but is equally suited for full-fledged software programs. It supports a variety of programming paradigms, including object-oriented, functional, and imperative approaches.
The language is dynamically-typed, i.e. variables do not have a type at development time. The type of objects is only known at runtime. This often helps in rapid-prototyping, and allows for some constructs that are invalid in statically-typed languages.

Qt Framework. Development of the Qt[8] framework—spoken "cute"—started in 1991. Trolltech, Qt's original developer, was acquired by Nokia in 2008.
Currently, Qt is available with different licenses, enabling the use in open-source as well as in commercial projects.
Qt is a vast and stable framework to assist the programming of graphical desktop applications. Parts of the framwork—e.g. threading, event processing, database access—can also be useful in projects that do not have a graphical frontend.
Qt provides a library containing a large number of "widgets" (such as windows, dialog boxes, lists, and tables) as well as other tools that all help work with the library—e.g. the QDesigner which itself is a graphical application to put together window layouts that can be utilised in hand-written source code.

PySide Language Bindings. The Qt library is written in pure C++. In order to be able to use it from within Python, so-called language bindings need to be used. Qt's popularity led to the development of bindings for almost all modern programming languages with Python being no exception.
Up to 2010, the only available Python binding for Qt was the PyQt library. BioPARKIN was built atop PyQt at first but has since been moved ontop the newer PySide binding library.[9]
The most obvious advantage of PySide is its use of the liberal LGPL—the GNU Lesser General Public License.[10] This license allows for the use of software built atop PySide in both free/open-source and commercial/closed-source use cases. PyQt, on the other hand, is based on the more restrictive GPL[11] (without the preceding "Lesser") which does not permit deployment in commercial closed-source software.

6.3.2 Design Patterns

This section illustrates a number of software design principles and patterns that have been used in BioPARKIN. See Section 6.3.4 for information about the actual implementation of some of BioPARKIN's features, or look up the developer documentation for more details.

[6] The developer documentation will be available at http://www.zib.de/en/numerik/csb/software/bioparkin.html
[7] http://www.python.org
[8] http://qt.nokia.com/
[9] http://www.pyside.org
[10] http://www.gnu.org/licenses/lgpl.html
[11] *http://www.gnu.org/licenses/gpl.html*

Model-View-Controller Paradigm. The Model-View-Controller (MVC) paradigm (often referred to as an architectural pattern) is mostly drawn upon when designing software that has a graphical user interface (GUI). The idea of MVC and all its variants is to decouple the visual representation of the software—the GUI—from the computational logic—often called *business* logic. MVC is a very general paradigm—and a de-facto standard in GUI programming—and often interpreted in slightly different ways. The following interpretation is the one we use in the context of BioPARKIN.

Model
: Comprises all the data that is encapsulated by the program—in case of BioPARKIN this includes the actual SBML model as well as all the current settings, the loaded experimental data and so on. The Model is connected to the actual data source—e.g. files, databases—and can also contain business logic, most often very data-centric methods.

View
: Represents the user interface. It can contain some logic for (simple) input validation but no complex business logic.

Controller
: Represents the gateway between View and Model and can contain (parts of) the business logic.

Simple approaches to build a software with a GUI often intermingle interface and logic code within the same code structures—e.g. files and/or classes. Most of the time, this tight coupling is unfortunate because changes in the business logic obviously entail changes in the GUI parts of the code. It also makes it difficult for teams of developers and designers to work together on such parts of the program.
Realising the MVC approach, decoupling of code leads to better collaboration possibilities, code with clearer structure that is easier to modify and understand, and often to less error-prone software (in part because decoupling facilitates automated testing of individual software parts).

MVC and Qt. BioPARKIN's GUI is based on Qt which provides an excellent tool—Qt Designer—to build user interfaces. By using Qt Designer, BioPARKIN's UI is defined in its own set of `.ui` files. For use within Python, these .ui files are converted to actual Python code. This Python code is never changed by hand but has to be recreated if the corresponding `.ui` file is changed.
In BioPARKIN these Python classes are used by inheriting from them. Another equally valid approach would be to instantiate each Python UI class on its own and use the UI objects from within the hand-written business logic code.
Either way, the distinction between `.ui` files containing the UI definition and hand-written Python files implement the V and C in MVC.
The separation of the Model is realised by having individual classes to encapsulate all of the data-related parts of BioPARKIN.

Events. Whenever different parts of a software system need to talk to each other, events are commonly used—resulting in the so-called event-based asynchronous pattern if used in conjunction with multi-threading (see next section). This is especially true for software that has a graphical user interface (GUI). Parts of the GUI often reflect the current state of the program. Whenever the state changes, the GUI should reflect this. Using events for this behaviour is more elegant and less error-prone than hardwiring the state-changing object to the GUI.
In Qt, the event system is implemented using Signals and Slots. Classes can define Signals which can be subscribed to from other classes (although, signaling within the same class is possible and often makes sense, too). The subscribing class defines a Slot method which is called whenever the associated Signal is emitted.
One of the most useful aspects about Qt's event system is that it safely works across threads. A Signal-emitting object can live within another thread than the subscriber (whose Slot should be executed). This feature is absolutely essential when working with GUI-based software where GUI and worker threads should almost always be kept separate (see Section 6.3.2 on multi-threading).

Signals and Slots are used throughout BioPARKIN. The MVC architecture (see Section 6.3.2), for example, is based on events in large parts: model classes emit Signals so that all subscribed Views (such as table views) know when to update themselves (thereby implementing the so-called observer pattern).

Multi-Threading. In current operating systems, each individual software is commonly executed within its own process. Each process in turn, can consist of one or more threads. Threads are handled by the operating system and can run (virtually) concurrently on the CPU.
GUI-based software almost always uses at least two threads. One for the GUI and another where the "heavy lifting" is done. The idea behind this separation is to keep the UI responsive (e.g. correctly displaying the current state of the program) even when the software is doing something computationally expensive.
Qt provides a QThread class which works in conjunction with Qt's Signals and Slots event system (see Section 6.3.2). BioPARKIN extends QThread and provides an intermediate "abstract" class[12] to readily report progress from within inheriting classes. These inheriting classes can either report progress while they do their computations or they can just give the information that they started and stopped. In the first case, the GUI will display a progress bar. In the latter case, a so-called "throbber" animation will inform the user that a computation is happening. Progress information is passed to the UI using Signals and Slots.
Within BioPARKIN most (but not yet all) classes that do non-UI-related work are based upon this progress-enabled threading system. Most importantly, all computations delegated to PARKINcpp are wrapped this way.[13]

Services. Services—sometimes also called Service Providers—are classes that provide well-defined functionality to different parts (i.e. other classes) of a program that is managed centrally (i.e. by only one object in memory).[14]
In BioPARKIN, a number of services exist—e.g. for accessing program-wide options, for accessing the main window's status bar and for displaying messages in the warnings window. The most widely used service is needed to access data (both simulated and experimental). By this construction, different parts of the program can seamlessly work together on data. For example, the Data Browser can load experimental data into memory while another class can feed these data into PARKINcpp to identify parameters.
Services should be available (programatically) from anywhere in the codebase. In order to do so without passing the same references to every object, BioPARKIN uses a singleton approach (see Listing 9). A singleton is a class that returns the same instance whenever it is instantiated. This means only one copy of this class can exist in memory at the same time. A singleton service object that is created once (e.g. in the main `bioparkin.py` file, the entry point of BioPARKIN) can easily be accessed by other parts of the code that are invoked later. Whenever a service is about to be used, it is just instantiated as a new object. The programmer does not have to consider or know whether an instance of the service already exists or not.[15]

[12] In this case "abstract" means a class that should not be instantiated directly. Python has no support for "real" abstract classes.

[13] Currently, PARKINcpp does not report incremental progress information to BioPARKIN, so there is no way to display a progress bar when computing sensitivities, etc. The throbber animation is displayed instead.

[14] This is often combined with Service Locators to completely separate service provider classes from consumer classes, because neither one has references to the other. This more elaborate approach is not used within BioPARKIN. In BioPARKIN, the consumers have direct "knowledge" of the providers.

[15] Other languages (like the strongly typed C#) have a type of class that is static. These classes can be accessed without being instantiated and are often used to define services. Python has no concept of static classes, thus the singleton pattern is used here.

```
1  class DataService(QObject):
2    """ A data manager for accessing experimental
3    data and simulation data results.
4    The service can read data files and provide
5    the data in a convenient data structure.
6    The __new__ method is overridden to make
7    this class a singleton. Every time it
8    is instantiated somewhere in code,
9    the same instance will be returned.
10   In this way, it can serve like a static class.
11   """
12
13   _instance = None
14
15   # making this a Singleton, always returns the same instance
16   def __new__(cls, *args, **kwargs):
17     if not cls._instance:
18       cls._instance = super(DataService, cls).__new__(cls, *args, **
             kwargs)
19     return cls._instance
20
21   def __init__(self):
22     # only create on first init
23     if not hasattr(self, "data"):
24       super(DataService, self).__init__()
25       self.data = None
```

Listing 9: Part of the DataService class to illustrate overriding the `__new__()` method to create a singleton.

6.3.3 Project Structure

The following is a brief overview of the file and folder structure of BioPARKIN. For more details refer to the developer manual [3].

Python Modules and Packages. In Python terminology, every standard `.py` file is a module (which contains one or more classes).
Python packages contain modules that are self-sufficient and provide a certain functionality. Such packages can be imported by other code parts and act like a library. Done right, this can make code reusable within a project as well as across projects.
A Python package is just a folder with any number of Python files/modules (providing the functionality of the package) and a special `__init__.py` file in it. This file is most often empty (it is empty in all of BioPARKIN's packages) but can define some additional importing behaviour if necessary.

Files and Folders. The following list outlines the most important packages within BioPARKIN's code base and may serve as a rough impression of where to search for code providing a specific feature.

Root folder
: Most importantly, BioPARKIN's root folder contains the BioPARKIN.py file that is used to start the program. Within this file, the main window and important Signal and Slot connections are initialised.

`backend`
: Provides base classes that can be used to build a BioPARKIN-compatible computation backend. Details are provided in Section 6.3.4.

`backend_parkincpp`
: Inheriting from the backend package, the actual connection with the PARKINcpp library is realised here. Details are provided in Section 6.3.4.

`basics`
: Contains the threading base class and other sub-packages.

  `helpers`
  : A small collection of helper classes to deal with enumerations, files and some SBML-related exception cases are defined here.

  `logging`
  : Logging is integrated throghout all parts of BioPARKIN's code base. The necessary logging adapters for Python's own logging system are located here. For more details, see Section 6.3.4.

  `widgets`
  : Shared code for custom Qt widgets is placed here.

`contrib`
: A place for 3rd-party code snippets (but not packages currently) that are used in BioPARKIN.

`datamanagement`
: The backbone of BioPARKIN's data handling, the `DataSet` class and the `EntityData` class, are defined here. More details are provided in Section 6.3.4.

`images`
: This is not a Python package. Instead, this folder contains icons and images for use in BioPARKIN.

`odehandling`
: Generation and management of ODE systems (from a SBML reaction network) is provided by this package. For details, see Section 6.3.4.

`parkincpp`
: This package contains the compiled PARKINcpp library (`_parkin.pyd` on Windows, `_parkin.so` on Linux) and the necessary SWIG-generated parkin.py file to access this library (see Section 5.2).

`sbml_model`
: Loads an SBML model file and wraps every SBML entity into helper objects. Parameter Sets are also handled here.

`sbml_views`
: Views used in BioPARKIN's main window are located here (model view, entity tree view, entity detail view). The SBML warning window can also be found in this package.

`services`
: Holds the services of BioPARKIN, most notably, the data service (see Section 6.3.2 and Section 6.3.4).

`simulationworkbench`
: This is a large and important package that defines the UI of the Simulation Workbench (shown within the Workbench tab of the main window) as well as all the functionality to gather settings, SBML entities, parameter sets, and experimental data, in order to pass these to the computation backend (e.g. PARKINcpp).

`widgets`
: The Results window and all custom widgets that are used within the Results window are found within this sub package—e.g. plot widget, table widget. The Data Browser widget and the Timepoint Chooser window can also be found here.

6.3.4 Implementation Details

This section illustrates some of the decisions that were made in the process of designing the code architecture of BioPARKIN.

Logging. Python provides a mature logging functionality that is leveraged in BioPARKIN. Logging libraries usually offer different logging levels to provide graduated levels of detail for different logging targets. For example, the console may show only the "info" and "error" logging levels while a log file may also contain log entries of "warning" and "debug" levels.
The logging targets (such as console and files) provided by Python's logging module can be extended by inheriting from its `logging.handler` class.
BioPARKIN defines two such logging handlers:

`QtLoggingHandler`
: This handler can connect to a compatible Qt widget to display logging information within that widget. BioPARKIN provides the `QtLoggingView` widget which uses a plain text edit field to show logging information. Currently, neither the `QtLoggingHandler` nor the `QtLoggingView` are in use within BioPARKIN's UI. (The implications on usability were greater than had been anticipacted. Users were constantly irritated by the prominently placed stream of status information.)

`StatusBarLoggingHandler`
: Furnished with a `QStatusBar`, this handler accepts log messages that are shown on the status bar. Note that the logging level for this handler should be set appropriately, e.g. to "info", so that not all possible logging output is shown on the status bar.

BioPARKIN logs to three different targets:

1. The console window. By keeping an eye on this window, the user gets additional information about what is happening within BioPARKIN.
2. So-called rotating log files. This means there is not a single log file, but six of them. The main log file is named `bioparkin_log.txt`. The additional files are named `bioparkin_log.txt.1` to `bioparkin_log.txt.5`, and serve as a backup of older log entries. The newest entries are always found in the main `.txt` file.
3. The status bar which only shows log entries of the "info" level.

Defining logging statements (with appropriate levels) throughout BioPARKIN's code in this manner is an efficient way to keep the user properly informed—let it be in detail via the log files and console, or let it be the more usual status information found on the status bar.
Note that BioPARKIN has a command line switch "`--debug`" to enable debugging output on the console and within the log files. Starting BioPARKIN without this switch, debugging entries will not show up at all.

Data Handling. All data processed by BioPARKIN—whether it is experimental data loaded from a file, or simulated data from a computation—is handled by the `DataService` object (for information about services, see Section 6.3.2).
The idea behind this service is to have one single point of access for all data-related queries. Data in BioPARKIN is stored in a two-tiered class structure with `DataSet` and nested `EntityData` objects.

`DataSet`
: The `DataSet` is meant to logically combine all data that can be associated with one set of data. For example, data from one file will be hold by one data set. Also, the results from one computation run will be put into one data set. Data sets have different types:

`SIMULATION` For all simulated timeline data.

`EXPERIMENTAL` For experimental timeline data.

`SENSITIVITY_DETAILS_SUBCONDITION` For sensitivity results with subcondition data.

`SENSITIVITY_DETAILS_JACOBIAN` For sensitivity results with Jacobian matrices data.

`SENSITIVITY_OVERVIEW` For sensitivity overview result data.

`ESTIMATED_PARAMS` For estimated parameter values.

`DataServices`
: The `DataService` object delivers the correct data according to these type if they are requested by other parts of the program. For each data type there exists a corresponding getter method, e.g. `get_simulation_data()`, or `get_estimated_param_data()`.
The actual data in a data set are held by `EntityData` objects which, in turn, are stored as an `OrderedDict`. The key for each `EntityData` within this dictionary is the associated SBML entity whenever possible. Sometimes, however, there is no SBML entity to associate data with. In such instances, the keys can be strings. The `helpers.sbmlhelpers` package has a method that tries to find an SBML entity within a given model based on the entity's ID.

`EntityData`
: An `EntityData` object holds all data of one entity with regard to an experiment or computation. This entity can be an SBML entity (and almost always is) but does not have to be.
Each `EntityData` instance holds a list of `DataDescriptor` and of `DataPoint` entries. The actual meaning of these lists depends on what the `EntityData` represents.
When dealing with timeline data (both experimental and simulated), each `EntityData` object represents one Species and its associated data. In this case, the `DataDescriptor` list is a list of timepoints (i.e. each entry in the `DataPoint` list "describes" one timepoint).
If data are not timepoint-based, the use of the very general "descriptor" term becomes clear. For example, in the case of Jacobian matrix sensitivity data, there is one `DataSet` per matrix. Within each `DataSet` there is one `EntityData` object per parameter. Thus, we look at sensitivities from a parameter-based point of view (the same is true for sensitivity overview and subcondition data). Now, the `DataDescriptor` list holds the IDs of Species. So, each entry in the `DataPoint` list corresponds to the sensitivity value of one species (with respect to the parameter the current `EntityData` is representing). Values and species are matched by list indices.

ODE Generation and Handling. In most cases SBML models do not define ODEs (Ordinary Differential Equations) directly.[16] Connections between the entities in the model are defined by reactions and kinetic laws (given by mathematic formulas). Each species may be a reactant or product in any number of reactions. Those reactions' kinetic laws need to be reformulated to one ODE per species.
Basically, this is done by going through all reactions that a species participates in and either add or substract its kinetic law from the ODE term. This is facilitated by the `ODEGenerator` class in the `odehandling` package. This ODE generation also performs some very basic simplification of the resulting right sides of the ODEs. The stoichiometry values of a reaction are also taken into account if any are given.

Handling Assignment Rules. SBML assignment rules can define arbitrary right sides that can be assigned to different types of targets, most importantly to SBML species and parameters. Those assignments need to be evaluated by the integrator at every timepoint.
In order for PARKINcpp to handle those assignment rules, BioPARKIN currently replaces the targets (mainly parameters) within the ODEs where they appear. I.e. targets are effectively replaced if they are defined by an assignment rule. That way, PARKINcpp does not need any special functionality to handle such rules. On the downside, assignment rules are no entity on their own within PARKINcpp and can not be displayed in the results (neither within a data table nor within a plot).

[16] SBML models can define ODEs by Rate Rules. These rules usually define some aspects of the model, but never comprises the whole model as such.

Backend. The computation backend is structured into two important parts:

`BaseBackend`
: The base backend class (in the `backend` package) is a very small "abstract" class that needs to be inherited and implemented by each and every backend that should be usable by BioPARKIN. Currently, there is only the PARKINcpp-based implementation. A FORTRAN-based implementation (generating and compiling FORTRAN code on-the-fly) did exist previously but was removed once PARKINcpp reached mature functionality and stability.

`BaseAstConverterTemplate`
: The idea of the AST[17] converter class is to provide the ability to convert SBML `ASTNode`s into any mathematical representation that is needed by the actual backend. This base class (also found in the `backend` package) provides one "abstract" method for each mathematical symbol that will be interpreted by BioPARKIN, e.g. the methods `handlePlus()`, `handleMinus()`, `handlePower()`.

Embedding PARKINcpp. PARKINcpp offers a Python-compatible interface using the SWIG wrapper library (see Section 5.2). As such, for all wrapped C++ classes there exist Python counterparts which can be used as normal Python classes.
The following list briefly describes the most important classes that are needed to be instantiated (from within Python) to use the PARKINcpp library. More details are given in the developer documentation.

`BioSystem`
: The `BioSystem` class is the main place to describe the model. It has lists or dictionaries of species, parameters, ODEs, events, etc. ODEs are given by using PARKINcpp's `Expression` objects. These objects are created by the actual PARKINcpp-specific implementation of the `BaseAstConverterTemplate` class mentioned above.

`BioProcessor`
: Setting up a computation is done via the `BioProcessor` class. On instantiation, this class gets a reference to a `BioSystem` object so that it is able to access the model at hand. In addition, it includes all settings (a part of which can be changed in the GUI) and is given computation-relevant information such as experimental data and species/parameter thresholds. Once the object is set up, conveniently exposed methods for all different computation types can be called from within Python—i.e. `bioProc.computeModel()`, `bioProc.computeSensitivityTrajectories()` and so on, see Section 5.1).

The API of both `BioSystem` and `BioProcessor` (as well as of other PARKINcpp classes exposed to Python) is kept open to continuous improvement to reflect real-world usage and, thereby, to make it easier to integrate PARKINcpp in other projects. Currently, PARKINcpp is only used by BioPARKIN, but it is a mature library on its own and can readily be integrated in other projects.

7 Numerical Experiments

This section illustrates the use of BioPARKIN and PARKINcpp with actual models. First, two models developed by the Computational Systems Biology group at Zuse Institute Berlin are presented. The third model was obtained from the BioModels database[18], a website with curated SBML models [16].

[17] AST is an abbreviation for Abstract Syntax Tree. Such a tree represents an expression with a certain syntax by its node and edge structure. Thus, in principle, it could hold a mathematical formula as well as an expression of a programming language. For example, the term $a + b$ results in one "+" AST node with two "a" and "b" children nodes.

[18] http://www.ebi.ac.uk/biomodels-main/

	GynCycle	BovCycle	BioModel 008
# Species	73	41	5
# Parameters	133	60	5
# Reactions	66	28	13
BioPARKIN Simulation	4.2s	1.5s	0.74s
BioPARKIN Sensitivity Overview	3min 55s	31s	1.17s
BioPARKIN Sensitivity Details	3min 05s	29s	0.99s
COPASI Simulation	$\approx$ 3.4s	$\approx$ 0.5s	$<$ 0.25s
COPASI Sensitivities	$\approx$ 3min 30s	$\approx$ 7.5s	$<$ 1s

Table 2: Comparing computation times for models of different sizes. Integration interval was 0s to 100s in all cases. Sensitivities were computed for all parameters. For BioPARKIN's Sensitivity Details, three timepoints (10s, 20s, 30s) are selected. All other setting used BioPARKIN default values, e.g. RTOL = 10^{-7}. Times include preparing the ODE system and plotting the results. COPASI run times have been measured by hand.

7.1 GynCycle

7.1.1 Description of the Model

The GynCycle model—developed in parts by the Computational Systems Biology group at Zuse Institute Berlin—presents a differential equation model for the feedback mechanisms between a number of hormones during the female menstrual cycle. In contrast to other models, this model does not involve delay differential equations. The resulting mathematical model describes several hormone profiles throughout the menstrual cycle, and is able to correctly predict changes following administration of single and multiple doses of two different drugs. The model contains 42 species and 114 parameters resulting in an ODE system with 40 equations [23].

7.1.2 BioPARKIN and the Model

The GynCycle model is fairly large. For the runtime measurements, a slight different model is used that includes a coupled PK/PD model describing a variant of the drug administration. It amounts to 73 species, 133 parameters, 66 reactions, and 2 assignment rules. A single forward simulation takes 4.2 seconds on an Intel Core 2 Quad CPU at 2.8 GHz with regard to single core usage.[19]
On the same machine, computing the Sensitivity Overview for all parameter/species pairs takes 3 minutes and 55 seconds. Detailed sensitivities for three timepoints are calculated in 3 minutes and 5 seconds (see Table 2 for comparisons). [20]
Here, BioPARKIN served as a tool to explore the model and its parameter space. Together with its predecessor POEM (an unreleased, in-house tool based on the same numerical principles, see Section 1.2), it was able to develop and to fine-tune a highly descriptive and predictive model for a complex human pathway that has direct relevance to real-world applications [23].

7.2 BovCycle

7.2.1 Description of the Model

Bovine fertility is subject of extensive research in animal sciences, especially because fertility of dairy cows has declined during the last decades. The regulation of estrus is controlled by the complex interplay of various organs and hormones. The Computational Systems Biology group at Zuse Institute Berlin developed a mechanistic mathematical model of the bovine estrous cycle that includes the processes of follicle and corpus luteum development and the key hormones that interact to control these processes. Exploiting this model it is possible to obtain a set of equations and parameters that describes the system consistent with empirical knowledge [2, 24].

[19] Generating the ODE system accounts for 2.9 of the 4.2 seconds. This is currently done for every computation. A future version of BioPARKIN might improve this and compute the ODE system only after the model has changed significantly.

[20] All computations are done using the standard settings of BioPARKIN (e.g. RTOL = 10^{-7}, ATOL = 10^{-9}).

7.2.2 BioPARKIN and the Model

The BovCycle model is smaller than the GynCycle model with 41 species, 60 parameters, and 28 reactions. A single forward simulation takes 1.5 seconds on an Intel Core 2 Quad CPU (at 2.8 GHz) where, again, only a single core is used. Computing the Sensitivity Overview for all parameter/species pairs takes 31 seconds. Detailed sensitivities for three timepoints are calculated in 29 seconds (see Table 2 for comparisons).
In this use case, BioPARKIN enabled the researchers to successively improve the model with each design iteration. Procedures such as parameter estimation and sensitivity analysis proved to be absolutely essential in this context as they guide design decisions by giving insight into hidden dependencies of the model.

7.3 BIOMD008 (Annotated SBML Database)

The model with ID 008 is one of the oldest entries in the BioModels database, and models cell cycle control using a reversibly binding inhibitor [11].
This model is selected here for demonstration purposes because it is small as it comprises only 5 Species, 5 Parameters, 13 Reactions, and 2 Assignment Rules. The smaller a model is, the easier it is to understand why changes in one part of the model (e.g. parameter values) affect computation results.
Albeit being small, nevertheless, the model is of the cell cycle type and, in principle, exhibits a stable limit cyclic which is interesting by itself to look at sensitivity values, etc.
No experimental data have been available for this model. In order to test BioPARKIN's parameter identification, the results of a simulation run are perturbed and used as input.[21]
Last, but not least, a noteworthy, instructive conclusion will be drawn from this model at the end of this section.

7.3.1 Parameter Identification

Key questions of pratical relevance in parameter identification tasks are almost always how much data is sufficient and, even more importantly, how much data is necessary to successfully identify the unknown parameters. In this theoretical scenario, obviously, all possible data of each of the five species are available.
A specific parameter (V3p) is changed (from 0.3 to 1.0), and the goal is to reconstruct the original parameter value. In a sequence of identification runs, each species is selected to be the only species for which data is available.
For three of the five species (M, Y, and Z), the original value of V3p is reconstructed without any difficulties. The parameter identification, however, is not successful at all if one of the other two species (C and X) is selected as data source. Rerun of the identification a $2^{nd}$and 3^{rd} time improves the parameter value but does not correctly identify it, either. Figure 7.1 shows a screenshot of the Parameter Sets tab after these computations.

7.3.2 Sensitivities

For the purpose of this section, only sensitivities with respect to parameter V3p are examined. In the previous section, this parameter is used to test the parameter identification.
The Sensitivity Overview for Biomodel 008 computed by BioPARKIN results in a plot of the sensitivity values of all parameter/species pairs over time. This plot can be filtered to show only the sensitivity trajectories with respect to parameter V3p (see Figure 7.2).
Parameter V3p displays a cyclic sensitivity across all species. It seems that a change in V3p influences the least the time course of species Y and Z while it has more influence on species C, M, and X (see Section 2.5 for details about the stability of the solution).

[21] This is possible if BioPARKIN is started with the command line parameter "`--debug`". In this case, the Settings & Action tab of a Simulation Result table offers the option "Save as Pseudo-Experimental Data". Additionally, the Data Browser displays one more option to perturb data in the Actions tab.

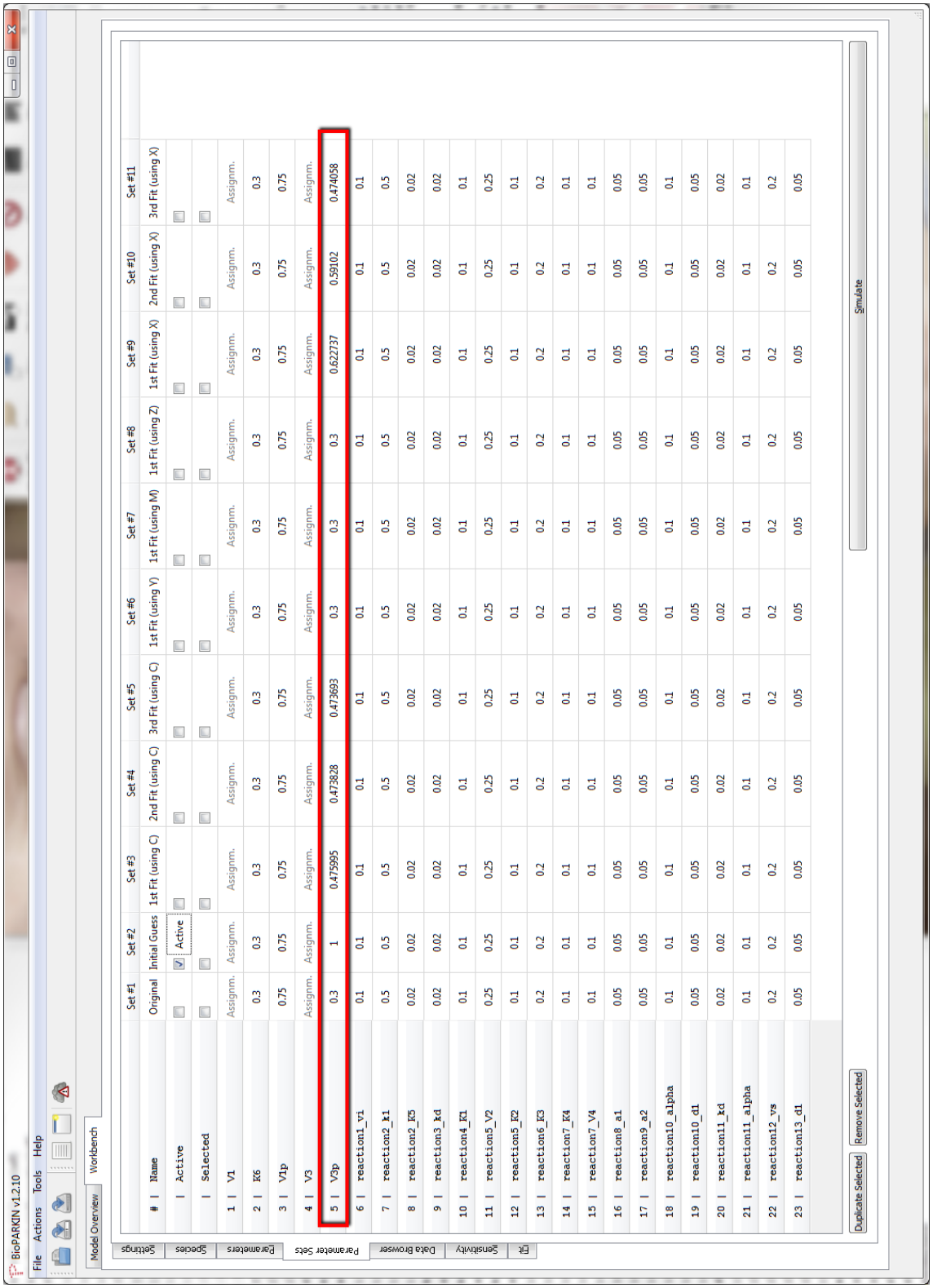


Figure 7.1: A screenshot of BioPARKIN showing the Parameter Sets tab after nine parameter identification runs. The red box was drawn onto the screenshot to highlight the parameter in question (V3p). The two leftmost columns show the original value (0.3) and the value that is used as input for the computations (1.0), respectively. Columns 3 to 11 show the result of the individual species. The species for which data have been available is given in the `Name` row.

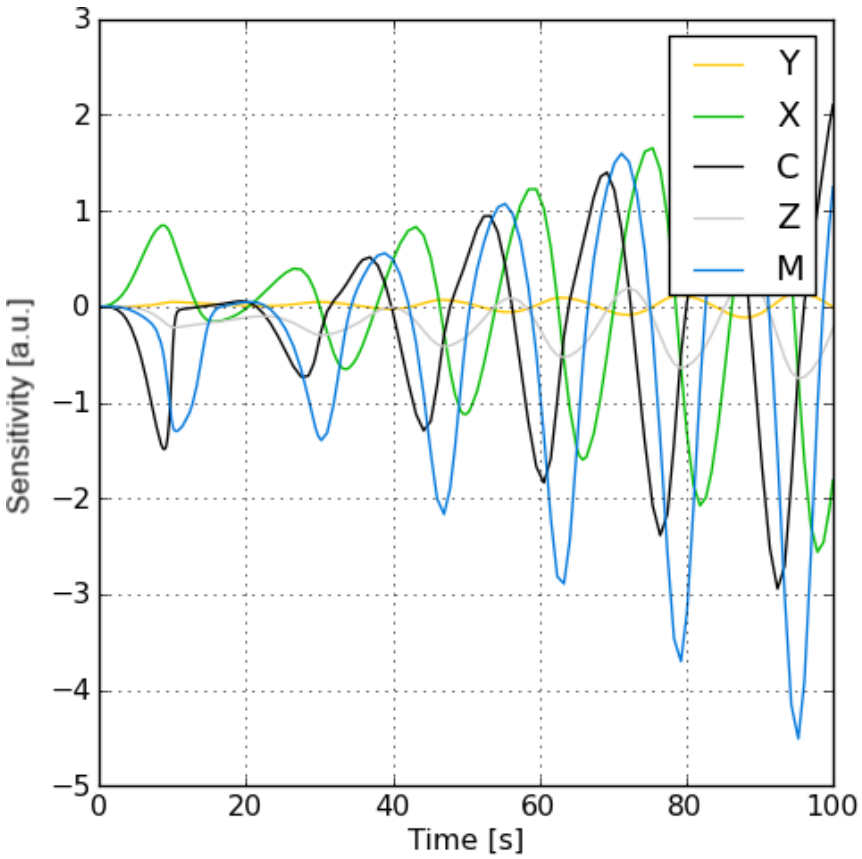


Figure 7.2: Sensitivity Overview of the BioModel 008 for the parameter V3p. Normalised sensitivity trajectories are plotted against the simulated time interval.

7.3.3 A Noteworthy Caveat

Observing the sensitivity trajectories in Figure 7.2, a researcher might anticipate that having experimental data for species Y and Z only will be least preferable to identify parameter values. The actual parameter estimation tests, however, contradict this assumption. If data of the two species with the lowest apparent sensitivity are provided, the V3p parameter value can be determined without a problem. On the other hand, providing data of apparently sensitive species does not necessarily guarantee parameter identifiability (see species C and X).
Key point, here, is that the sensitivity analysis is not always suitable to anticipate which parameters are more likely to be identified than others. In fact, sensitivities highly depend on the actual parameter set and therefore, they are only meaningful *at the end* of an successful identification run. Thus, it really should always be kept in mind that the sensitivity results are merely meant as an explorative *a priori* tool that might aid the researcher to get a better understanding of the model.

8 Outlook

A complex software such as BioPARKIN is really never complete. Some changes can lead to improved usability and/or efficiency. Other changes expand the functionality of the software and make it possible to tackle problems (or aspects of problems) that previously could not be handled by the software.
This section summarises some of the most worthwhile changes to PARKINcpp and BioPARKIN in order to make the workflow of researchers more efficient and to extend the ability to gain insights into models and data.

8.1 Numerical Library

Parallelisation. Computation times could be significantly decreased by harnessing the power of multi-core CPUs. This could be achieved, for example, by providing other matrix implementation packages—needed for the numerical linear algebra work—which especially exploit a multi-core computer architecture. The code base of PARKINcpp is prepared for the addition of such matrix packages (see Section 5.3).

Runtime Optimisation. Another approach to improve computation times could be based on speeding up the right-hand side evaluation of the ODE systems. This could be achieved by incorporating a so-called just-in-time compilation technique to avoid the costly traversal of the expression trees.

Additional Identification and Optimisation Algorithms. In order to provide more features in the numerical library, several other identification and optimisation algorithms could be included, especially global ones. This would, in principle, enable researchers to assess and to classify the identification results of the unknown biological system under investigation even more clearly.

Additional ODE Solvers. PARKINcpp currently uses LIMEX to solve possibly stiff ODE systems. Other numerical ODE solvers might be better suited to, for example, solve delay equations, thus widening the range of models that can be studied using PARKINcpp and BioPARKIN.

8.2 Graphical User Interface

Data Browser. The Data Browser already enables the user to handle different sources (e.g. files) of data. However, data from different sources are currently only to be combined by invoking an external application (such as a spreadsheet software). The Data Browser can be improved by adding a feature that allows the user to merge data from different sources in an ever increasing sophisticated way (e.g. putting only selected information in the resulting data set and managing name collisions).

Improved Data Handling. As the software was used more often in actual day-to-day scenarios, its use cases became clearer. The underlying data handling architecture (see Section 6.3.4) has to be improved in order to better accomodate for complex usages (e.g. comparing data that is generated based on different models).

Results Management. This is closely tied with the previous point (data handling). The management of results has to be improved so that results can be compared more easily across models. In order to make the results reproducible, meta data will be associated with every result, containing all settings and variables that lead to this particular result. BioPARKIN should also be able to save this data and reload it together with all the meta data intact. This not only improves communication between researchers, but also makes it easier for one user to understand what he exactly did some time ago.

Model Editing. BioPARKIN currently does not allow for editing the model structure in a significant way (e.g. adding and removing entities). While changes can be made using other SBML-compatible editors, it would be more efficient to be able to do so inside BioPARKIN. This additional feature will also be accompanied by safeguards and helpers that will check the integrity of a model. For example, when a parameter ID is changed, this change has to be propagated correctly throughout the whole model, including all reactions, rules, etc.

Network View. As the development of BioPARKIN began, providing a visual representation of the model was one of the main goals (including clever filtering techniques for handling very large models, etc.). There exists a basic implementation of such visual network view which is currently disabled. Future efforts can focus on the visual representation to further improve the user's understanding of the model's structure and dependencies.

Handling of Multiple Models. In principle, the code base of BioPARKIN can handle multiple models at the same time. This, however, greatly increases complexity and the potential for bugs. A future version of BioPARKIN will re-enable this feature and make it easier to generate (and compare) data for different models.

9 Conclusion

As a field, systems biology is getting more attention, and is gaining more practitioners around the world every year. With the increased size of the community the importance of establishing standards becomes more pronounced.
BioPARKIN tries to inject mathematical knowledge—attained at the Zuse Institute Berlin in the last 30 to 40 years—into this growing community. Ideally, this knowledge enables researchers to generate meaningful and reliable results even faster. In order to make this possible, BioPARKIN combines a basis of long-standing mathematical principles with compliance to system biology standards, most importantly SBML, and an accessible interface. The SBML format is one of the most important standards in systems biology to facilitate collaboration of researchers at all levels (physicians, biologists, mathematicians, etc.). The interface strives to wrap complicated structures and settings (especially with regard to the numerical backend) into an user-friendly package that can be used correctly by non-specialists (e.g. non-mathematicians).
BioPARKIN is split into two parts—the numerical library in C++ and the graphical user interface (GUI) in Python—in order to achieve several advantages. The crucial, yet computation-intensive numerical algorithms are embedded in an efficient C++ library while the GUI is coded in Python which enables rapid interface changes if needed (e.g. to adapt the user interface to new insights into user behaviour). Another important advantage is the independent availability of the PARKINcpp library for use in other related projects.
Both parts are available under the LPGL which is a flexible open-source license allowing for the use of the software in both open and closed (i.e. commercial) projects.
The numerical core of BioPARKIN, the PARKINcpp library, is based on well-known and long-standing mathematical algorithms that have been actively used in chemical physics since the 1980s,

but that have not been applied to the field of systems biology yet. The successful use of BioPARKIN (and its in-house predecessor POEM) to model and simulate the GynCycle and BovCycle models demonstrates that this approach is valid and promising.
Hence, a new software package has been presented that is available free of charge to the community in order to support and to speed up the development of mathematical models by providing state-of-the-art numerical solutions to common problems in the field of systems biology.

10 Acknowlegdements

The authors want to thank P. Deuflhard for the kind introduction into this topic, and many helpful and excellent discussions. This report is written in sincere rememberance of U. Nowak who sadly deceased in June 2011. Without his sophisticated contributions, too, this work would have been nearly impossible.

References

[1] D. Beazley et al. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pages 129–139, 1996.

[2] H. M. T. Boer, C. Stötzel, S. Röblitz, P. Deuflhard, R. F. Veerkamp, and H. Woelders. A simple mathematical model of the bovine estrous cycle: follicle development and endocrine interactions. Technical Report 10-06, Zuse Institute Berlin, 2010.

[3] Computational Systems Biology Group. *BioPARKIN Release 1.2.0 - User Manual.* Zuse Institute Berlin. Available at: http://www.zib.de/en/numerik/csb/software/bioparkin.html.

[4] A. Cornish-Bowden et al. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[5] E. Demir et al. The BioPAX community standard for pathway data sharing. *Nature Biotechnology*, 28(9):935–942, 2010.

[6] P. Deuflhard. *Newton Methods for Nonlinear Problems – Affine Invariance and Adaptive Algorithms.* Number 35 in Springer Series in Computational Mathematics. Springer, 2004.

[7] P. Deuflhard and U. Nowak. Efficient numerical simulation and identification of large chemical reaction systems. *Berichte der Bunsengesellschaft für physikalische Chemie*, 90(11):940–946, 1986.

[8] P. Deuflhard and U. Nowak. Extrapolation integrators for quasilinear implicit ODEs. In P. Deuflhard and B. Engquist, editors, *Large Scale Scientific Computing*, pages 37–50. Birkhäuser, 1987.

[9] R. Ehrig, U. Nowak, L. Oeverdieck, and P. Deuflhard. Advanced extrapolation methods for large scale differential algebraic problems. *Lecture Notes in Computational Science and Engineering*, 8:233–244, 1999.

[10] A. Funahashi, N. Tanimura, M. Morohashi, and H. Kitano. CellDesigner: a process diagram editor for gene-regulatory and biochemical networks, 2003.

[11] T. Gardner, M. Dolnik, and J. Collins. A theory for controlling cell cycle dynamics using a reversibly binding inhibitor. *Proceedings of the National Academy of Sciences*, 95(24):14190, 1998.

[12] M. Hassenzahl and N. Tractinsky. User experience–a research agenda. *Behaviour & Information Technology*, 25(2):91–97, 2006.

[13] W. Hedley, M. Nelson, D. Bellivant, and P. Nielsen. A short introduction to CellML. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 359(1783):1073, 2001.

[14] W. Hedley, P. Nielsen, and P. Hunter. XML languages for describing biological models and data. *Annals of Biomedical Engineering*, 28, 2000.

[15] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. COPASI - a complex pathway simulator. *Bioinformatics*, 22(24):3067, 2006.

[16] N. Le Novère et al. BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic Acids Research*, 34(suppl 1):D689–D691, 2006.

[17] N. Le Novère et al. The systems biology graphical notation. *Nature Biotechnology*, 27(8):735–741, 2009.

[18] R. Machné, A. Finney, S. Müller, J. Lu, S. Widder, and C. Flamm. The SBML ODE Solver Library: a native API for symbolic and fast numerical analysis of reaction networks. *Bioinformatics*, 22(11):1406, 2006.

[19] J. McCarthy and P. Wright. Technology as experience. *Interactions*, 11(5):42–43, 2004.

[20] P. Mendes. GEPASI: a software package for modelling the dynamics, steady states and control of biochemical and other systems. *Computer applications in the biosciences: CABIOS*, 9(5):563, 1993.

[21] U. Nowak and P. Deuflhard. Towards parameter identification for large chemical reaction systems. In P. Deuflhard and E.Hairer, editors, *Numerical Treatment of Inverse Problems in Differential and Integral Equations*. Birkhäuser, 1983.

[22] U. Nowak and P. Deuflhard. Numerical identification of selected rate constants in large chemical reaction systems. *Applied Numerical Mathematics*, 1(1):59–75, 1985.

[23] S. Röblitz, C. Stötzel, P. Deuflhard, H. M. Jones, D.-O. Azulay, P. van der Graaf, and S. W. Martin. A mathematical model of the human menstrual cycle for the administration of GnRH analogues. Technical Report 11-16, Zuse Institute Berlin, 2011.

[24] C. Stötzel, J. Plöntzke, and S. Röblitz. Advances in modelling of the bovine estrous cycle: Administration of PGF2α. Technical Report 11-17, Zuse Institute Berlin, 2011.

[25] L. Strömbäck and P. Lambrix. Representations of molecular pathways: an evaluation of SBML, PSI MI and BioPAX. *Bioinformatics*, 21(24):4401, 2005.