A robust simulator for physiologically structured population models

A robust simulator for physiologically structured population models

Michiel Van Dyck, Xavier Woot de Trixhe, An Vermeulen and Wim Vanroose

Abstract—A framework to simulate physiologically structured population (PSP) models on high performance compute (HPC) infrastructure is built. Based on the model of a single cell, billions of cells can be simulated in an efficient way, allowing fast simulation of the interaction of an entire organ with other body parts. Through combination of three state-of-the-art algorithms, the simulation time is decreased with multiple orders of magnitude. First: PSP modelling exploits the fact that a lot of the cells behave identically at the same time which results in multiple orders of magnitude speed-up. The second speed-up is achieved by using an unconditionally stable, partial differential equation solver which allows big time-stepping by trading off speed with precision. The third speed-up is due to the fact that the framework is designed with HPC cluster use in mind. The PSP simulator is mathematically derived to have maximal stability. Simulation results are validated and simulation speed and accuracy are measured.

1 Introduction

1.1 Computation

Simulation of an entire organ within the human body remains a computational challenge. Even more, when we simulate the organ behaviour based on the accumulated response of a few billion cells, with each cell response described by a biological model. PSP modelling allows an accelerated simulation of these multi-scale models by representing a number of cells by a representative state variable. Simulation of billions of cells can (in some cases) be speeded up by a few orders of magnitude.

1.2 Simulator result

The simulator constructed in this paper is designed to allow the simulation of a large range of PSP models without any parameter tuning. To achieve this, emphasis was put on the stability of the simulator. The accuracy of the simulation depends mainly on the discretisation level, which in turn is proportional with the number of calculations, and so the computation time. By emphasizing stability, the simulator can return for a very wide range of discretisation levels/ computation time, a stable estimate of the organ simulation.

1.3 Paper outline

First we introduce the PSP model of [4] which will be used to illustrate the different mathematical equations of the PSP model.

In section 3, the method is implemented and the main PSP-solver with its different sub-solvers are analysed. We illustrate the difficulties of using typical ordinary differential equation (ODE) and partial differential equation (PDE) solvers. In 3.2, we explain how these problems are tackled by using implicit ODE solvers and an unconditionally stable PDE solver. For the ODE system, emphasis is put on the different algorithms to solve the system of non-linear equations. For the PDE solver, we explain in 3.5 thoroughly the semi-Lagrangian discretisation method of [12] which we converted here to cope with PSP models. Also boundary conditions, and conservation laws are handled briefly.

In section 4, the numerical simulations results are compared to the solution derived in [4]. We analyse the error in function of the discretisation size. In section 5 we look at some applications. In section 6 we discuss the method. To conclude the paper in section 7.

1.4 Reference PSP Example

To illustrate different properties of a PSP model and the PSP simulator, the PSP model from [4] is used as a reference. In [4], a PSP model is used to study the dynamics of negative viral RNA (-vRNA) within Hepatitis C patients. The model describes the infection of healthy liver cells(Hepatocytes) by Hepatitis C virions. This infection results in the presence of -viral RNA strands in the cell. The cell then produces +viral RNA strands which results in the production of new virions. In turn these virions spread in the liver to infect other cells. The model also simulates the influence of a drug, which interferes with this virion production process. This way, a simulation can be done of how an infected liver can respond on a drug injection. The model describes the liver based on cellular mechanisms (drug-cell-virion interaction) combined with macroscopic models e.g. the injection of a drug in the patient.

The parameters for the mathematical model are: The number of viruses V, the number of healthy cells T, the number of infected cells I, the total amount of viral RNA in the liver R and a drug concentration C. New healthy cells are created at a rate s, but are infected by an infection rate βVT. A term which will return as a source in the distribution of infected cells on the bound. The healthy cells die at a rate $d \cdot T$. Virions are produced by a production factor q multiplied by the total amount of viral RNA in the liver R. The virions die with virion death rate c.

- *M. Van Dyck and W. Vanroose are with the Department of Mathematics - Computer Science, University of Antwerp, Antwerpen, BE 2000. E-mail: see https://www.uantwerpen.be/en/staff/michiel-vandyck/*
- *X. Woot de Trixhe and A. Vermeulen are with Janssen Pharmaceutica, Beerse, BE 2340.*

Manuscript received September 21, 2016; revised August 26, 2016.

The drug concentration C in the body will, from insertion, gradually increase up to a steady state concentration C_{ss} with a rate k.

$$\frac{dY}{dt} = \begin{bmatrix} \frac{dT}{dt} & = & s - \beta \cdot V \cdot T - d \cdot T \\ \frac{dV}{dt} & = & q \cdot R - c \cdot V \\ \frac{dC}{dt} & = & k \cdot (C_{ss} - C) \end{bmatrix} \quad (1)$$

The total amount of viral RNA in the liver R and the total amount of infected cells I is calculated by:

$$\Phi(n) = \begin{bmatrix} R(n) = \int r \cdot n(r,t) dr \\ I(n) = \int 1 \cdot n(r,t) dr \end{bmatrix} \quad (2)$$

Each individual cell in the liver is characterised by r: the number of negative viral RNA strands in the cell. The rate of change of the number of viral RNA strands in the cell is defined by the function g which consist of a term with a production rate α and a term with a risk rate μ. Both quantities change with the drug concentration C in the body.

$$g(x) = \left[\frac{dr}{dt} = \alpha(C) - \mu(C) \cdot r\right] \quad (3)$$

A last parameter $n(r,t)$, represents the amount of cells with r -vRNA strands in the liver, at time-point t. The PDE (eq. 4) models the frequencies of the different types of cells and how they will change over time.

The δ factor describes the death rate of the infected cells. The newly infected cells βVT are cells with no viral RNA yet and are added to the distribution at $n(0,t)$. The initial population of the cells at start is defined by $n_0(r)$.

$$\frac{\partial n(r,t)}{\partial t} + \frac{\partial (g(C,r) \cdot n(r,t))}{\partial r} = -\delta \cdot n(r,t) \quad (4)$$

with

$$g(0) n(0,t) = \beta \cdot V \cdot T \quad (5)$$

$$n(r,0) = n_0(r) \quad (6)$$

More details of this particular example can be found in [4].

1.5 Introducing a generalised PSP model

In general, a PSP model describes a multi-scale model in which, equations modelling the individual level, are coupled with equations modelling the environment level. The interaction between the individual level and the environment level is calculated by means of a distribution. To update this distribution a PDE has to be solved. So the PSP model consists of 3 levels, each defined by its own state:

1.5.1 PSP state variables

1.5.1.1 i-state: The i-state describes the state of an individual, by which the state, in general, is denoted by x. A set of ODE's, g, describe the dynamic behaviour of these state variables (eq. 7). In the reference example, the individual is a cell and the state is characterised by the -vRNA variable r.

$$x = [r]$$

Function $g()$ models the production of viral RNA and incorporates the cellular effects of the drug (eq. 3).

1.5.1.2 p-state: The p-state vector n describes the population of individuals over all possible i-states x. A set of PDE's describe the change of the distribution in i-state and time. The advection term with advection speed $g()$, models the state changes of all the individuals. A risk function $\lambda(Y,x)$ typically represents the birth/death rate of the individuals (eq. 8a).

Boundary conditions of the PDE (eq. 8b) describe the border $\partial\Omega$ of the (PDE) domain Ω. Here we can define for instance the in/out flux off the population along the border. In the example, this represents the infection rate. Newly infected cells are added from the border into the population distribution. $\hat{x}$ is the inward-pointing normal vector on the border $\partial\Omega$ [4], [8].

To simulate a PDE, an initial distribution $n_0(x)$ should be defined to start (eq. 8c). When discretised, this can be a set of (x,n) values, with x gridpoints sampling the domain Ω and n the amount of cells in that particular state.

When the vector x has size 1, the distribution of the population over the different i-states can be represented with a vector n. When x has size > 1, n becomes a matrix or tensor with dimension the number of i-state variables. The size of each dimension equals the number of discretisation points used to represent each i-state variable. In the reference example, x has 1 i-state variable $[r]$, so n is a vector and each entry n_i encodes the amount of cells in the liver which have x_i viral RNA strands in them.

1.5.1.3 e-state: The e-state Y describes the environment. The environment dynamics are described by a set of ODE's f which depend on Y and Φ (eq. 9a). Φ represents the accumulated/macroscopic state of all cells (eq. 9b). To simulate the equations, we also need a set of initial values Y_0 (eq. 9c). The function f describes the interaction between multiple macroscopic variables and the accumulated microscopic responses. But the function f can also describe the interaction of the organ with e.g. another macroscopic entity. The macroscopic state Φ is not calculated by summing all individuals but instead derived from the p-state distribution integrated over all the different i-states (eq. 9b). In the reference example, the total amount of viral RNA in the liver is calculated by integrating the product of the RNA per cell, times, the number of cells that are in that particular i-state (eq. 2, 29a). In the general case, each individual is weighted with a factor $\phi(x)$ which is a function of its i-state x (eq. 9b).

1.5.2 PSP equations

In this section we describe the mathematical equations of a standard PSP model. These equations are used throughout the paper to explain the PSP solver. The arguments of some of the functions are sometimes removed to keep the equations condensed but they should be clear from the context.

- Cell/individual (i-state):

$$\frac{\mathrm{d}x}{\mathrm{d}t} = g(Y,x) \quad (7)$$

- Tissue/population (p-state):

$$\begin{aligned}\frac{\mathrm{d}n}{\mathrm{d}t} &= \frac{\partial n(x,t)}{\partial t} + \nabla_x \cdot (g(Y,x)\, n(x,t)) \\ &= -\lambda(Y,x)\, n(x,t) \qquad (8a)\end{aligned}$$

$$\text{B.C.}: \forall x \in \partial\Omega : g(x)\, n(x,t)\, \hat{\mathrm{x}} = b(Y,x) \qquad (8b)$$

$$\text{Initial values: } n(x,0) = n_0(x) \qquad (8c)$$

- Host/environment (e-state):

$$\frac{\mathrm{d}Y}{\mathrm{d}t} = f(Y, \Phi(n(x,t))) \qquad (9a)$$

$$\Phi(n(x,t)) = \int_\Omega \phi(x)\, n(x,t)\, \mathrm{d}x \qquad (9b)$$

$$\text{Initial values: } Y(t=0) = Y_0 \qquad (9c)$$

2 Constructing the PSP solver

The PSP state vector p concatenates the 3 state variables/vectors Y, n, x (eq. 10). To simulate the PSP model, we have to update the different states of the PSP model over time. We are able to formulate the PSP simulation as an ODE system $\Re$ (eq. 11). We chose to update n at fixed gridpoints x, for this we use the partial derivative $\frac{\partial n}{\partial t}$ from eq. 8a: in the ODE system $\Re$.

$$\mathrm{p} = \begin{bmatrix}\text{Env}\\ \text{Pop}\\ \text{Ind}\end{bmatrix} = \begin{bmatrix}Y\\ n\\ x\end{bmatrix} \qquad (10)$$

$$\dot{\mathrm{p}} = \Re(\mathrm{p}) = \qquad (11)$$

$$\begin{bmatrix}\dot{Y}\\ \dot{n}\\ \dot{x}\end{bmatrix} = \begin{bmatrix}\frac{dY}{dt}\\ \frac{\partial n}{\partial t}\\ \frac{dx}{dt}\end{bmatrix} = \begin{bmatrix}f(Y,\Phi(n))\\ -\nabla_x \cdot (g(Y,x)\, n(x,t)) - \lambda(Y,x)\, n(x,t)\\ g(Y,x)\end{bmatrix}$$

A lot of techniques exist to solve an ODE. A wide range of them can be described by the General linear Method (GLM) [2].

$$\begin{bmatrix}P\\ p_n\end{bmatrix} = \begin{bmatrix}A & U\\ B & V\end{bmatrix}\begin{bmatrix}\Re(P)\\ p_{n-1}\end{bmatrix} \qquad (12)$$

With $p_n = \begin{bmatrix}\mathrm{p}_n, & \mathrm{p}_{n-1}, & \mathrm{p}_{n-2}, & \dots & \mathrm{p}_{n-h}\end{bmatrix}^T$ a vector containing a number of psp states at previous time-steps and $P = \begin{bmatrix}\mathrm{p}^{s_1}, & \mathrm{p}^{s_2}, & \mathrm{p}^{s_3}, & \dots & \mathrm{p}^{s_{sn}}\end{bmatrix}^T$ a vector containing different stage vectors (psp states between 2 time-steps). For example, when choosing $A = 0, U = 1, B = dt, V = 1$ we get the explicit Euler ODE solver:

$$p_n = p_{n-1} + dt\, \Re(p_{n-1}) \qquad (13)$$

when $A = dt, U = 1, B = dt, V = 1$ we get the implicit Euler ODE solver:

$$p_n = p_{n-1} + dt\, \Re(p_n) \qquad (14)$$

The implicit solver has a much wider area of stability than the explicit Euler ODE solver [14]. So to build a PSP simulator with great stability we start from the implicit Euler ODE solver (eq. 14).

$$\begin{bmatrix}Y_n\\ n_n\\ x_n\end{bmatrix} = \begin{bmatrix}Y_{n-1}\\ n_{n-1}\\ x_{n-1}\end{bmatrix} + \begin{bmatrix}dt\, f(Y_n, \Phi(n_n))\\ dt(-\nabla_{x_n} \cdot (g()\, n_n) - \lambda()\, n_n)\\ dt\, g(Y_n, x_n))\end{bmatrix} \qquad (15)$$

From this, we get a system of non-linear equations to solve (eq. 15). To solve this, we use an iterative solver. Different iterative solvers exist, an easy one to implement is the fixed point (f.p.) iteration. Each variable is depicted by a ^{*_i} to indicate the solver iteration, the time step is indicated with the $_n$ subscript. An S matrix is introduced to deal with the partial differential equations. For now we just assume that $S\, n$ is the solution of the advection part of the PDE of eq. 8a. The construction of this S matrix is explained in Section 3.5.1. Each fixed point iteration, we evaluate the following expression:

$$\begin{bmatrix}Y_n^{*i}\\ n_n^{*i}\\ x_n^{*i}\end{bmatrix} = \begin{bmatrix}Y_{n-1}\\ S_n^{*i-1}(x_n^{*i}) \cdot n_{n-1}\\ x_{n-1}\end{bmatrix} + \begin{bmatrix}dt\, f(Y_n^{*i-1}, \Phi())\\ dt\, (-\lambda_n^{*i-1})\, n_n^{*i-1}\\ dt\, g(Y_n^{*i-1}, x_n^{*i-1}))\end{bmatrix} \qquad (16)$$

Each time step, the f.p. iteration is initialised with the solution of the previous time step: $p_n^{*0} = p_{n-1}$. The iteration:

$$p_n^{*i} = p_{n-1} + dt\, \Re(p_n^{*i-1}) \qquad (17)$$

is done until convergence is reached: i.e.

$$\mathcal{C}(p_n^{*i} - p_n^{*i-1}) < c_T \qquad (18)$$

With $\mathcal{C}$ a multivariate function: e.g. $\sum, max$, abs, 2-norm, ... and c_T a certain convergence accuracy threshold.

2.1 Exploiting the PSP structure for solving the non-linear PSP system

Some steps can be performed to increase the convergence speed of the solver. Instead of iterating over the entire system each iteration, we can accelerate the solver by block Jacobian preconditioning [5], [10]: i.e. solving each of the 3 entries of p uncoupled from each other. The convergence rate CR then becomes:

$$\mathrm{CR}_{jac} = (1 - dt\, J_D)^{-1} dt\, J_{LU} \qquad (19)$$

instead of

$$\mathrm{CR}_{fp} = dt\, J_\Re \qquad (20)$$

for the fixed point case. With $J_\Re$ the Jacobian of $\Re$ and $J_\Re = J_D + J_{LU}$ with J_D the block diagonal elements of $J_\Re$ and J_{LU} the lower and upper elements of $J_\Re$. In almost all cases the convergence rate will become much better. We should note that each iteration, 3 systems have to be solved, so each iteration is more costly.

3 General solution scheme

Implementing the previous solver scheme, we arrive at the next solver layout consisting of 5 iteration loops.

- Solve the ODE system g for a number of physiologically relevant individuals: i.e. simulate (all) the different cell states x for time-step dt.
- Solve the PDE system for dt.
- Solve the environment ODE system f for dt.
- Repeat previous steps until convergence of $\Re$
- Repeat previous steps for each time step

For each step in the solver, different algorithms exist: To solve the ODE systems g, f and $\Re$, different ODE solvers can

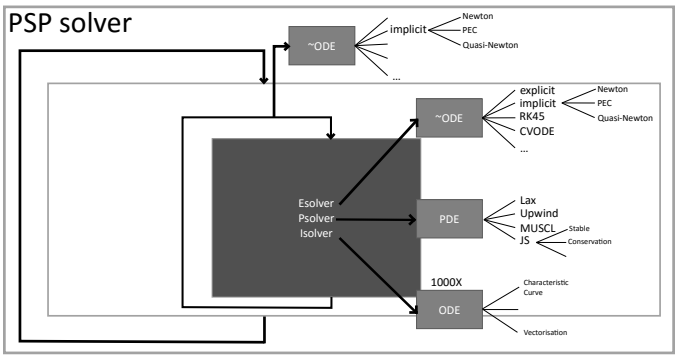

Fig. 1. Schematic of the PSP solver layout.

be used which are reviewed in section 3.1.2. An empirical analysis of these different solvers is made in section 4. For now we chose to solve:

- g: with RK4,
- f: with implicit Euler.
- $\Re$ with implicit Euler.

For the PDE system, also a number of PDE solvers exist. A review of these are made in section 3.1.1.

We chose to use the the Jos Stam modified method of characteristics (JS-MMC) [12], but implemented also other ones for comparison.

In fig. 2 a more detailed schematic overview of the PSP solver is shown, showing the choices in PDE solvers.

A lot of options exist, a trade-off has to be made between speed, accuracy and stability.

Speed and solution accuracy are analysed numerically in section 4. Stability issues of the different solvers are illustrated in the next section.

$p_2 = \mathrm{PSP}_{\mathrm{update}}(p_0, t)$

$\mathrm{PSP}_{\mathrm{update}}$

Loop while: $t_i < t$

$t_i = t_i + dt$

$p_2 = \mathrm{PSP}_{\mathrm{step}}(p_1, dt, t)$

$\mathrm{PSP}_{\mathrm{step}}$

Loop while: $\frac{Y_2 - Y_2^+}{\|Y_2^+\|} > C_t$

$Y_2^{*i-1} = Y_2^i$

$g_2 = g(Y_2^{*i-1}, x, t + dt)$

$n_2 = \mathrm{P}_{\mathrm{run}}(n, Y_2, dt, t)$

$\mathrm{P}_{\mathrm{run}}$

Loop: $n_2 = \mathrm{P}_{\mathrm{step}}(n, Y_2, dt, t)$

$\mathrm{P}_{\mathrm{step}}$

switch:

Explicit	$n_2 = n_1 + dt\, K\, g_1\, n_1$
Lax	$n_2 = Lax(n) + dt\, K\, g_n$
First order upwind	$n_2 = n_1 + dt(K^+ g_n^- + K^- g_n^+) g_1 n_1$
Implicit	$n_2 = n_1 + dt\, K g_2 n_2$
Crank-Nicholson	$n_2 = n_1 + dt/2\, K g_1 n_1 + dt/2 K\, g_2 n_2$
MUSCL	$n_2 = n_1 + dt\, M(dx, n, g, x)$
MUSCL-Runge Kutta	
	$k_1 = M(dx, n, g, x))$
	$k_2 = M(dx, n + dt/2\, k_1, g, x))$
	$k_3 = M(dx, n + dt/2\, k_2, g, x)$
	$k_4 = M(dx, n + dtk_3, g, x)$
	$n_2 = n_1 + dt\, M(dx, n, g, x)$
Stam-MMC	$n_2 = S\, n_1$

$n_2 = n_2 - \lambda n_2$

$\Phi = calc\Phi(n_2)$

$Y_2^{*i} = \mathrm{E}_{\mathrm{run}}(Y_2, dt, Y_1, t, \Phi)$

$\mathrm{E}_{\mathrm{run}}$

while: $\frac{\dot{Y}_2 - Y_2}{Y_2} > C_t$

$\dot{Y}_2 = Y_2$

$Y_2 = Y_1 + dt\, f(Y_2, t + dt)$

Fig. 2. Schematic of the PSP solver layout: M is the Kurganov and Tadmor (KT)-MUSCL scheme update matrix [7], K is a central difference advection discretisation matrix, K^- and (K^+) are the left and right first order upwind finite difference discretisation matrices g_n^+ and g_n^- are $\max(g_n, 0)$ and $\min(g_n, 0)$, S is the sum of 2 interpolation weight matrices constructed by the Jos Stam - modified method of characteristics (JS-MMC) [12].

3.1 Standard solver stability issues

3.1.1 Standard PDE Solver Stability issues

As mentioned before, a lot of PDE solvers exist: e.g. Eulerian discretisation with explicit or implicit Euler update [9], different upwind schemes [9], the Lax method [3] and Monotonic Upstream-Centred Scheme for Conservation Laws (MUSCL) [7]. Also total Variation Diminishing flux limiters [13] can be used to filter numerical oscillations. Another family of PDE solvers is based on the Lagrangian discretisation of the field, e.g. particle methods [6] and smoothed particle hydrodynamics (for flow simulation). There are also PDE solvers based on both i.e. the semi-Lagrangian methods: e.g. the Arbitrary Lagrangian and Eulerian (ALE) methods and the JS-MMC.

The PDE problem the PSP simulator must solve appears to be almost entirely advective. Advection is notoriously difficult to stabilize with common/Eulerian discretised PDE solvers. With these PDE solvers, the maximal dt time step size is bounded by the Courant Friederich Levinson (CFL) number. This constraint states that the CFL should be kept < 1 for most PDE solvers and ≈ 1 for some more advanced solvers.

$$CFL = g(x)\frac{dt}{dx} < 1 \tag{21}$$

With $g(x)$ the advection speed, dt the time- , and dx the space-discretisation size.

Even when these constraints are met, some typical issues arise which are shown in fig. 3. In this figure different PDE solvers are used to solve the PDE of the reference example [4], (4). In fig. 3.a an Eulerian discretisation scheme is used with explicit Euler resulting in uncontrolled oscillations. Implicit Euler updates result in controlled oscillations as shown in fig. 3.b. The Lax method (fig. 3.c) averages out these spatial oscillations resulting in a stable method, but these results are too smooth to accurately represent the moving front. These dispersion effects are also present in different upwind schemes (not shown). More advanced smoothing is done by using flux limiters: higher order methods are used on smooth parts and lower order / more stable methods at the complex parts. In fig. 3.d we see the MUSCL method with flux limiters doing a good job of simulating the PDE in an accurate way without any numerical oscillations. The CFL conditions should be met or the method becomes unstable. To retain stability when using bigger time step-sizes, a Lagrangian or semi-Lagrangian method can be used. The JS-MMCC we chose is described in great detail in section 3.5.1.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2810077, IEEE/ACM Transactions on Computational Biology and Bioinformatics

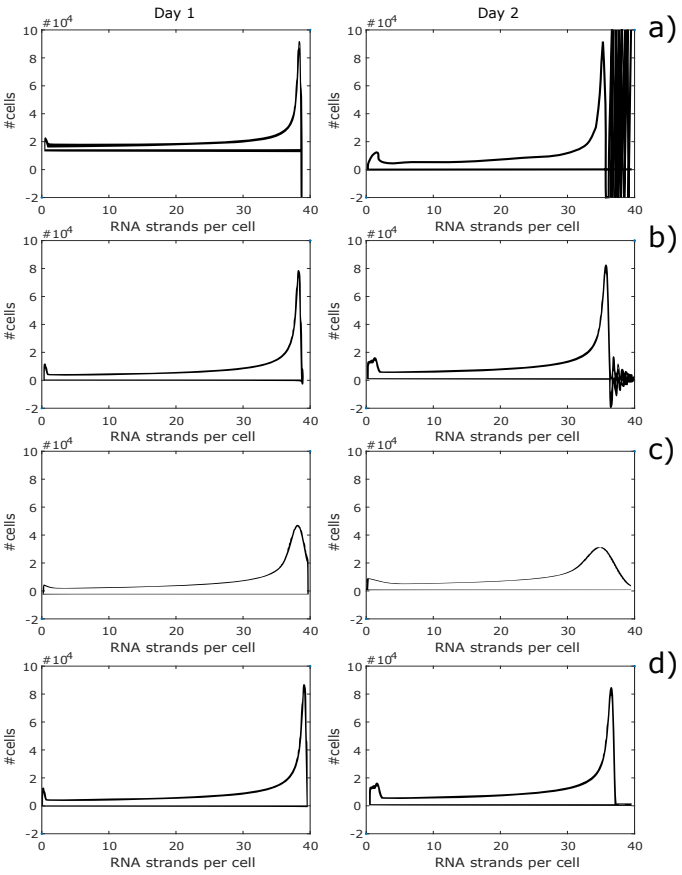

Fig. 3. Different simulation issues while solving an advection problem with different types of PDE solvers. a) Euler, b) implicit Euler c) Lax method d) MUSCL with flux limiters

3.1.2 Standard ODE solver issues

Also for ODE solvers there is a lot of choice: Explicit, implicit or semi-implicit methods. Higher order methods using multiple-stages, multiple-steps or a combination. Also adaptive time-step algorithms can be used with different error estimation algorithms.

When using an explicit ODE solver, the number of calculations for each iteration is fixed and so, when the step-size becomes bigger, the system will return an increasingly more inaccurate value. This results in a divergence/failure of the ODE solver. This is illustrated in (fig. 4: dotted line) for a classic ODE example: $\dot{y} = -4y$. Instead, when the step-size increases, the implicit method loses fidelity but still manages to produces a stable approximation of the real solution. (fig. 4: striped line)

Implicit Euler algorithms are the most stable choice. When accuracy is needed, explicit higher order Runge-Kutta (RK) method might be preferable. For relative stiff ODE equations, higher order implicit solvers can be used e.g. Backward differentiation formula (BDF), Adams-Moulton, singly diagonal implicit Runge Kutta (SDIRK) or IMEX (implicit explicit) methods. For more info, we refer the reader to the ODE solver literature regarding initial value problems. [1], [14]

When using an implicit system, a stable method to solve the non-linear system of equations must be found. For this a fixed point iteration can be used. When the Jacobian can be approximated, a quasi-Newton method can be used, which will typically converge faster. These two methods to solve a system of non-linear equations are evaluated in the next section.

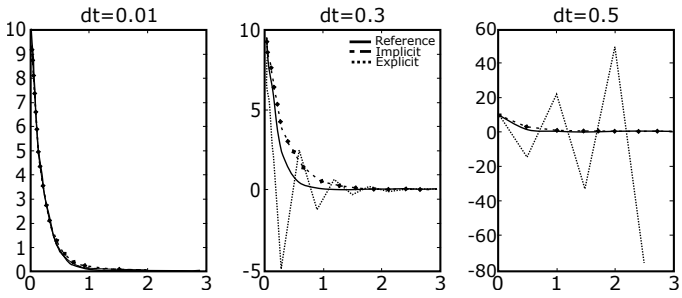

Fig. 4. Implicit vs explicit ODE solver approximation of the real solution. We see that the explicit approximation diverges when the step-size is increased. The implicit method loses fidelity when the step-size increases but manages to produce a stable approximation of the real solution.

3.2 PSP solver

For solving the PSP system with a block Jacobi solver, at each iteration, the different sub-problems: solving environment y, solving tissue n and solving cellstate x should be solved. Next, we will explain the solver for each of these sub-problems.

3.3 Environment ODE solver

The environment model Y can be solved with any solver which can solve an initial value problem. For stability implicit solvers are preferred. To solve the non-linear system of implicit ODE solvers, a fixed point iteration or quasi-Newton solver can be used.

3.3.1 Implicit Euler with Fixed point iteration

$$y_n^{*1} = y_{n-1} + dtf(y_{n-1}) \tag{22a}$$
$$y_n^{*2} = y_{n-1} + dtf(y_n^{*1}) \tag{22b}$$
$$y_n^{*3} = y_{n-1} + dtf(y_n^{*2}) \tag{22c}$$
$$\ldots \tag{22d}$$
$$\text{Until: } (y_n^{*i+1} - y_n^{*i}) < C_T \tag{22e}$$

With C_T a user defined threshold value. We note here that the number of iterations does not directly increase the accuracy of the solution (As opposed to Runge-Kutta (RK) methods where each sub-step, the order of the quadrature approximation is increased.), but these iterations guarantee the stability of the algorithm. We start the iteration with the 1st order explicit Euler update rule 22a (which has minimal stability), each iteration we converge more towards the implicit Euler approximation (which has maximal stability). We clearly trade speed for stability here.

3.3.2 Implicit Euler with Newton method

We can also use a quasi-Newton solver which in general will converge faster. Starting from eq. 22a and 22b, we want to find the roots of h.

$$h(y_n) = y_n - y_{n-1} - dtf(y_n) \tag{23}$$

To get an approximation of the gradient, when y is a vector, we need to solve G from the following equation:

$$G_n \cdot \Delta y = \Delta h \tag{24}$$

But this system is under-determined, so a regularisation constraint needs to be placed to be able to solve it. Broyden's constraint is to minimize the change with the previous estimate of G (in Frobenius norm). The Broyden update function for G is:

$$G^{*i} = G^{*i-1} + \frac{\Delta h - G^{*i-1}\Delta y}{\|\Delta y\|^2}\Delta y \tag{25}$$

When G is known, we solve the linear eq. 26 for $(y_n - y_{n-1})^{*i}$.

$$G_n \cdot (y_n - y_{n-1})^{*i} = h(y_n^{*i-1}) - h(y_{n-1}) \tag{26}$$

Then we update y: $y_n^{*i} = y_{n-1} + (y_n - y_{n-1})^{*i}$

We repeat until the residual is smaller than some user defined threshold value c_T:

$$(y_n^{*i} - y_n^{*i-1}) < c_T \tag{27}$$

Other quasi-Newton methods can be used as well e.g. Broyden–Fletcher–Goldfarb–Shanno (BFGS), which has a direct update scheme to update G^{-1} avoiding the linear system solve in eq. 26.

3.4 The Φ state

The Φ variables are the macroscopic/environment values representing the combined effect of all the individual cells together. They are defined as:

$$\Phi(n) = \int_\Omega \phi(x) n(x,t) dx \tag{28}$$

We approximate the integration numerically by a finite summation. In the reference example of [4], the 2 macroscopic variables to evaluate the environment dynamics are: the total amount of -viral RNA strands, R, and the total amount of infected cells, I. These values are calculated as:

$$R = \int_{x_{\min}}^{x_{max}} r(x) n(x,t) dx \approx \sum_{x_{min}}^{x_{max}} r \cdot n \tag{29a}$$

$$I = \int_{x_{\min}}^{x_{max}} 1 n(x,t) dx \approx \sum_{x_{min}}^{x_{max}} n \tag{29b}$$

Other quadrature methods exist. A first order approximation has the advantage that it makes sure that the conservation laws are respected.

3.5 Population state PDE solver

The PDE we have to solve to calculate the distribution of the p-state over the different i-states is :

Tissue (p-state):

$$\frac{\partial n(x,t)}{\partial t} + \nabla_x \cdot (g(Y,x)\, n(x,t)) = -\lambda(Y,x)\, n(x,t) \tag{30a}$$

$$\text{Boundary conditions: } g(x)\, n(x,t)\, \hat{\mathrm{x}} = b(Y,x) \tag{30b}$$

$$\text{Initial conditions: } n(x,0) = n_0(x) \tag{30c}$$

3.5.1 Semi-Lagrangian method

We solve the PDE as described by Stam [12], this PDE solver is unconditionally stable, fully conservative, and uses a semi-Lagrangian discretisation scheme. It is also known as the modified method of characteristics. This method results in an interpolation matrix S which is used to update the distribution n for the advection part by multiplying it with the previous distribution.

$$n_n = S(x) \cdot n_{n-1} \tag{31}$$

3.5.2 The S matrix

The S matrix is built by combining two interpolation matrices (eq. 32). The interpolation matrices are built by calculating the forward and backward characteristic curves at all the different spatial discretisation points x_k. (fig. 5) The characteristic curve for each of these points is constructed by integrating the intra-cellular model g with an ODE solver.

$$S \sim W_b + W_f \tag{32}$$

$$W_b \simeq \begin{bmatrix} (1-w_1) & (w_1)_{(i_1,1)} & 0 & \cdots \\ & \cdots & & \cdots \\ \cdots & (1-w_k)_{(i_k-1,k)} & (w_k)_{(i_k,k)} & \cdots \\ & & \ddots & \cdots \\ 0 & 0 & \cdots & w_K \end{bmatrix} \tag{33}$$

$$W_f \simeq \begin{bmatrix} (1-v_1) & & \vdots & \cdots \\ (v_1) & \cdots & (1-v_k)_{(k,i_k-1)} & \cdots \\ & & (v_k)_{(k,i_k)} & \\ & & \vdots & \\ 0 & 0 & \cdots & v_K \end{bmatrix} \tag{34}$$

The W_b matrix stores the interpolation weights from the backward characteristic curves. i.e. the characteristic curves which arrive at all the grid points x (fig.5.1). The W_f matrix stores the interpolation weights from the forward characteristic curves: i.e. the characteristic curves which start at all the grid points x. (See fig.5.3)

Calculation of the characteristic curves for each grid point $x(k)$ can be done with any ODE solver. Depending the stiffness and desired accuracy, one ODE solver is better suited than the other. For an explicit Euler solver this becomes:

$$\forall k \in [1,K] : x_b(k) = x(k) - dt\, g(Y) \tag{35}$$

$$\forall k \in [1,K] : x_f(k) = x(k) + dt\, g(Y) \tag{36}$$

The General linear method allows to generalise over a broad family of ODE solvers.

$$\forall k \in [1,K] : \begin{bmatrix} X \\ x_b(k) \end{bmatrix} = \begin{bmatrix} A & U \\ B & V \end{bmatrix} \begin{bmatrix} -g(Y,X) \\ x(k) \end{bmatrix} \tag{37}$$

$$\forall k \in [1,K] : \begin{bmatrix} X \\ x_f(k) \end{bmatrix} = \begin{bmatrix} A & U \\ B & V \end{bmatrix} \begin{bmatrix} g(Y,X) \\ x(k) \end{bmatrix} \tag{38}$$

with K the number of grid-points and $dx = \frac{x_{\max} - x_{\min}}{K}$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2810077, IEEE/ACM Transactions on Computational Biology and Bioinformatics

For all endpoints of all the characteristic curves, we calculate the relative distance w_k to the 2 nearest grid points: $x(i_k)$ and $x(i_k + 1)$. i_k denoting the grid index. When the grid is monotone and uniform, i_k can be calculated with eq. 39: i.e. flooring the division of the distance by the grid-size.

$$\forall k \in [1, K] : i_k = \lfloor \frac{x_b(k) - x_{\min}}{dx} \rfloor + 1, \tag{39}$$

In case of a non-uniform but monotone grid the index can be looked up by a binary search algorithm in $\approx O(\log_2(\#\text{ grid-points}))$ steps. i_k Defines the position of w_k in W_b, the interpolation value of w_k is calculated by:

$$w_k = \frac{x_b(k) - x(i_k)}{dx} \tag{40}$$

To resolve problems at the boundaries we set: (assuming 1 the first array index)

$$\text{Left bound: } \forall i_k < 1 => i_k = 1 \text{ and } x_b(k) = x(1) \tag{41}$$

$$\text{Right bound: } \forall i_k > K => i_k = K \text{ and } x_b(k) = x(K) + dx \tag{42}$$

We do the same for the forward integrated grid points.

$$\forall k \in [1, K] : i_k = \lfloor \frac{x_f(k) - x_{\min}}{dx} \rfloor + 1, \tag{43}$$

$$\forall i_k < 1 => i_k = 1, x_f(k) = x(1) \tag{44}$$

$$\forall i_k > K => i_k = K, x_f(k) = x(K) + dx \tag{45}$$

$$v_2 = \frac{x_f(k) - x(i_k)}{dx} \tag{46}$$

As denoted in 33 and 34, w_k and v_k define the weights in the W_b and W_f matrix. i_k defines the column or row position respectively. We have to make sure the conservation laws in the PDE are respected. For this, 2 corrections are added.

If s_c, the sum of the elements of a column of W_b, is bigger than 1, the entries for that column are normalized to sum to 1. In the other case, if $s_c <= 1$, no correction is made to w_b.

$$s_c = \left[1, 1, \ldots 1\right] W_b \tag{47}$$

$$W_b^* = W_b \times \text{diag}(1/\max(1, s_c)) \tag{48}$$

The second correction is to make sure all points are advected. For each column in W_f, when $s_c < 1$, all elements in the column are multiplied with $1 - s_c$. In the other case, if $s_c >= 1$, the elements of that column in W_f is set zero.

$$W_f^* = W_f \times \text{diag}(\max(1 - s_c, 0)) \tag{49}$$

The S matrix, solving the PDE, is created by accumulating these 2 matrices together:

$$S = W_b^* + W_f^* \tag{50}$$

As long as Y does not change, also $g(Y, x)$, i.e. the characteristic curve functions don't change and so the S matrix can be reused. To solve the PDE again we just have to multiply this S matrix with the previous cell distribution again. In most cases, Y will change and the matrix will have to be rebuilt.

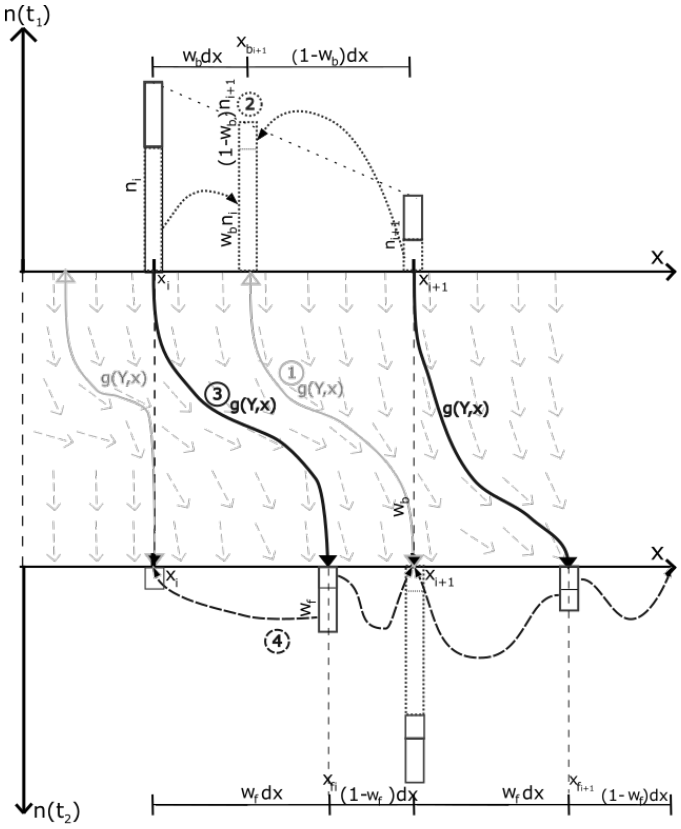

Fig. 5. Schematic representation of the different steps to construct the interpolation matrix to solve the PDE based on the modified method of characteristics. [12] [15].

3.5.3 Sources, sinks and boundaries

While the advection term gives a change in the shape of the distribution, there are also source and sinks terms which change the amount of cells in the distribution. These are captured in the $\lambda(Y, x)n$ term. The λ term is solved with an ODE solver: e.g. explicit Euler

$$n_n^{*i+1} = n_n^{*i} - \lambda(Y, x)\, n_n^{*i} \tag{51}$$

In the example model, λ is a constant $-\delta$ and accounts for the death of infected cells (eq. 4). But this $\lambda(Y, x)$ also allows for a very wide range of other models with dynamic sources and sinks on different places in the domain. To guarantee stability of the solver in these more difficult cases, an implicit Euler solver is advised.

$$n_n^{*i+1} = n_n^{*i} - \lambda(Y, x)\, n_n^{*i+1} \tag{52}$$

Sources and sinks at the bounds are typically integrated within the boundary conditions, but work the same. In case of the Hepatitis C example from [4], we can also see how it also allows to model the exchange from cells from the environment level to the distribution. The boundary conditions at $x = 0$ models the amount of cells that become infected each time step: i.e. the flux at the boundary is defined. The influx is integrated over the time step dt (eq. 53): In the example this result is the number of newly infected hepatocytes (eq. 5) and these are accumulated onto the first point of the physiologically structured population

distribution. Because these cells are in the model also subtracted from the healthy cells environment variable T (eq. 1), mass balance is achieved. Next the PDE update is done. Due to the Lagrangian update and the conservation law corrections, these new cells are advected correctly towards their new physiological state.

$$n(0)_{n+1} = n(0)_n + \int_0^{dt} b(Y, x)dt \tag{53}$$

3.6 The intra cellular model

The intra cellular ODE model is solved when constructing the characteristic curves. For each intra cellular ODE variable, one dimension in the PDE domain is needed. (In some cases the increase of variables can be simplified so that we do not need to increase the dimensionality of the PDE problem.)

Since these ODE's need to be calculated for all grid points, these operations represent the main workload of the algorithm. For this, the implementation of these operations are heavily tuned for performance. The equations are precompiled and optimised for faster evaluation and are executed in parallel by use of the CPU vectorisation units and distributed over multiple cores.

4 Numerical experiments

4.1 Comparison with analytical results

In [4] a solution for a Hepatitis C simulation of 14 days was derived. This solution is used as a reference to analyse the accuracy of the constructed numeric PSP solver.

4.2 Accuracy of the solution

4.2.1 Time step size

In table 1 we see how the solution becomes more accurate in function of the number of time steps. The stepsize = 14 days / #Time-Steps.

TABLE 1
CFL number, number of time-steps, spatial discretisation size, relative error, computation time and speed-up for decreasing number of time steps, with implicit Euler with f.p. iteration for $\Re$, implicit Euler with (Broyden's quasi-) Newton solver for the environment, JS-MMCC for the PDE and Implicit Euler for the characteristic curve.

CFL	Time-steps	dx	Rel. Error	Time (s)	speed-up
1	40000	0.01	0.01%	1949	x1
10	4000	0.01	0.03%	287	x7
100	400	0.01	0.2%	24	x81
1000	40	0.01	2%	3	x650
10000	4	0.01	40%	0.3	x6496

We see we get a pretty good approximation of the solution when we use 40000 time-steps. To get a CFL number smaller than 1, 40000 time-steps of equal size are needed. When we decrease the number of time steps, the error with the reference solution increases, but not dramatically. We can still get a suitable approximation of the solution ($\pm 2\%$) with only 40 time steps, resulting in a 650 times speed-up over traditional Eulerian PDE solvers. Or 1949 seconds versus 3 seconds. Simulation results are plotted in figure 6. As explained in section 3.1, only when the CFL is smaller than 1, Eulerian PDE solvers can be used. Comparing the error change for low with high CFL, we do see that, we need more and more time steps to get the same increase of accuracy. This is partially due to the fact we have a first order method. Another part is that other discretisations become more important: e.g. the spatial discretisation.

4.2.2 Spatial step size

When we decrease the spatial discretisation dx of the PDE, we record the simulation errors of table 2.

TABLE 2
CFL number, spatial discretisation size, number of time-steps, relative error, computation time and speed-up for decreasing spatial resolution, with implicit Euler with PEC solver

CFL	dx	Time-steps	Rel. Error	Time (s)	speed-up
1600	0.001	250	0.57%	82.693	x1
160	0.01	250	0.57%	7.9789	x10
16	0.1	250	0.60%	1.831	x46
8	0.2	250	0.62%	1.5424	x53
4	0.4	250	0.66%	1.4061	x59
1.6	1.0	250	0.79%	1.302	x63
5	1	80	1.2%	0.48272	x171
2.5	2	80	1.4%	0.4558	x182
1	5	80	1.6%	0.48089	x172

The decrease of spatial resolution results in an increasing error. The change is not in the same proportion as with the time step size. We also note that due to the use of vectorisation in the algorithm, the increase in calculation time for increasing the number of spatial grid-points is not drastic for grids with dx= 1 to 0.1, respectively 50 to 500 grid-points (in the domain 0-50), $1.3 => 1.8$ sec. When comparing dx= 0.01 and 0.001 (5000 and 50000 grid-points) we do see a proportional computation time increase $7.9 => 82$ sec. $\approx \times 10$. Since these operations can be done in parallel, more compute cores/nodes can be used to decrease the computation time again. We see there is a linear relationship between the number of grid-points and the CPU workload. But the CPU workload can be done in parallel.

4.3 Non-linear intra cellular models

For non-linear intra-cellular models, it is non-trivial sometimes impossible to derive a solution as was done in [4]. The big advantage of the numerical PSP solver constructed here, is that it is equally capable of approximating these less trivial to find solutions.

In fig.7.1c, the linear case, we clearly see that the intracellular model: i.e. the viral RNA growth rate, is linear in function of the intracellular state x: e.g. the number of virions in the cell. The different lines represent how this rate changes over time due to the modelled addition of a drug. (Each line is 1 day)

In 7.2c we see the relation between the number of virions and the production of new viral RNA strands modelled with a non-linear cellular model and see how this rate changes over time due to the addition of the curing drug. In 7.1b and 7.2b we see the distribution of the number of cells in each of the different characteristic states, plotted for each day. 7.1a and 7.2a show the simulation results of the environment variables: i.e. the total number of cells T and the total amount of virions V and I and R are the total number of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2810077, IEEE/ACM Transactions on Computational Biology and Bioinformatics

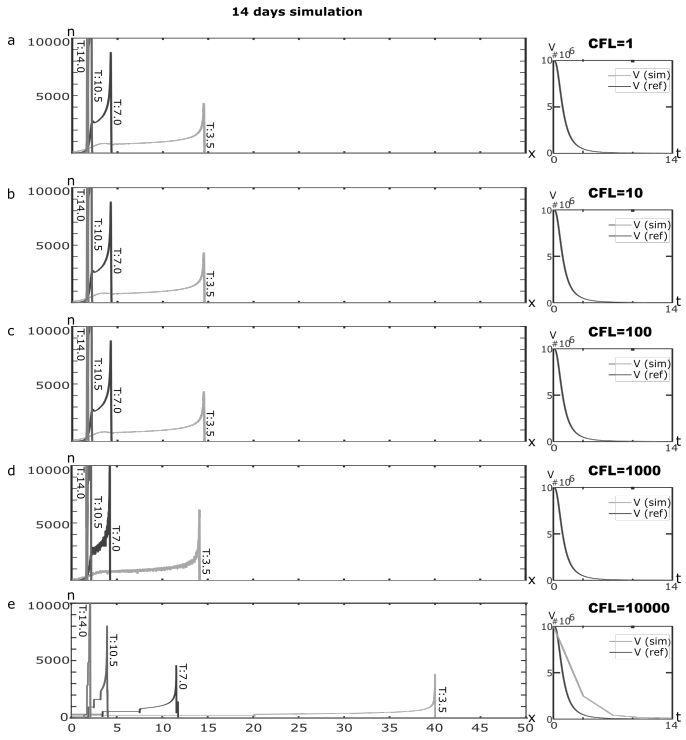

Fig. 6. Comparison of the solution of the PDE (left) and environment variable (Viral Load) (right) of the PSP model solved with a) CFL 1 b) CFL 10 c) CFL 100 d) CFL 1000 e) CFL 10000. The distribution of the P-state is shown for time-step 3.5, 7, 10.5 and 14.

infected cells and total number of viral RNA strands in the liver.

In fig. 7.3 we see a simulation where the drug is inserted after 4 days instead of day 0. The first 3 days we see in fig. 7.3b the progression towards the steady state without a drug (x=7). Then, due to simulation of the drug injection, we see the distribution change and converge towards another steady state point (x=2).

So, we see here that the PSP simulator is capable of simulating a wide variety of models and experiments: Linear, non-linear intra-cellular models, time varying parameters, non-steady state initial conditions. These examples illustrate that all these models can be simulated in a stable way.

4.4 Time steps vs implicit iterations

In table 3, we use an implicit method to solve the env. ODE i.e. Crank Nicholson, with 3 different number of time-steps. We see that the solver does not fail for a very wide range of the times-steps solver parameter. But we do see that the computation time does not decrease linearly with the number of time-steps and in this case even increases when lesser time-steps are used. This can be explained by looking at the number of iterations the fixed point solver needs to do to converge.

#JSCN is the number of times the intracellular model g is calculated. This number divided by the number of characteristic curves and the number of time-steps results in the average number of iterations per characteristic curve. We see that the fixed point solver needs, on average 5960 iterations for the 65 time-steps case vs only 77 iterations for the 265 time-steps case, to solve the implicit system.

TABLE 3
Error and detailed overview of the number of function evaluations in the PSP solver for 3 cases with different time-step-size.

#Time-steps:	65	265	2650
Rel. Error:	13.8654%	3.5199%	0.2977%
CPU time [s]:	1644.7	94.4	310.62
#PSPUpdates:	65	265	2650
#Eruns:	17898	1400	5949
#PSPSteps:	65	265	2650
#JSCN:	1.937E8	1.019E7	2.964E7
#JSCN/char:	5960	77	22.3

So the version with the least time-steps needs ≈ 20 times more rhs evaluations resulting in an $\approx$ 20 times longer simulation time. (*times are for non vectorised/optimized code)

For this model we see that it is sometimes better to use more/smaller time-steps than to try to reduce the number of time-steps to a bare minimum, to speed things up. When we increase the number of time-steps too far, the total computation time increases again. This is illustrated by the last example, where 2650 time-steps are used and the computation time increases again.

Another important conclusion from this table is that the implicit method allows to make rough but stable predictions when the system becomes more difficult. By reducing the number of time-steps, we converted the easy problem into a much harder/ more stiff system. We see that then, the stability of the implicit solvers is really necessary to keep the overall solver stable.

At last, it should be noted that in this example a fixed point iteration was used to solve the implicit equations. When using e.g. quasi-Newton solvers, this effect is less pronounced, since for these methods the number of iterations to solve the system is much smaller so quasi-Newton solvers allow even bigger step-sizes, with still a clear compute benefit. As an example, in table 2 we used the Broyden method for the same problem, and we see almost no effect of the increased compute-times for bigger time-steps and see an almost linear speed-up from 40000 up to 4 time-steps.

So this example also illustrates the importance of a good equation solver. When the equations can be solved quickly, the number of time-steps is proportional with the computation time.

4.5 The order of the method

For the different ODE's, i.e. $\Re$, f and g, multiple solvers can be used. We investigate here the influence of the order of the method for solving f and g. The ODE solver for $\Re$ is fixed to implicit Euler to maintain maximum stability. In the first three results of table 4, (exp. 1, 2, 3), we explore the order of the explicit ODE solvers (respectively order 1, 2 and 4). The experiments confirm that for the same spatial and time discretisation the error is significantly lower when using higher order methods.

When using implicit methods, experiment 4 and 5, i.e. Crank-Nicholson and implicit Euler, we see a reduction of the error compared to the explicit methods. This indicates that the step-size is still large and that the larger stability of these implicit methods here also translate to a more accurate simulation. The implicit Euler method reduces the error

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2810077, IEEE/ACM Transactions on Computational Biology and Bioinformatics

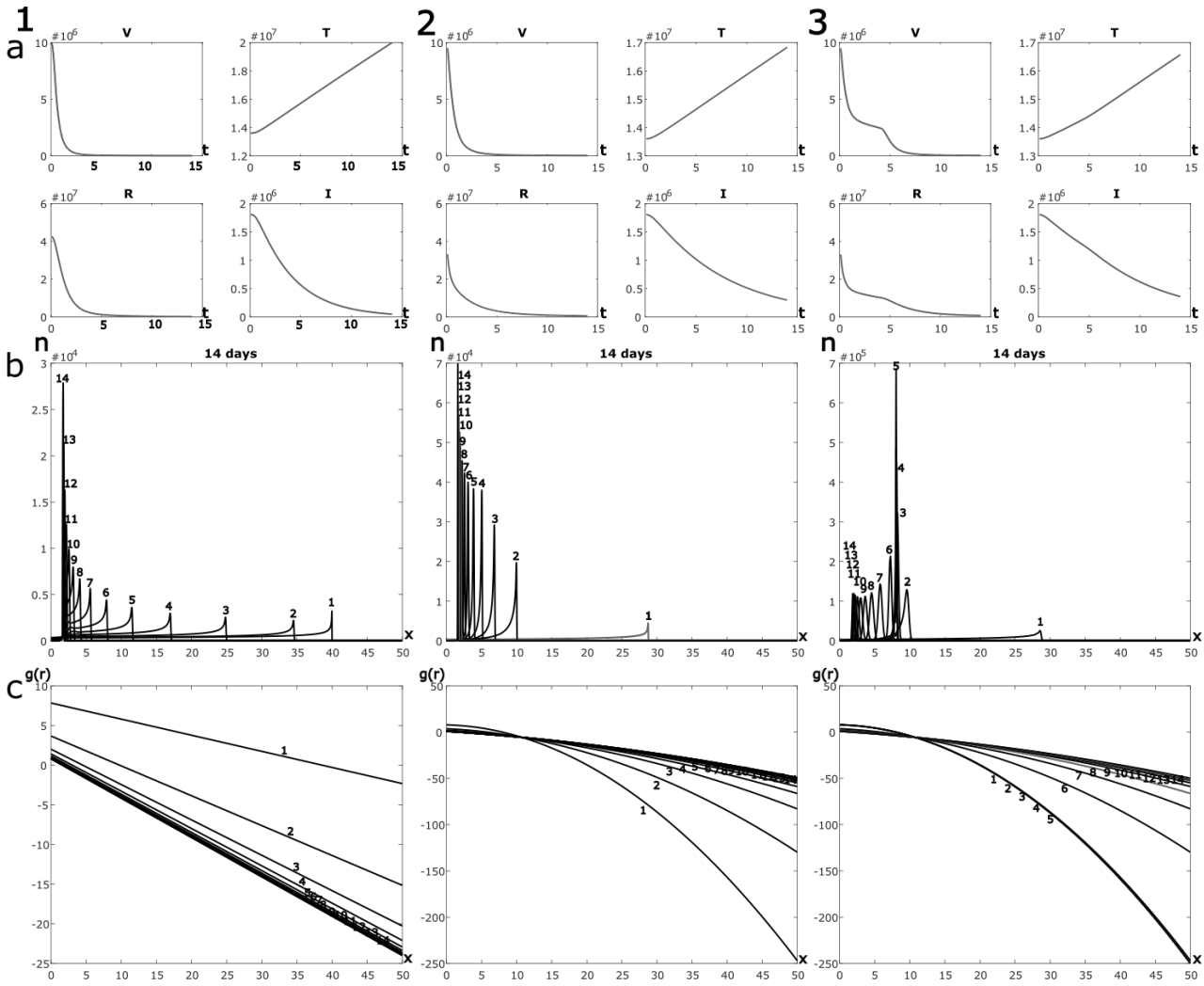

Fig. 7. Solution of a 14 day PSP simulation for the 1) linear and 2) non-linear intra-cellular model. 3) non-linear model with treatment started after 4 days instead of day 0 in the previous cases, and a non-steady-state initial value. a) Shows the environment variables: V-Total number of virions, T-Total number of cells, R-Total amount of viral RNA in the liver, I-Total amount of infected cells; b) the distribution of the cells for 14 days c) the production of viral RNA in a cell i.f.o. the amount of viral RNA in the cell. Plotted for each day 14 days of simulation.

even more than the second order implicit method: C.N.. The increase of stability of the implicit Euler method over the second order C.N. method translates itself again in a more accurate simulation result. In experiment 6, we use 4 times as much time steps to compare the explicit Euler method with RK4 again. This time, we take into account that RK4 does 4 rhs evaluations at 4 intermediate stage points. The error of the RK4 method is still clearly the smallest one, showing the benefit of the Chebyshev interpolation used in the RK4 method over the piecewise linear interpolation in the explicit Euler method. Also, the RK4 method is more stable than explicit Euler.

In experiment 7 we try to compare the impact of the ODE solver used for the characteristic curves versus the impact of the ODE solver used for the environment f.

We see that the error in 7 is $\approx$ equal to the Euler/Euler solver configuration in 1. So increasing the order of the characteristic curve in this PSP problem has little to no influence on the error. Experiment 8 and 9 are examples where the time-steps are 10 times increased in size. Using the Heun/Heun method, this results in a completely wrong solution, but due to the Implicit Euler outer loop, the method did not crash totally. Looking at the computation time: 70 seconds, we can conclude that a lot of implicit iterations had to be done to stabilise the Heun/Heun method. When using RK4/RK4 we get a very fast method with an error of $\approx 1.6\%$. In experiment 10 we show the influence of vectorisation and pre-compilation of the code. Comparing with the non-vectorised version from experiment 3 we see that this result has a clear 10 fold increase in speed.

In experiment 11, we confirm that for this problem, the error is mostly dependent on the ODE solver of the environment and much less on the ODE solver of the characteristic curve. By using the Heun method for the characteristic curve, we get almost the same accuracy as with CN/CN in exp. 4, but in a lot less time.

The last experiment was an attempt to get a very accurate solver with only 600 time steps. And so we used Implicit Euler for the environment and a second order implicit method for the characteristic curve. But this resulted

in slightly less accurate solution than IE/IE, in exp. 5. This also points towards the need for stability when building the characteristic curves. Leaving the implicit Euler/Implicit Euler as the most accurate solver when discretising this particular problem in 600 time-steps.

TABLE 4
Some experiments with different solver combinations. (Times are of the non-vectorised code, except when noted otherwise)

Exp.	method	nSteps	Rel. error	time(s)
1	IE/Euler/Euler	600	1.96%	119.9
2	IE/Heun/Heun	600	0.97%	101.8
3	IE/rk4/rk4	600	0.27%	56.3
4	IE/CN/CN	600	0.47%	216.8
5	IE/Impl.E./Impl.E	600	0.19%	297.2
6	IE/Euler/Euler	2400	0.45%	259.2
7	IE/Euler/Heun	600	1.94%	149.7
8	IE/Heun/Heun	60	249.21%	70.1
9	IE/rk4/rk4	60	1.62%	7.1
10	IE/rk4/rk4(vectorised)	600	0.27%	5.2
11	IE/CN/Heun	600	0.47%	37.9
12	IE/Impl.E./CN	600	0.21%	284.0

5 APPLICATIONS

5.1 Adaptive time stepping

5.1.1 Motivation

In the previous section the time step was held fixed. A big speed-up can be achieved by using variable time-stepping. For this we need an error estimating function to control the step-size. Different approaches exist, and a combination of approaches is used.

5.1.2 Controlling the PSP step-size

$$p_2(t_2) = p_1(t_1) + dt\Re(p) \tag{54}$$

This step-size dt can be adjusted automatically based on a number of criteria. Depending on the problem and the user requirements, other criteria are relevant. Currently we monitor 4 variables to define criteria.

- $maxEa$: The maximum absolute change of the environment variables per time-step
- $maxEr$: The maximum relative change of the environment variables per time-step
- t_{ss}: The time-step-size
- $nSolve$: The number of iterations to solve the implicit system

Based on these values, different constraints can be set, triggering different actions. Currently three values are checked, to adjust the time step-size (t_{ss}).

- $maxEa >$ PSP.ABS_ERROR $\Rightarrow t_{ss} = 0.9t_{ss}$
- $maxEa < 0.7$ PSP.ABS_ERROR $\Rightarrow t_{ss} = 1.13t_{ss}$
- $maxEr >$ PSP.REL_ERROR $\Rightarrow t_{ss} = 0.9t_{ss}$
- $maxEr < 0.7$ PSP.REL_ERROR $\Rightarrow t_{ss} = 1.13t_{ss}$
- $nSolve > 0.7$ PSP.MAXITER $\Rightarrow t_{ss} = 0.9t_{ss}$
- $nSolve < 0.5$ PSP.MAXITER $\Rightarrow t_{ss} = 1.13t_{ss}$

Two other criteria have to be met to confirm convergence of the solver: If one of the constraints is not met, the solver will discard the solution of this time-step and will do the computation again with a smaller time-step.

- $maxEa <$ PSP.MAX_ERROR and
- $nSolve <$ PSP.MAXITER .

When all time-step size modifications are done, a check is done if t_{ss} is inside the interval [PSP.dt_min, PSP.dt_max]

- $t_{ss} >$ PSP.dt_max $\Rightarrow t_{ss} =$ PSP.dt_max
- $t_{ss} <$ PSP.dt_min $\Rightarrow t_{ss} =$ PSP.dt_min

The way dt is adjusted, each time a constraint is not met, can be tuned for the problem at hand. We note that the time steps used in the sub-solvers for environment, tissue and cell can be different and more refined than the overall PSP time-step-size. The PSP solver step-size, defines the interaction between the different sub-solver results.

After simulation at variable time-steps, the results are interpolated to generate the numbers at the requested points in time. For this we need to buffer the intermediate results. Due to the adaptive time-steps, the size of the buffer is unknown at compile time and should be adjusted during execution. This is a time consuming operation. Doubling the reserved memory space, each time the buffer becomes too small is a heuristic which makes a good balance between size and speed.

5.1.3 Error estimation

By setting the PSP max error we also constrain the limit of the expected numerical error.

$$\hat{error} > nTimesteps \times \text{PSP.MAX.ERROR} \tag{55}$$

5.2 Simulating a clinical trial

Simulation of a clinical trial can be interesting for trial optimisation and/or evaluating the efficacy of a certain trial run for validating an hypothesis.

A trivial implementation would be to simulate from one event to the next, but setting the simulation step-size based on the mathematical complexity, followed by a linear interpolation, is a more computationally efficient approach. The next sequence can be followed to simulate a clinical trial of multiple persons: The computational load is distributed over the different cores and nodes of a super cluster as follows:

5.2.1 A clinical trial simulation

1. Separate the entire trial into trials, of just one individual and distribute the work over compute core/nodes. fig. 8 (Remark: When two or more trials are mathematically equivalent, this approach is not maximal efficient; since the same simulation is done twice.)
2. Separate data entries based on event ID: i.e. invasive dose events and non-invasive measurement events.
3. Start the simulation of the model up to the first invasive event. Store the intermediate results in a matrix T_{race}.
4. Simulate the invasive event. Append the simulation results to T_{race}.

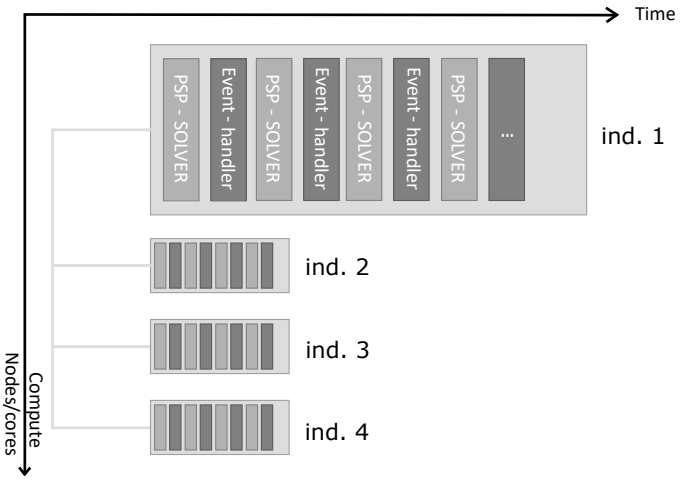

Fig. 8. Distribution of the workload over the different cores/nodes of the compute cluster

5. (Re)Set the PSP solver in high resolution mode, to cope efficiently with the discontinuous behaviour of the event.
6. Simulate from this invasive event to the next invasive event with the (variable time stepping) PSP solver.
7. Repeat step 4, 5, 6 for all invasive events.
8. Simulate from the last invasive event towards the last measurement event.
9. Interpolate the intermediate PSP results in the T_{race}, to get the values of the variables at the points in time recorded in the clinical trial.
10. .. (In case of parameter fitting: Calculate the difference between simulation and measurements, adjust the model-parameters and start again)

A simulation of a patient receiving multiple doses of treatment is shown in figure 9.

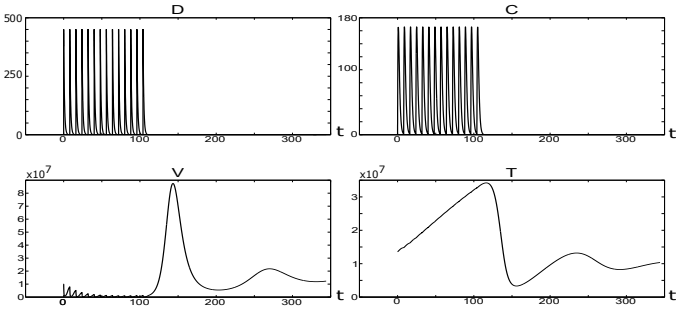

Fig. 9. Simulation of 1 patient in a clinical trial receiving multiple doses of medication over a certain period of time. D: Drug dose, C: Drug concentration, V: number of viral cells, T: number of Total Cells i.f.o. time

5.3 Fitting the model

To show another application of the PSP simulator, we show how it can be used in a parameter estimation set-up:

5.3.1 Fitting a model to an individual

In fig. 10 it is shown how the PSP solver was used in combination with the Levenberg Marquardt algorithm [11]

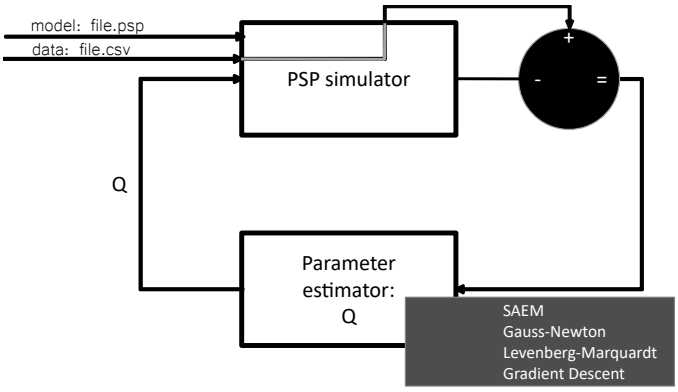

Fig. 10. Schematic overview of usage of the PSP simulator in a parameter estimation setting.

to estimate a model-parameter c (see eq. 1) based on artificial environment-data with noise. The original model, data and result are shown in fig. 11 and confirm a good parameter estimate.

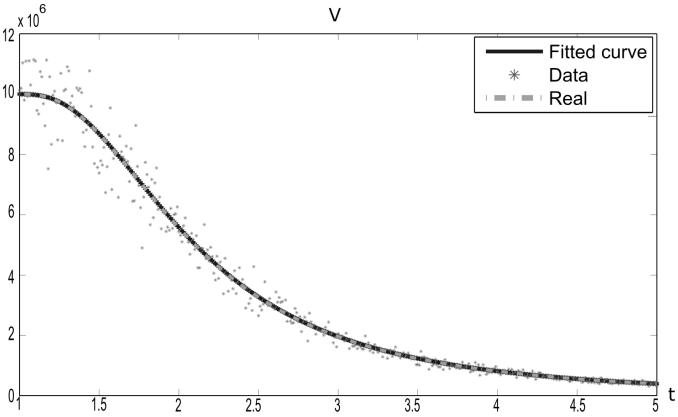

Fig. 11. Use case of the solver for estimating a parameter based on artificial data with normally distributed noise.

5.3.2 Fitting a non-linear mixed effects (NLME) model using stochastic approximation expectation maximisation (SAEM):

In a medical trial, multiple patients are observed. Sometimes, based on the distribution of the parameter over all patients, some assumptions can be made on the value of the parameter for a single person. If we incorporate this information in the model fit function, we get a (non-linear) mixed effect model.

$$y = f(x, X\beta + Zb) + \epsilon \tag{56}$$

Here y denotes the observation vector, β is the fixed effect vector with X the fixed effect design matrix and b the random effect vector with Z the random effect design matrix. ϵ is the observation error vector. b and ϵ are $\sim \mathcal{N}(0, \Psi)$ and $\sim \mathcal{N}(0, \sigma_\epsilon^2)$. with Ψ the covariance matrix and σ_ϵ^2 the variance of the error. f a function and x a data matrix of individual-specific predictor values.

If we want to fit parameters of this non-linear mixed effects model, a commonly used, robust, parameter fitting

TABLE 5
NLME estimates for different clinical trial set-ups on artificial data

#pers.	# meas. p. person	est. mean	est. var.	Init est.	CPU Time
4	5	5.1391	2e-16	8.63	20min
14	10	4.9706	0.1170	8.63	2h30
140	30	5.055	0.1242	8.63	19h30

algorithm is the SAEM algorithm. Based on artificially created data, polluted with normally distributed noise, we show in table 5, that we were able to fit the parameter c (see eq. 1) based on data from 4, 14 and 140 patients, each patient having between 5 and 30 measurements. The mean and standard deviation was set to be 5 and 0.3. We see in table 5 that the algorithm converged each time to a less or more accurate prediction depending on the amount of data given. Since SAEM requires a lot of iterations to converge, it takes a long time to compute. When a lot of data is available of different individuals, the user can get a clear benefit from using multiple cores on a multi-core machine or multiple nodes on a cluster to simulate the parameter values in parallel.

6 Discussion

Since a lot of mathematical approaches can be used to simulate an entire organ, or a mathematically equivalent multi-scale model, we want to discuss the performance improvement of the PSP framework as formulated here: i.e. the PSP framework with the PDE formulation and the PDE solved with the semi-Lagrangian method.

We want to point out that the best method to use, is based on the number of unique cells there are in the simulated organ. The semi-Langrangian method allows for adaptive approximation of these differences, at different levels of detail (see Figure 12).

For example, when we have 2 billion cells in the liver, we can calculate 2 billions cell ODE's. But when the state of the cell can be described by a linear combination of a few characteristic cells, we can simplify the simulation of 2 billion ODE's by solving one PDE.

When the different states can be described by one parameter, this becomes a 1 dimensional PDE. When more parameters are needed, this becomes a higher dimensional PDE to solve.

The downside is that solving higher dimensional PDE's becomes, very fast, a very difficult problem to solve when using the classical Eulerian methods. The number of PDE discretisation points blows up exponentially and also the time step should be kept very small satisfying the CFL constraint. As long as there are fewer cells than possible states, it remains more efficient to simulate each cell individually than to solve the PDE. Alternatively, we could use the Lagrangian method for solving the PDE. i.e. The Lagrangian method will solve the PDE by simulating each particle individually, which is computationally equivalent to solving the 2 billion ODE's.

The nice thing about the semi-Lagrangian method is that it combines the Eulerian and the Lagrangian approaches and allows to solve the PDE in the PSP model by solving it in a more Eulerian or more Lagrangian fashion by using more or fewer PDE grid-points. This includes the spatial grid-points but also the temporal grid-points, i.e. the time step size.

When using many grid-points and small time-steps, we approach the Eulerian PDE solvers. When using big time-steps and no (fixed) grid-points, we approach the Lagrangian PDE solvers.

Regarding the number of points which should be used in the semi-Lagrangian approach, we must understand that we approximate the behaviour of each cell by a linear combination of characteristic cells. (These characteristic cells are the ones for which we calculate the characteristic curves for: i.e. the discretisation points of the PDE.) The more 'diverse'/non-linear the possible states a cell can be in, the more characteristic cells that should be simulated. In turn, the more computationally intensive it becomes to solve the PDE for all these characteristic cells.

Conclusions regarding the optimal number of grid-points versus the number of different states are also applicable for the number of different inputs. When each cell has a unique input, it might not make sense to simulate characteristic cells, since each characteristic cell, should be simulated for all possible inputs. Again when the amount of different inputs is limited or can be derived from a linear combination of 'characteristic' inputs, the PSP approach makes sense again.

In the reference example, we assume that the input for each cell is the same, i.e. the environment the cells are in, is the same for all cells.

Due to the linear interpolation, in some cases, the number of grid-points can be reduced down to 1: i.e. As in [4].A. the mean field assumption is valid, so the average grow rate equals the grow rate of the average cell $avg(g(x)) = g(avg(x))$.

This also means that the total increase (of viral RNA strands in the liver) equals the increase in the average cell, times the number of cells. This in turn means we can model the organ in this case by simulating only 1 characteristic cell. In this case the PDE simplifies to an ODE and the computational benefit is maximal: i.e. calculating 1 easy PDE(=ODE) vs 2 billion (liver cell) ODE's. Here the PSP model equals a macroscopic model of the entire organ: i.e a set of macroscopic ODE equations. (This allowed [4] to simulate a PSP model with a standard ODE solver and we can use these results for validation)

The big advantage of the PSP model is that it allows the organ to be modelled by its macroscopic behaviour, or by its individual cell behaviour, or by a combination of both, in the same framework. The advantage of the current PSP solver is that it can turn from an Eulerian towards a Lagrangian PDE solver or combination.

So being able to gradually move from one extreme to the other, we can tune the method and get the best of both methods allowing optimal performance along a wide range of models.

7 Conclusion

The PSP modelling framework allows the model complexity to be increased, from the organ scale up to a detailed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2018.2810077, IEEE/ACM Transactions on Computational Biology and Bioinformatics

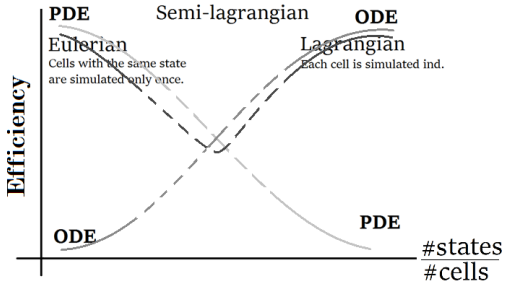

Fig. 12. Representation of the efficiency of different PDE solvers in function of the ratio number of states over number of cells.

simulation of each single cell. This way, gradually more information from the cellular level can be incorporated in the organ/environment model. The increase in detail does not necessary induce longer computation times since the numerical method derived in this paper uses a semi-Lagrangian PDE solver which allows to adjust the computation times and still maintain stability in almost all of its cases. The accuracy of the solution is of course depended on the computation time, but it is illustrated that good approximations can be found in exponentially less time. It is up to the user to set his preferred balance between computation time and accuracy. The coupling between the ODE solver for the environment model and the ODE solver for the individual model is maximally stabilised by the implicit Euler approach. This results in a stable PSP solver which can be used for exploration of a wide variety of models. Furthermore, the solver was proven to be capable of simulating all the models generated by the Levenberg-Marquardt algorithm for parameter fitting.

REFERENCES

[1] John Butcher. General linear methods for ordinary differential equations. *Mathematics and Computers in Simulation*, 79(6):1834–1845, 2009.
[2] John C Butcher, Zdzislaw Jackiewicz, and WM Wright. Error propagation of general linear methods for ordinary differential equations. *Journal of Complexity*, 23(4):560–580, 2007.
[3] C. K. Chu. Numerical methods in fluid mechanics. *Advances in Applied Mechanics 18*, 1978.
[4] X Woot de Trixhe, W Krzyzanski, F De Ridder, and A Vermeulen. vrna structured population model for hepatitis c virus dynamics. *Journal of theoretical biology*, 378:1–11, 2015.
[5] Markus Hegland and Paul E Saylor. Block jacobi preconditioning of the conjugate gradient method on a vector processor. *International journal of computer mathematics*, 44(1-4):71–89, 1992.
[6] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, Inc., Bristol, PA, USA, 1988.
[7] Alexander Kurganov and Eitan Tadmor. New high-resolution central schemes for nonlinear conservation laws and convection–diffusion equations. *Journal of Computational Physics*, 160(1):241–282, 2000.
[8] Johan A Metz and Odo Diekmann. *The dynamics of physiologically structured populations*, volume 68. Springer, 2014.
[9] David Potter. *Computational physics*. John Wiley and Sons Ltd, 1973.
[10] Yousef Saad and Maria Sosonkina. Distributed schur complement techniques for general sparse linear systems. *SIAM Journal of Scientific Computing*, 21(4):1337–1356, 12 1999.
[11] G. A. F. Seber and C. J. Wild. *Nonlinear Regression*. NJ: Wiley-Interscience, 2003.
[12] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.
[13] P. K. Sweby. High Resolution Schemes Using Flux Limiters for Hyperbolic Conservation Laws. *SIAM Journal on Numerical Analysis*, 21:995–1011, October 1984.
[14] Morris. Tenenbaum and Harry Pollard. *Ordinary differential equations: an elementary textbook for students of mathematics, engineering, and the sciences*. Dover Publications, 1963.
[15] Michiel Van Dyck and Herbert Peremans. Realtime 3d sensor based air flow reconstruction. In *Eurographics 2012-Posters*, pages 41–42. The Eurographics Association, 2012.

Michiel Van Dyck Dr. ir. ing. Michiel Van Dyck is currently researcher at the Department of Mathematics - Computer Sciences, investigating computer aided medicine construction and validation. Received his PhD in 2012 for his research on biomimetic air flow sensors. Received a master degree in engineering science: electronics, data processing and automation in 2006 and a masters degree in applied engineering: electronics hardware design in 2004.

Xavier Woot de Trixhe Ir. Xavier Woot de Trixhe is currently a senior scientist at Janssen Pharmaceutica and doctoral researcher at the KUL. Previously he worked as a scientist at Exprimo NV (2008 -2011) doing PK-PD Modelling. He has received his master degree in Engineering, Electronics: Medical Devices at the University of Leuven in 2006.

An Vermeulen Prof. dr. apr. An Vermeulen graduated as a pharmacist from the UGhent in 1989, and received her PhD from the same university in 1994, with a thesis entitled: The influence of ageing on the enantioselective pharmacokinetics of beta-blockers in the rat. In May 1992, she joined Janssen Pharmaceutica NV where she consecutively worked in the Preclinical & Clinical Pharmacokinetic departments, and the Advanced Modelling and Simulation group. Currently, she is a Clinical Pharmacology and Pharmacometrics consultant of the QS Consulting group within Quantitative Sciences (80% assignment). She also works as a visiting professor in Pharmacokinetics at the UGhent (20% assignment).

Wim Vanroose Prof. Wim Vanroose is - since 2006 - a Faculty member of the department of Mathematics and Computers Science at the University of Antwerp. After his PhD in 2001 in computational physics he has spent three years as a computational scientist at the Computing Sciences Division of the Lawrence Berkeley National Lab where he developed solvers for physics simulations that run on the NERSC supercomputers. Between 2004 and 2006 he worked on numerical multiscale methods for kinetic models at the K.U.Leuven. He currently leads research in PDE solvers for complex systems in various areas of science.