Assessment of Julia vs. Python for a Cell Mechanics Application

Hampus Fröjdholm & Carmen Lee

Project in Computational Science: Report

February 2020

Institutionen för informationsteknologi

Contents

Acronyms

CBMOS	Centre-based Model Overlapping Spheres. i, 1, 2, 3, 8, 13, 14, 15, 16
JIT	just-in-time. 8, 13, 15, 16, 17
ODE	ordinary differential equation. 2, 3, 7, 8, 9, 10, 13
REPL	Read/Evaluate/Print/Loop. 7, 8, 11
SIMD	single instruction multiple data. 6, 13

1 Introduction

Since its introduction in 1991, the programming language Python has gained considerable popularity. In recent Stackoverflow's yearly developer surveys [1, 2], it has frequently ranked amongst the top languages. The language Julia was created in 2009 and unveiled to the public in 2012. The relatively young niche language, designed for scientific programming, has also gained traction in recent years. Its co-creators won the James H. Wilkinson Prize for Numerical Software in 2019 [3] and the number of GitHub stars for Julia doubled since the stable release of Julia 1.0 in 2018 [4].

Though Python is easy to use as a high-level general-purpose language, it is not specifically built for achieving high performance. High performance computing using Python centres around the SciPy-ecosystem [5], with a number of libraries written in other compiled languages, providing efficient algorithms and data structures for the Python community. In comparison, Julia is created for scientific and high-performance computing while incorporating concepts from well-established programming languages, such as Matlab, Lisp and Python among others.

Our objective is to compare the two programming languages in terms of performance. Through the implementation of a centre-based model for tissue growth simulation, we are able to measure the performance of both in a setting of scientific computing.

The remainder of the section first introduces the centre-based model which is the conceptual model of how cell interactions are treated in our simulation. This is followed by a description on the Centre-based Model Overlapping Spheres (CBMOS) Python package, based on which we implemented our Julia version of the simulation. Section 2 compares Julia to Python with respect to syntax, language features, library ecosystem and workflow in a scientific computing context. Section 3 describes our Julia package and provides a brief user guide. Section 4 presents the experimental results and we conclude in the last section with some remarks on future extensions.

1.1 Centre-based model in tissue mechanics simulation

Centre-based models are widely used for illustrating the mechanics of cell populations [6, 7]. In a centre-based model, cells are regarded as two- or three-dimensional spheres of the same radius. The spheres are capable of interacting with each other when the cells overlap (the distance d between the cell centres is less than twice the cell radius). Fig. 1 illustrates the interaction mechanisms between the cells. When d is very small, the cells repel. When the cells move apart, there is a point where the cells are at rest, after which cells attract before moving too far away from each other.

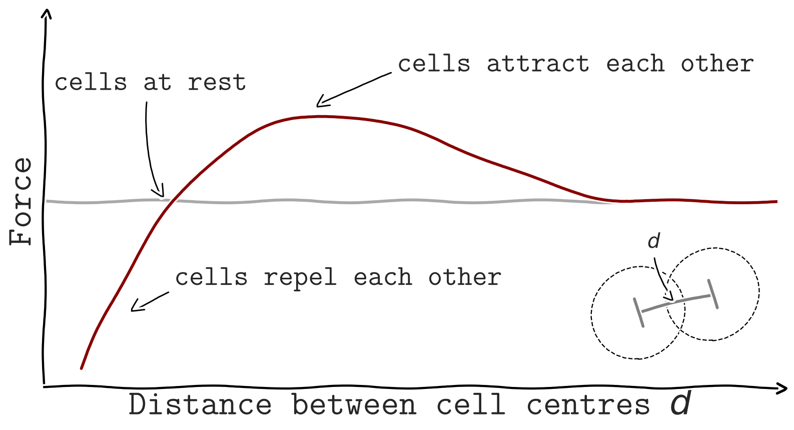

Figure 1: Repulsion/adhesion strength varies with the distance between cells.

It is assumed that cells divide given a predefined cycle-time, which is normally distributed. The newly added cells push the population into a new spatial configuration. The mechanical repulsion or adhesion, conditioned on the distances between the cells, can be modelled using different force models depending on the assumptions on the interaction. Fig. 2 illustrates a comparison between a few different force functions. As we can see, the models differ in magnitude and how rapidly the forces change in relation to the change in distance. In total, CBMOS implements 11 different force models.

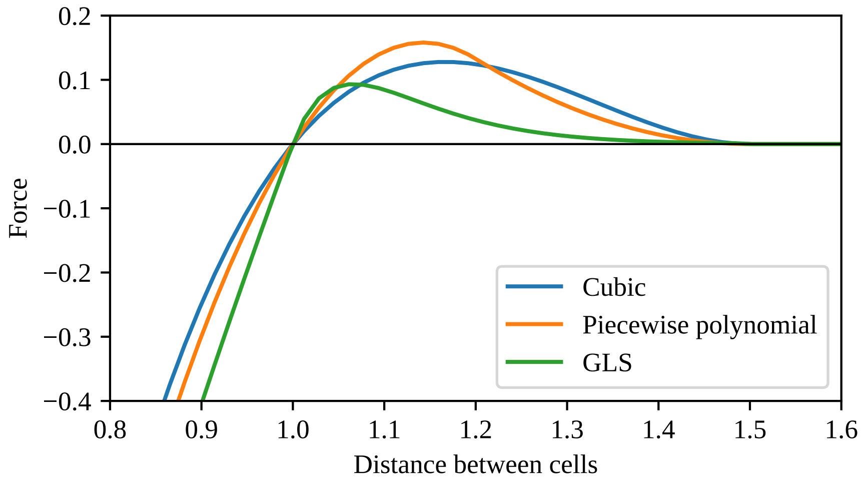

Figure 2: Examples of different force-function models. The radius of influence is set to 1.5 unit length

The forces are balanced on each cell centre. According to Newton's second law of motion, for a single cell[1]:

$$m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = \mathbf{F}_{visc} + \mathbf{F}_{cell}. \tag{1}$$

The left-hand side of Eq. 1 is the inertial force. $\mathbf{F}_{visc}$ is the viscous force or the drag force and $\mathbf{F}_{cell}$ the total repulsion or adhesion force exerted on the focal cell from its neighbours. The viscous force, defined by Stokes' law under an extremely small Reynolds number, is proportional to the velocity $\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t}$:

$$\mathbf{F}_{visc} = -\eta\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t}. \tag{2}$$

The Reynolds number is defined to be the ratio between the inertial force and the viscous force [9, p. 27]. The Reynolds number for living cells in the substrate environment is evaluated at the magnitude of 10^{-4} or 10^{-5} [10]. Under this condition, the inertial force on the left-hand side of Eq. 1 is ignored. With the left-hand side being zero, for a cell i, a first-order differential equation is obtained [6]:

$$\eta\frac{\mathrm{d}\mathbf{r}_i}{\mathrm{d}t} = \mathbf{F}_{cell,i}(t) = \sum_{j\in\mathcal{N}_i(t)} \mathbf{F}_{ij}(t), \tag{3}$$

where $\mathbf{r}_i$ is the position of the cell centre, $\mathcal{N}_i$ the set of cells within the radius of influence. $\mathbf{F}_i$ denotes the sum of the forces exerted on the focal cell by nearby cells and η the damping constant, which is also the viscosity of the fluid with the unit Pa· s. Once all the forces are obtained for a time step, the ordinary differential equations (ODEs) are solved numerically to obtain the new positions.

[1]The equation could also contain an additional term–the elastic restoring force if the deformation of the cell itself is considered [8].

1.2 Centre-based Model Overlapping Spheres (CBMOS)

CBMOS is a simulation package written in Python by researchers at the Division of Scientific Computing, Department of Information Technology, Uppsala University[2]. Currently, there exist several computational frameworks for cell mechanics simulation, e.g., CompuCell3D [11], TiSim [12], Chaste [13] and PhysiCell [14]. However, most of these frameworks focus on domain-specific tasks such as cancer cell growth, instead of the numerical aspect of the simulation. The CBMOS framework aims to provide a light-weight tool where different force models and solvers can be tested for performance. Thus, the framework enables the exact choice of the preferred force function, its parameters, the type of solver and the time step. The simulation creates a `CBMSolver` object that takes a list of `Cell` objects as input and returns all the lists of `Cells` at each time step (see Fig. 3). The `Cell` object contains the following information: ID, position, birth time, next division time and parent ID. The solver also includes methods handling the cell splitting events and integration of the ODE in Eq. 3.

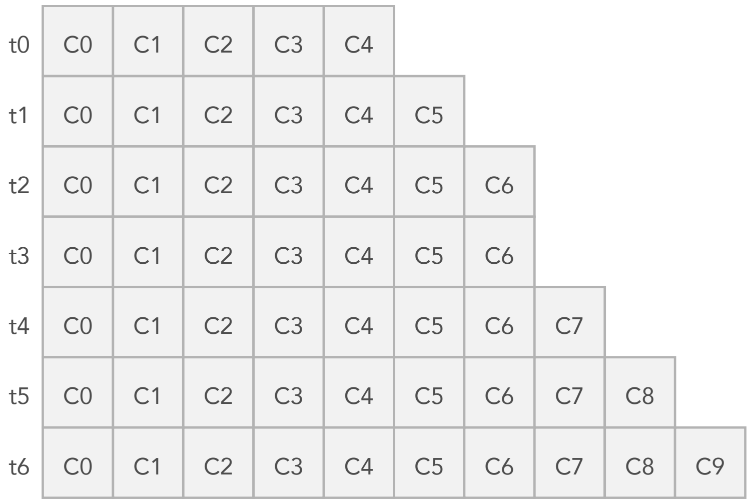
t0 C0 C1 C2 C3 C4
t1 C0 C1 C2 C3 C4 C5
t2 C0 C1 C2 C3 C4 C5 C6
t3 C0 C1 C2 C3 C4 C5 C6
t4 C0 C1 C2 C3 C4 C5 C6 C7
t5 C0 C1 C2 C3 C4 C5 C6 C7 C8
t6 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9

Figure 3: Schematic for the `CBMSolver` output. `CBMSolver` returns the cell history which is a list of lists containing the `Cell` objects at each time step. Each row is the list of `Cells` at the respective time step. Each `Cell` object contains the information of ID, position and parent ID among others.

A minimal working example for a 1D simulation is shown in Lst. 1. The choice of the force function is logarithmic, and the numerical integration method is Euler forward. Users can opt for more complex numerical method using the `scipy.integrate.solve_ivp` API. The last element of `history` is the end state of all the cells.

```python
import numpy as np
import force_functions as ff
import euler_forward as ef
import cell as cl

dim = 1
cbm_solver = CBMSolver(ff.logarithmic, ef.solve_ivp, dim)
cells = [cl.Cell(0, [0], proliferating=True),
         cl.Cell(1, [0.3], proliferating=True)]
t_data = np.linspace(0, 1, 101)
history = cbm_solver.simulate(cells, t_data, {}, {})
```

Listing 1: CBMOS simple usage example.

[2]Work-in-progress by Sonja Mathias and Adrien Coulier.

2 Comparing Julia to Python in a scientific computing context

This section compares the Julia syntax to Python and lists the major language features of Julia that we used when implementing CbmosJulia. A short discussion on the numerical computing ecosystem and the workflow follows at the end.

2.1 Language features

Syntax. There are a few notable syntax differences Python users should look out for coming from Python to Julia. Superficially the syntax of Julia is much more similar to that of Matlab. For instance, Julia is indexed from 1 instead of the familiar 0-index for Python, the last index is `end` instead of -1, and the last end index in slicing is included (see the examples in Tab. 1).

Math-like notations and Unicode characters are supported in Julia. For example, users can redefine the XOR function using the Unicode character ⊕:

```
julia> ⊕(a::Bool, b::Bool) = xor(a, b)
⊕ (generic function with 1 method)

julia> ⊕(true, true)
false
```

This can then be used as an infix operator [15]. In fact, most operators in Julia are just functions with mathematical symbols as their names. The familiar `lambda` functions in Python are anonymous functions in Julia defined as `x->f(x)`. Ordinary functions are enclosed by the function declaration statement `function ... end`. The keyword `end` is also used to indicate the end of all other block structures in Julia. The return statement of a function is optional as the function will return the value of the last statement. A useful construct in Python is the `with`-statement using context managers. A similar effect of automatic resource management can be achieved in Julia using the `do`-block. The `do`-block is a more convenient syntax for using anonymous functions as arguments. The Julia equivalent to opening a file with the `with`-statement can be seen below:

```
open("filename.txt") do file_handle
    # Do something
end
```

Table 1: Examples of syntax differences between Python and Julia

Use	Python	Julia
Slicing arr=[1, 2, 3]	`> arr[0:2]` `[1, 2]`	`> arr[1:2]` `[1, 2]`
varargs	`def f(*argv):`	`function f(x...)`
varkwargs	`def f(**kwargs):`	`function f(; kwargs...)`
Typing	`def f(x:int) -> int:`	`function f(x::Int8)::Int8`

Tab. 1 lists some examples of the syntax differences between Python and Julia. An exhaustive list comparing Julia to Python and a few other languages can be found at the official Julia website [16].

Multiple dispatch. The core programming paradigm of Julia is multiple dispatch, which refers to performing function dispatch dependent on the run-time type of all parameters. This can be seen as a generalisation of single dispatch, common in object-oriented languages such as Python, where the function being executed depends on the type of the object. In Julia, when a function is defined for the first time, a generic function is defined with a single method. Subsequent definitions of the same function with different parameter types add methods to the generic function. For instance, we can define `inner_prod(u::Vector, v::Vector)` and `inner_prod(f::Function, g::Function)` as two methods of the same function. When the inputs are `Vector`s, the function can compute the dot-product of the two vectors; when the types are `Function`s, it can perform an integration of the two functions within some defined bounds. The core of Julia uses multiple dispatch heavily, for example the addition operator, `+()`, has 166 methods for adding anything from integers to ranges and matrices. To check the methods of a function `func` you can use `methods(func)`. Implementing a new operation of addition for custom types is just a matter of defining more methods to `Base.+`.

Types. Julia has a powerful type system, featuring parametric types and both abstract and a number of concrete types. In this report we focus on abstract types and one of the concrete types, namely composite types. One thing to remember about types in Julia is that types belong to values, not variables. A value is bound to a variable name, that can be optionally typed, but the type is part of the value. Additionally, at run-time there are no abstract types, only concrete types. As such, abstract types are useful only for the construction of type hierarchies. The `Number` abstract type is the super-type of all number types in Julia. A function taking a parameter of type `Number` accepts any number type, such as integers, floating point numbers or complex numbers. At the top of the type hierarchy there is the abstract type `Any`. All types are implicitly and explicitly subtypes of the `Any` type. Thus, a variable with the `Any` type declaration can hold any value. Abstract types can be defined using the `abstract type` keyword.

Composite types are also known as records or structs. They store other values in named fields. In many other languages the only type that can be defined by the user are composite types. They are defined using the `struct` keyword. Along with the type Julia also allows the definition of a constructor for the type. If none is specified there will be a default constructor generated. Constructors can be used to allow for easier construction of the type.

Singletons. Julia has a special kind of composite type named singletons. These are empty composite types and there can only be a single instance of such types. The only property of singletons is the type. If a singleton is passed as parameter to a function, it can dictate the function's dispatch. This property is very useful when we want to take advantage of the multiple dispatch in Julia. For instance, if we have multiple definitions of the function but the numerical inputs are the same, we can use a type parameter to indicate the difference. This is particularly relevant for the force function in the CbmosJulia package, where the force function performs different computations dependent on the choice of the force model. The code in Lst. 2 shows the declaration of two of the force functions.

```julia
abstract type ForceFunction end
struct Linear <: ForceFunction end
struct Cubic <: ForceFunction end

function force(::Linear, r; μ=1., s=1., rA=1.5)
    # Do something
end

function force(::Cubic, r; m=1., a=5., s=1., rA=1.5)
    # Do something
end

function force(::Morse, r; m=1., a=5., s=1., rA=1.5)
    # Do something
end
```

Listing 2: Example declaration of two of the force models.

The `force()` function declared in Lst. 2 takes a type parameter besides the mandatory parameter r and other optional parameters given after the semicolon. If we want to call the `force()` function for the cubic force-model, we would write the function as `force(Cubic(), r; kwargs...)`. Similarly, a linear force-model will use the `Linear` singleton to call the correct implementation (`force(Linear(), r; kwargs...)`). According to the Julia performance tips [17], breaking a "compound function" into multiple definitions rather than relying on `if`-statements helps the compiler call the most applicable code directly. Therefore, it is suitable to use the design choice above of having separately defined force functions with similar function signatures.

In-place operations. An in-place function refers to a function where one or more of the parameters passed by reference are manipulated by the function. This can be more efficient since there is no extra space needed for copying values. Common in-place operators that can be defined for Python classes are `iadd()`, `isub()` and `iconcat()`. Many Python packages such as NumPy and pandas also offer in-place options. Although present in Python, much more emphasis is placed on in-place operations in Julia.

In Julia, in-place implementations of functions are indicated by an exclamation mark at the end of the function name. The vectorised "dot" operator in front of an operator broadcasts the operator to all elements and returns an in-place assignment. This is the same as in Matlab, but works for all functions since Julia operators are normal functions.

Macros. Macros in Julia are a handy feature which allows for a simple interface and code generation. They allow for meta-programming, which means using Julia code to process and modify Julia code, and to even control if and when the modified code runs [18]. It is also possible to change the code to make it run faster for example when running code in parallel. A very common macro in Julia is `@time`. When added in front of a Julia command, it inserts commands before and after the code prior to it being evaluated, which will record the elapsed time and memory usage after the code is being run. Another useful example is the `@gif` macro. A simple usage is shown below:

```
@gif for i=1:100
    plot(...)
end every 10
```

The above usage of the `@gif` macro saves every 10th frame and outputs an animation gif at the end. Writing custom macros is similar to writing ordinary functions in Julia with the **`macro`** keyword:

```
julia> macro square(a)
           return a * a
       end
@square (macro with 1 method)
julia> @square 5
25
```

Explicit vectorisation. Another interesting feature of Julia is the preference for for-loops combined with the macro `@simd` (single instruction multiple data) over broadcasting using the dot notation. So in terms of syntax, the code looks de-vectorised. The premise is that the loops being annotated are perfectly parallel. The macro `@simd` allows the compiler to take extra liberties to allow loop re-ordering and automatic vectorisation when it sees fit, which can potentially improve the execution time [19]. The performance improvement is discussed in Rackauckas's DifferentialEquations Tutorials [20].

For a more thorough walk-through of Julia and all its features the official language documentation is a great resource [21].

2.2 Standard library and numerical library ecosystem

The most common packages for scientific computing in Python include NumPy containing the base implementation of multidimensional arrays and vectorised operators, SciPy for optimization, linear algebra and other technical computing and Matplotlib for result visualisation.

In Julia a lot of the functionality provided by NumPy and SciPy is present in the standard library, most notably the LinearAlgebra and SparseArrays modules which define functions for working with matrices and vectors, including the sparse variants. LinearAlgebra also includes wrappers for the underlying LAPACK and BLAS libraries.

For solving the system of ODEs, we use the Julia package DifferentialEquations [20]. It provides Julia implementations of solvers for differential equations which we will discuss in detail in Section 3.1. Other useful libraries such as Distributions, LinearAlgebra and Random offer self-explanatory functionalities. Plots is the current defacto standard plotting library for Julia. It supports multiple backends for plotting and can even use Matplotlib. It is enhanced by RecipesBase for plots of custom types of other libraries. For performance measuring, BenchmarkTools supplies the essential tools for tracking and comparing benchmark results.

2.3 Workflow for scientific computing research

Because Julia is relatively new, it is difficult to outline a workflow that is complete or future proof. We strongly encourage readers to consult a few additional guides online and test out different ways before settling with one. Below is a short summary of our approach.

Julia scripts can easily be run in the Julia Read/Evaluate/Print/Loop (REPL) console simply by using the command `include("<script name>")`. This effectively runs the script line by line and will also make all modules and functions defined in the script available in the global namespace.

To make a Julia package/module available in the current namespace, so-called importing, there are two ways: 1) `using <modulename>`, 2) `import <modulename>`. Both methods import the names exported in the module to the namespace. Exporting means, for example, that a function or type is explicitly declared with `export <name>` in the module definition. Without doing so, the names are not directly available but can still be reached using the dot notation similar to how Python works. The command `using <modulename>` imports all the exported names of the module into the global namespace. While `import <modulename>` only makes the module itself directly available. To extend a function defined in another module, the function needs to be imported using the `import` keyword or the module imported with `using <modulename>`. The subtle differences between the two approaches are better illustrated with examples as shown in Tab. 2.

Table 2: Comparison of available import statements with an example module `M`, which contains two exported functions `f1` and `f2` and one additional function `f3`, which is not exported (adapted from the official Julia documentation [22]).

Command	Available in the namespace	Functions extendable
`using M`	`f1, f2, M.f1, M.f2, M.f3`	`M.f1, M.f2, M.f3`
`using M: f1, f3`	`f1, f3`	
`import M`	`M.f1, M.f2, M.f3`	`M.f1, M.f2, M.f3`
`import M.f1, M.f3`	`f1, f3`	`f1, f3`
`import M: f1, f2`	`f1, f2`	`f1, f2`

To be able to import a local module, its project environment must first be activated. The code snippet below shows how to activate the project environment in the folder where the project is located. In the julia REPL, press `]` to switch to `pkg>` (the native package manager mode). Press `enter` to return to the REPL console.

```
shell> cd MyProject.jl
shell> julia
...
(v1.3) pkg> activate .
Activating environment at `<MyProject Path>/Project.toml`
(MyProject.jl) pkg>
julia> using MyProject
```

Local packages can also be installed into the global environment in development mode using `dev` instead of `add` at the `pkg>` REPL. This way the code will be linked instead of copied and changes will be visible in the global environment immediately. See the example below.

```
shell> cd MyProject.jl
shell> julia
...
(v1.3) pkg> dev .
```

For prototyping in Julia, it is convenient to use IJulia, a Julia-language backend similar to IPython in Python. It can be launched in the same way as launching a Juypter notebook. More information regarding installation and usage can be found on JuliaLang (https://pkg.julialang.org/docs/IJulia).

Due to the time it takes for Julia's just-in-time (JIT) compiler to warm up and precompile code, it is important to keep the Julia session alive for as long as possible. Using the library Revise, code can be recompiled when it is changed. Our workflow is to keep most of the simulation code and functions in an external file and use the REPL or an IJulia notebook with Revise to run the code. This way the Julia session is kept alive while allowing the simulation code in the external file to be changed. The biggest drawback using this method is that some structures are not allowed to be redefined, most notably composite types, requiring the Julia session to be restarted when such changes are made. Another problem is that Revise might sometimes not update the code, this is most often due to a syntax error in the external file. However, debugging can be a challenge as the error messages are not always clear.

Having described Julia's language features compared to Python, we will delve into our implementation of the CBMOS package in Julia in the next section.

3 Porting Python CBMOS to Julia

In this section we describe the Julia version of CBMOS, CbmosJulia. The main dependencies, the structure of the package and some examples are described. At the end we list the measures we took to improve the performance.

3.1 Package dependencies

The main dependencies are DifferentialEquations and RecipeBase. The former is essential for solving the systems of ODE and the latter for customised plots.

3.1.1 DifferentialEquations

The most important dependency of CbmosJulia is DifferentialEquations, which contains extensive functionalities for solving differential equations. For a continuous ODE, the general equation is:

$$u' = f(u, p, t) \tag{4}$$

where the p are the parameters of the model and t the time variable. f defines the rate of change of u. To solve the ODE, we also require the initial value u_0. The constructor `ODEProblem()` generates an `ODEProblem` struct, which is the input parameter for the generic `solve` function. A simple working example of exponential growth is given below [23]:

```julia
using DifferentialEquations
f(u,p,t) = 1.01*u
u0=1/2
tspan = (0.0,1.0)
prob = ODEProblem(f,u0,tspan)
sol = solve(prob)
```

DifferentialEquations also allows injection of user code into the solver using callback functions. A `Callback` function runs the user code when a given condition is fulfilled. There are several types of `Callback` functions at the users' disposal. They are capable of handling continuous and discrete time events. The callbacks can also be organised into a set to be passed to the solver. In our case, we opted for the `IterativeCallback`, which calls the callback at a specified integration time and then prompts for the next callback time. This allows us to iteratively apply the effect of moving the cells and finding the time for the next cell division event. An `IterativeCallback` is constructed as follows [24]:

```julia
function IterativeCallback(time_choice, user_affect!,
                           tType = Float64; initial_affect = false, kwargs...)
```

The function `time_choice(integrator)` returns the time of the next callback, and `integrator` is the previously defined `ODEProblem` struct. The function `user_affect!(integrator)` takes the same input and performs an in-place modification of the `integrator` thus it returns nothing. In our case, we used the callback to extend the ODE with the addition of the new cell and its necessary data.

3.1.2 RecipeBase

Another package we find very useful is RecipeBase, which is part of the JuliaPlots ecosystem [25]. It allows users to define transformations of custom types and attach attributes to allow plotting with the Plots package. With minimal code, users can implement complex visualisation routines. We use this package to define how the simulation results should be visualised.

A minimal working example plotting the growth of the cells given the `Growth` type and the cell population `cellPop` from the solver result is shown below:

```julia
struct Growth end
@recipe function f(::Growth, cp::CellPopulation)
    sol = cp.sol; t = sol.t; dim = cp.dim;
    y = [div(length(u), dim) for u in sol[:]]
end
plot(Growth(), cellPop, xlabel="time step", ylabel="number of cells")
```

Fig. 4 shows the result produced by the above code.

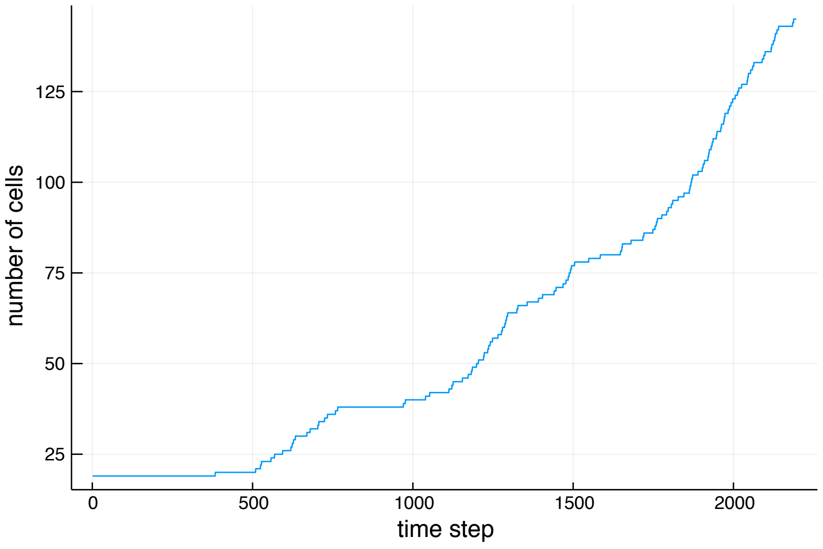

Figure 4: Example of using recipes with Plots and the custom types defined in CbmosJulia.

3.2 Structure of the package

The CbmosJulia package contains three modules (Tab. 3).

- The `cells` module contains useful tools for working with the simulation result. This includes the types `CellPopulation` and `CellHistory` as well as functions for extracting information from them. The `CellPopulation` type is essentially a wrapper around the `ODESolution` generated by the solver and other useful information we want to track for the cell population. `CellHistory` stores the ID, time and position information of a particular cell. The module also defines the plot recipes [26] which visualise different results based on the input type.
- The `forcefunctions` module defines the mathematical expressions of the supported force functions.
- `CbmosJulia` is the main module containing the definition of the ODEs, callbacks, event handlers and the simulation main method. The main function `simulate(args...)` returns a `CellPopulation` struct.

Table 3: Sub-modules in the CbmosJulia package

Module	Functionality
cells	Structs and functions for cell related operations
forcefunctions	Multiple dispatch of the force functions
CbmosJulia	Functions for the simulation

The simulation involves two iterative steps: cell division and cell relaxation. Cell divisions lead to tension between the cells that are subsequently relaxed as the population is pushed into new spatial configurations. A cell division happens when a cell splits into two: the child and parent separates symmetrically from the original position of the parent in a randomly chosen direction determined by `div_dir()`. The separation distance of the cells can be set but has a default of 0.3. The function `mod_divtime!()` performs an in-place modification of the division time for the parent cell and appends one for the child cell. The division event is, as previously mentioned, handled by the `IterativeCallback()` function in the DifferentialEquations package, which triggers the event when the next division time is reached. The integrator `solve()` solves the ODE system and updates the positions of the cell centres. This iterative process goes on until the specified final time is reached.

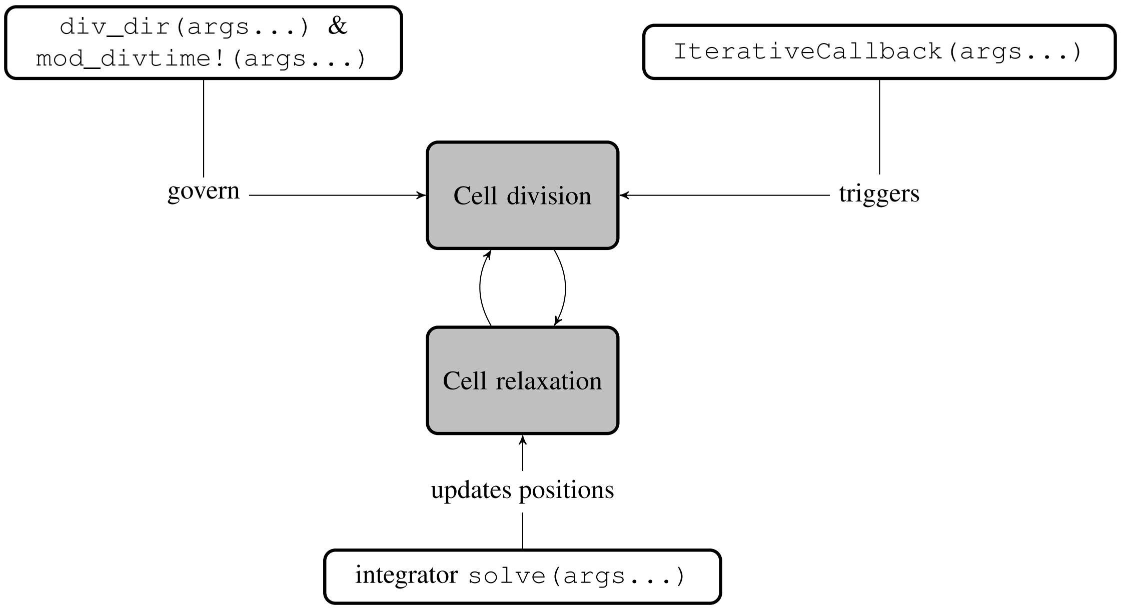

Figure 5: Events in the simulation

3.3 User guide

To import the package, in the Julia REPL console [27], run

```
pkg> activate <path-to-CbmosJulia>
julia> using CbmosJulia
```

This activates the CbmosJulia project and makes the package available. Lst. 3 shows the function signatures for the main simulation function. The semicolon `;` separates the positional and keyword arguments in Julia. Cell information, force function type, and time span are mandatory; the default distribution of the division time is a normal distribution centred at 24 hours. Lst. 3 is also a good example of the multiple dispatch, where the same function is extended given different argument types. The simulation supports supplying the cell positions in a 2D array or as the 1D flattened version. It also supports passing in the `CellPopulation` struct which could be the output of a previous simulation.

```
simulate(
    cells::Matrix{Float64},  # cell positions. In 2D: [x1, x2...; y1, y2,...]
    ftype::ForceFunction,    # type of the force function
    tspan,                   # tuple of time span
    dist=Normal(24);         # distribution of the division time
    forceargs=(),            # arguments for the force function
    solverargs=(),           # additional arguments for the solver
    seed=nothing             # seed for reproducibility
)

simulate(
    cells::Vector{Float64},  # position of cells. In 2D: [x1, y1, x2, y2 ...]
    dim::Integer, ftype::ForceFunction, tspan, dist=Normal(24);
    forceargs=(), solverargs=(), seed=nothing
)

simulate(
    cellpop::CellPopulation, ftype::ForceFunction, tspan, dist=Normal(24);
    forceargs=(), solverargs=(), seed=nothing
)
```

Listing 3: CbmosJulia simulation signatures

A minimal working example in 2D is given in Lst. 4. The example simulates the cell population for 30 h. The division time is set to be a standard normal distribution centered around 6 h. The numerical solver uses Euler forward algorithm with the time-step set to 0.01 h.

```
using CbmosJulia, DifferentialEquations, Distributions
# Initial cell configuration
n_x = 5
n_y = 5
xcrds = [(2 * i + (j % 2)) * 0.5 for i = 1:n_x, j = 1:n_y]
ycrds = [sqrt(3) * j * 0.5 for i = 1:n_y, j = 1:n_x]
cells = hcat([[x, y] for (i, (x, y)) in enumerate(zip(xcrds, ycrds))
                     if !(i in [1, 5, 6, 16, 21, 25])]...)

tf = 30.0   # final time
dt = 0.01  # simulation time step
μ = 1.0    # force function coefficent
# solverargs and forceargs can be either NamedTuple or Dict{Symbol, Any}
forceargs = (μ = μ,)
solverargs = Dict(:alg => Euler(), :dt => dt)

result = simulate(cells, Cubic(), (0., tf), Normal(6),
                  solverargs=solverargs, forceargs=forceargs)
```

Listing 4: CbmosJulia usage example. Some of the key features, such as choosing the force function and division time parameters are shown.

3.4 Performance optimisation

The Julia documentation contains a page with guidelines on writing fast Julia code [17]. When profiling CBMOS the construction of the ODE seemed to be a critical component. Focus was therefore spent on improving the performance of that part of the code. Additionally, since we are using the solvers from DifferentialEquations there is not much optimisation to be done in other parts.

ODE construction. In CBMOS the ODE is constructed using NumPy arrays. To avoid Python for-loops it uses exclusively NumPy functions. This leads to all ordered pairs of the cells being constructed and a lot of duplication. The first step in increasing the performance of the code is to de-vectorise. De-vectorising is when the explicit for-loops are written instead of using vectorised operations. In Python, we are forced to always use the vectorised operations of NumPy for performance. In Julia, using for-loops is often a lot faster.

De-vectorising allows us to improve the algorithm for calculating the forces between cells. Due to Newton's third law the force exerted by cell i on cell j is the same but opposite of the force exerted by cell j on cell i. In Julia we can take advantage of this and only calculate the force for every pair of cells once, essentially halving the number of loop iterations. Additionally, by de-vectorising the vector additions and norm calculations inside the inner loop we can avoid unnecessary heap allocations. Arrays in Julia are heap-allocated. Since the spatial dimension is low and the number of cells is large, the overhead of allocating numerous small arrays is very large. The result of de-vectorising in the inner loop is going from several memory allocations per cell combination to a total of 6 allocations for the entire function.

As a final boost to performance array bound checking was disabled, using the `@inbound` macro, and SIMD was enabled for the vector addition and norm calculations. To make disabling the bound checks and using SIMD safely a few considerations has to be taken into account, as detailed in the performance tips [17]. The type of the arrays are set to **`Array`** since indexing using the `1:n` style might not be safe with bounds checking disabled for all **`AbstractArray`**s. Also, when using SIMD the loop iterations have to be independent, which is the case for our vector addition and norm calculations. However, assigning into an **`AbstractArray`** might not be parallelisable, and that is another reason the type has to be constrained to **`Array`**. As a final check, the size of the array is checked to be the correct size before calculation is started.

Optional typing and type stability. The next part in writing fast Julia code is to make sure the types are correct. One way of doing this is to include type declarations for function parameters, return types and variables. The most important of these is the function parameters. By having the correct types, the JIT-compiler can produce efficient machine code and allows for what is called specialisation. Specialisation is when the compiler produces multiple versions of a generic function and selects the best one depending on the run-time type of the arguments.

Having the correct types is especially important for container types and structs. `Vector{Number}` is a one-dimensional array containing elements of the abstract type **`Number`**. The array of this type needs to be an array of pointers to numbers since a **`Number`** can be both floating-point or integer numbers (as well as rational or complex numbers). The type specification `Vector{<:Number}` allows for proper specialisation since in this case it is an array of elements of a subtype to **`Number`**. The distinction is that as long as the array at run-time only contains either floating-point numbers or only integers the array will be stored as a contiguous array in memory and the compiler can produce efficient code for these cases.

Another important part about types in efficient Julia code is type-stability. Type-stable code is code in which all types are known at compile time and the variables never change their type during run-time. This can be tricky when coming from Python since the type of a variable is not often taken into consideration. For example, the dimension of an array is part of the type, thus reshaping an array is not a type-stable operation. Assigning the result of a reshape to the reshaped variable is therefore not type-stable. If a performance critical function needs to perform a type unstable operation, it is recommended to split the function into a kernel function that performs the calculation and a function that does the conversion and then calls the kernel [17]. This way the kernel function, which does most of the work, can be efficiently compiled.

4 Results

In this section we provide details of our experiment and present the performance comparison results.

4.1 Benchmark setup

For benchmarking a Lenovo Y50-70 laptop with an Intel core i7 4720HQ running at 2.6 GHz and 8 GB was used. To avoid slow-downs due to swapping and since the simulation is quite memory intensive the swap partition was disabled.

At the time of writing, the latest versions of Julia and Python were used, namely, Julia 1.3.1 and Python 3.8.1. Julia was downloaded from the Julia homepage (julialang.org) and Python was installed through the Arch Linux repositories. Source code for the benchmarks is available upon request.

4.2 Initial configuration and parameters

A hexagon of 19 cells was used as the initial condition (Fig. 6). In the simulation, they divide and grow into a mono-layer covering a larger area (see Fig. 6). The cubic force function was used with parameters $\mu = 50$, $s = 1$ and $r_A = 1.5$. The division time distribution was a normal distribution with mean 6 and standard deviation of 1. Euler forward was used as the solver algorithm with the time-step 0.003 h. During the initial testing it was determined that with the given time-step, the longest simulation ran for 46 h before the laptop ran out of memory. At the end of the simulation, there were approximately 3000 cells. Also, a larger slow-down in CbmosJulia compared to CBMOS was observed for longer simulation times during initial testing. Thus, more simulation times were tested closer to 46 h. In total, 8 simulation times were tested starting from 20 h to 40 h with 5 h steps and then from 40 h to 46 h with 2 h steps.

Due to the stochastic nature of the simulation each configuration was run 10 times and the results were averaged. Since the simulations are long running, we do not anticipate significant random variations due to other running processes.

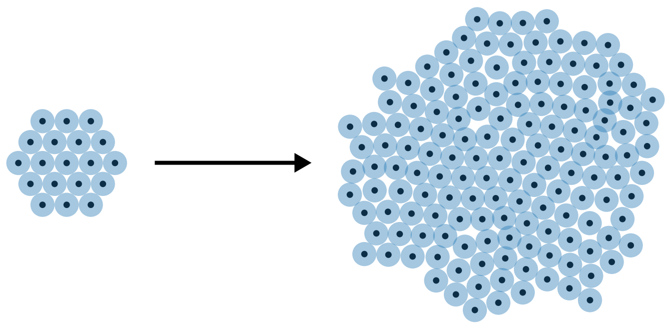
Figure 6: Left: initial cell configuration. Right: cell simulation after 20 h

The choice of time-step was determined by the maximum simulation time that we wanted to test and a brief convergence test. The test started with two cells, 0.3 unit apart. They were then let to relax until the simulation time 0.3 h. The simulation uses the cubic force function with default parameters and the Euler forward solver algorithm. The error was estimated by solving using a time-step 2 magnitudes smaller than the smallest time-step and using that as the reference solution. The rate of convergence was estimated by fitting the curve $C\Delta t^p$ and can be seen to be around 1 (Fig. 7). The solution appears to become unstable for time-steps around 0.1 h and ideally the chosen time-step should be smaller. As mentioned before, the time step was chosen to be 0.003 h in the experiment, which is within the stable region.

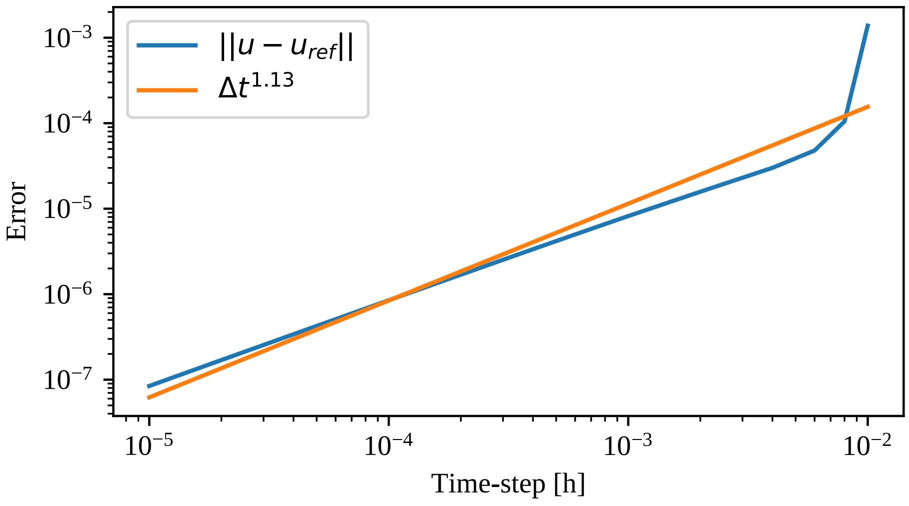

Figure 7: Testing convergence of the simulation. The test consists of placing two cells at the default separation distance after a division and letting them relax. The orange line is a fitted curve of $||u - u_{ref}|| = C\Delta t^p$, which shows that the rate of convergence to be close to 1 which is expected of the Euler forward algorithm.

4.3 Performance comparison

CbmosJulia performs considerably better than CBMOS in our experiments. Fig. 8 shows that the execution time grows exponentially for both versions. This is expected since the number of cells grows exponentially. CbmosJulia is consistently faster than CBMOS with one order of magnitude. The variations are reasonably small (the error bars represent two standard deviations), thus the estimate of execution time is rather stable. We notice a large variation at simulation time 30 h for CbmosJulia. This is because the warm-up in CbmosJulia is quite slow, the reason for which will be elaborated in Section 5. For some reason unknown to us, BenchmarkTools always started the benchmarking process at 30 h.

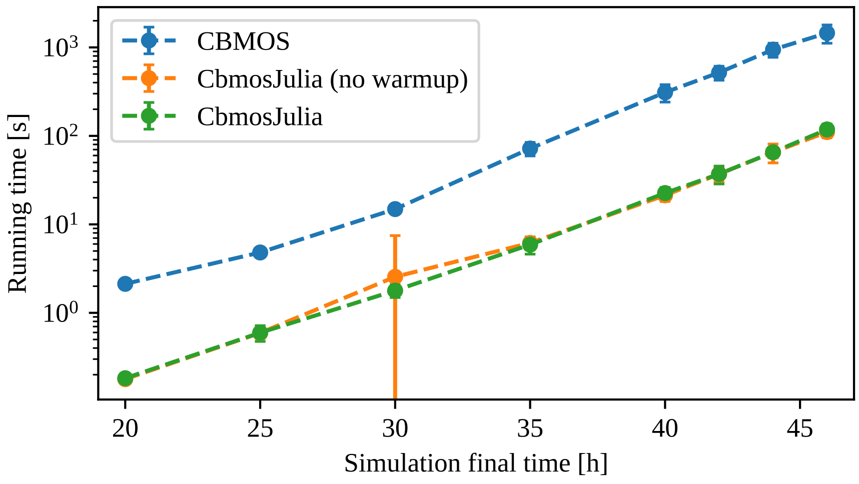

Figure 8: Running time comparison of CBMOS and CbmosJulia. The error bars represent two standard deviations. The effect of the JIT compilation can be seen when the Julia benchmark is run without warming up. It starts with the 30 h simulation which has a much higher standard deviation and a higher mean.

Fig. 9 illustrates the relative speedup of CbmosJulia in comparison with CBMOS. The upper bound of the blue region (best case scenario) is determined by the minimum runtime of CbmosJulia and the maximum runtime of CBMOS; the lower bound (worst case scenario) is decided by the maximum runtime of CbmosJulia and the minimum runtime of CBMOS. The relative speedup of CbmosJulia is around 10x, and it stays quite stable until the maximum simulation time we were able to test.

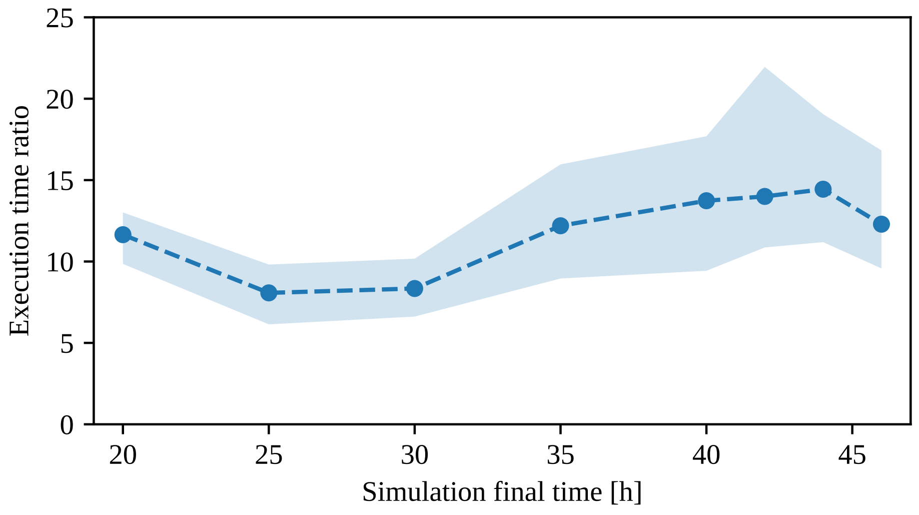

Figure 9: Relative speedup of CbmosJulia compared to CBMOS the blue region represents the best- and worst-case scenario. Where the top of the blue region is the minimum runtime of CbmosJulia divided by the maximum runtime of CBMOS and the bottom is the maximum runtime of CbmosJulia divided by the minimum runtime of CBMOS. The line in the centre represents the division of the mean execution times.

5 Conclusions and discussions

Julia has a lot of attractive features for working with scientific computing. The syntax is similar to that of Matlab, and it has powerful modules in its standard library for linear algebra applications. When the standard library is not enough, it has a package infrastructure that makes it easy to integrate other modules into your code. It has a growing ecosystem of packages for doing numerical work. Thanks to the JIT-compiler, missing features can be written in Julia, and they perform on a similar level as a C-extension written for Python.

Multiple dispatch is a powerful programming paradigm when working with mathematics. It functions similarly to how mathematical operators work, where the type of execution of an operation differs depending of the “types” of its inputs. It is easy to produce clean code in Julia when defining additional methods of a function.

Python is mature with more third-party packages. As a general-purpose language, it is capable of almost anything. Julia, on the other hand, is more focused and caters better to the niche. There is, however, the possibility of enjoying the benefits of both languages. Julia can interface with Python code through the PyCall library and even share data seamlessly between Python and Julia. Nevertheless, there is an overhead cost as we observe a slowdown when using CBMOS in Julia through PyCall.

When programming in Python, much of the performance concern is handled by external libraries. For example, we should write vector-like syntax using Numpy, rather than native for-loops. However, in order to take advantage of Julia’s high performance, users need to be mindful of the design choices. Depending on the use case, devectorised implementation might be faster than vectorised version. Experimentation is often needed in order to find the best available implementation.

The main drawback of using Julia is the time it takes for code to JIT-compile. Specially the "time to first plot", when plotting. For the initial hexagon cell configuration, the respective time to first plot is 7.749 s in Julia and 0.408 s in Python on a MacBook Pro with a 2.3 GHz Intel Core i9 processor. While adding the necessary dependencies only makes a marginal difference in Python (0.485 s), it results in a much longer start up time in Julia (18.118 s). The plotting libraries are also not as featureful as its Python equivalents. With changes to the workflow these problems can be mitigated, but not removed. As we mentioned in Section 2.3, it is important to keep Julia alive for multiple runs. In short, Python is a better language for fast prototyping.

Our results suggest that CbmosJulia performs much better than the Python counterpart. Though we think that the results reflect the performance, there are a few improvements for future experiments. For instance, the parameters of the force function could have been set such that the equations are less stiff, which in turn results in higher accuracy with the same time-step. The running time is highly dependent on the number of cells. To reduce the variance of the number of cells between the two versions at the sample final times, we could sample the final times at time points between division peaks. Additionally, we could increase the mean division cycle time to leave a time window where divisions are rare, which would decrease the variance even more. However, this will require a different set of final times. For future research, the potential performance impact of other parameters such as choice of force function could be investigated further.

We expect that there will be an increase of interest and adoption of the Julia language. Though there is room for improvement especially with regard to the time to first plot, it stands out amongst the open-source programming languages for the ease of use and its high performance.

6 Acknowledgement

We would like to thank our supervisors Sonja Mathias and Adrien Coulier for their valuable contribution in the form of weekly discussion and feedback. Without their input throughout the duration of the project, this report would not have been possible.

References

[1] Stackoverflow. *Developer Survey Results 2018*. https://insights.stackoverflow.com/survey/2018. Accessed: 2019-12-11.

[2] Stackoverflow. *Developer Survey Results 2019*. https://insights.stackoverflow.com/survey/2019. Accessed: 2019-12-11.

[3] MIT News. *Julia language co-creators win James H. Wilkinson Prize for Numerical Software*. https://news.mit.edu/2018/julia-language-co-creators-win-james-wilkinson-prize-numerical-software-1226. Accessed: 2020-01-03.

[4] Julia Computing. *Newsletter January 2019*. https://juliacomputing.com/blog/2019/01/04/january-newsletter.html. Accessed: 2020-01-03.

[5] *SciPy.org*. https://www.scipy.org/. Accessed: 2020-01-03.

[6] James M Osborne et al. "Comparing individual-based approaches to modelling the self-organization of multicellular tissues". In: *PLoS computational biology* 13.2 (2017), e1005387.

[7] Paul Van Liedekerke et al. "Simulating tissue mechanics with agent-based models: concepts, perspectives and some novel results". In: *Computational particle mechanics* 2.4 (2015), pp. 401–444.

[8] Garrett M Odell et al. "The mechanical basis of morphogenesis: I. Epithelial folding and invagination". In: *Developmental biology* 85.2 (1981), pp. 446–462.

[9] Frank M. White. *Fluid Mechanics*. McGraw-Hill, 2011.

[10] Edward M Purcell. "Life at low Reynolds number". In: *American journal of physics* 45.1 (1977), pp. 3–11.

[11] *CompuCell3D homepage*. http://www.compucell3d.org/. Accessed: 2020-01-30.

[12] *TiSim homepage*. https://www.hoehme.com/software/9-software/9-tisim. Accessed: 2020-01-30.

[13] *Chaste homepage*. http://www.cs.ox.ac.uk/chaste/. Accessed: 2020-01-30.

[14] *PhysiCell homepage*. http://physicell.org/. Accessed: 2020-01-30.

[15] JuliaLang. *Allowed Variable Names*. https://docs.julialang.org/en/v1/manual/variables/\url{\#}Allowed-Variable-Names-1. Accessed: 2020-01-25.

[16] JuliaLang. *Noteworthy Differences from other Languages*. https://docs.julialang.org/en/v1/manual/noteworthy-differences/\url{\#}Noteworthy-differences-from-Python-1. Accessed: 2019-12-15.

[17] JuliaLang. *Performance Tips*. https://docs.julialang.org/en/v1/manual/performance-tips/. Accessed: 2019-12-20.

[18] *Introducing Julia/Metaprogramming*. https://en.wikibooks.org/wiki/Introducing_Julia/Metaprogramming. Accessed: 2020-01-14.

[19] Arch D. Robison. *Vectorization in Julia*. https://software.intel.com/en-us/articles/vectorization-in-julia. Accessed: 2020-01-14.

[20] Christopher Rackauckas and Qing Nie. "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: *The Journal of Open Research Software* 5.1 (2017). Exported from https://app.dimensions.ai on 2019/05/05. DOI: 10.5334/jors.151. URL: https://app.dimensions.ai/details/publication/pub.1085583166%20and%20http://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/.

[21] JuliaLang. *Julia 1.3 Documentation*. https://docs.julialang.org/en/v1/. Accessed: 2020-01-25.

[22] JuliaLang. *Modules*. https://docs.julialang.org/en/v1/manual/modules/index.html. Accessed: 2020-01-30.

[23] JuliaDiffEq. *Ordinary Differential Equations*. https://https://docs.juliadiffeq.org/stable/tutorials/ode_example/. Accessed: 2020-01-04.

[24] JuliaDiffEq. *Callback Library*. `https://docs.juliadiffeq.org/stable/features/callback_library/\url{\#}IterativeCallback-1`. Accessed: 2020-01-04.

[25] Thomas Breloff. *RecipesBase.jl*. `https://github.com/JuliaPlots/RecipesBase.jl`. Accessed: 2020-01-14.

[26] JuliaPlots. *Recipes*. `https://docs.juliaplots.org/latest/recipes/`. Accessed: 2019-12-30.

[27] JuliaLang. *The Julia REPL*. `https://docs.julialang.org/en/v1/stdlib/REPL/`. Accessed: 2019-12-20.