SoftwareX 20 (2022) 101252

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

life^{x}: A flexible, high performance library for the numerical solution of complex finite element problems

Pasquale Claudio Africa

MOX, Department of Mathematics, Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133, Milano, Italy

ARTICLE INFO

Article history:
Received 22 August 2022
Received in revised form 14 October 2022
Accepted 27 October 2022

MSC:
35-04
65-04
65Y05
65Y20
68-04
68N30

Keywords:
High performance computing
Finite elements
Numerical simulations
Multiphysics problems

ABSTRACT

Numerical simulations are ubiquitous in mathematics and computational science. Several industrial and clinical applications entail modeling complex multiphysics systems that evolve over a variety of spatial and temporal scales. This study introduces the design and capabilities of life^{x}, an open source C++ library for high performance finite element simulations of multiphysics, multiscale, and multidomain problems. life^{x} meets the emerging need for versatile, efficient computational tools that are easily accessed by users and developers. We showcase its flexibility and effectiveness on a number of illustrative examples and advanced applications of use and demonstrate its parallel performance up to thousands of cores.

Code metadata

Current code version	v1.5.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00254
Code Ocean compute capsule	N/A
Legal Code License	LGPLv3
Code versioning system used	git
Software code languages, tools, and services used	C++ (standard $\geq$ 17), MPI, CMake $\geq$ 3.12.0
Compilation requirements, operating environments, and dependencies	deal.II $\geq$ 9.3.0, VTK $\geq$ 9.0.0, Boost $\geq$ 1.76.0
Link to developer documentation/manual	https://lifex.gitlab.io/lifex/
Support email for questions	pasqualeclaudio.africa@polimi.it

1. Motivation and significance

A broad range of applications in biology, medicine, physics, engineering, astronomy, energy, environmental, and material sciences can be described by multiple physical processes interacting at different spatial and temporal scales [1]. From the mathematical modeling perspective, such systems can be viewed as agglomerations of well-defined physics referred to as *core models*. This explains the emerging need to develop new universal computational frameworks for the numerical simulation of multiphysics, multiscale, and multidomain problems. Such tools should easily enable the realization of *in silico* experiments and provide a stable and intuitive development environment without compromising computational accuracy and efficiency.

The development of a tool of this kind plays a central role in decoupling the software development phase from the time-consuming process of performing different analyses – from

E-mail address: pasqualeclaudio.africa@polimi.it.

https://doi.org/10.1016/j.softx.2022.101252

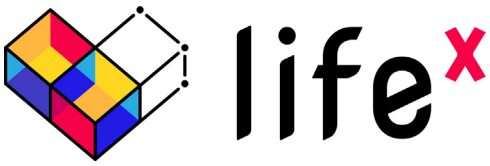

Fig. 1. life^x official logo. This image is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

forward simulations to sensitivity analysis, optimization, and uncertainty quantification – and in enabling the simulation of each core model in both standalone and various coupled configurations [2].

We introduce life^x (pronounced /ˌlaɪfˈɛks/, official logo shown in Fig. 1), an open source library for the numerical solution of partial differential equations (PDEs) and related coupled problems, released under the LGPLv3 license terms. It is written in C++ using modern programming techniques available in the C++17 standard and builds on the `deal.II` [3] finite element (FE) core. life^x aims at providing a flexible and intuitive but robust and high performance tool simplifying the definition of complex physical models and their parameters, coupling schemes, and post-processing.

life^x enables its users to shift the focus from technical numerics and implementation details toward the plain discrete mathematical formulation of the problems of interest. The library comes with extensive documentation and several examples and test cases that cover a wide range of applications and numerical strategies.

While serving similar purposes as existing multiphysics libraries in the open source community, such as `FEniCS` [4,5], `MFEM` [6], `MOOSE` [7], and `preCICE` [8], life^x offers several distinctive features, including the following:

- an intuitive user programming interface with extreme ease of use;
- modern programming paradigms by design, leveraging the C++17 standard, and up-to-date versions of third-party dependencies;
- parallel scalability up to thousands of cores;
- interoperability; that is, the possibility of importing and exporting data and meshes with common file formats, with particular reference to VTK;
- support for arbitrary FEs, among those available in the `deal.II` backend [9];
- the possibility to import meshes with either hexahedral or tetrahedral elements [3,10];
- a clean and meticulously documented code base.

Each of these features is outlined below.

2. Software description

life^x was conceived in 2019 as an academic research library within the framework of the iHEART project (see Acknowledgment) at the Politecnico di Milano, with a primary focus on mathematical models and numerical schemes for integrated simulations of cardiac function.

Since its initial design, many modules for the simulation of different core models have been added to the code base. The development of life^x was founded on strict coding conventions and practices [11]. The rapid increase in the number of developers and users testifies to the shallow learning curve of its kernel; it is fast and general enough to be used for diverse applications and merits being released as a standalone library.

Third-party dependencies of life^x include the following: deal.II (configured with support to PETSc [12] and Trilinos [13]), VTK, and Boost. life^x can be configured to use, by default, linear algebra data structures and algorithms from `PETSc`, `Trilinos` (either through the interfaces exposed by `deal.II` or directly) or `deal.II` itself; where needed, a specific datatype or solver provided by one of the three backends may also be hard-coded, disregarding the default type with which life^x was configured. All the code is natively parallel through the message passing interface (`MPI`); following a distributed memory paradigm, the global mesh is partitioned so that each `MPI` process owns and stores only a subset of cells.

This library aspires to maximum portability, having being deployed successfully on `Linux`, `Windows`, and `macOS` operating systems. This has motivated the use of advanced deployment technology, more specifically mk [14] (a set of portable, pre-compiled scientific packages for `x86-64 Linux` systems), life^x-env [15] (a set of build-from-source shell scripts explicitly inspired by candi), and Spack. Pre-built Docker images with all dependencies installed are also ready for download and use. More details can be found on the life^x documentation.

2.1. Software architecture

Structurally, the key features of life^x can be grouped into three main components:

1. An **abstraction layer** built on top of the `deal.II` FE library, exposing abstract numerical *helpers* as essential building blocks that foster the development of advanced data structures and numerical schemes for time integration, linearization, solving and preconditioning linear systems, imposing boundary conditions, and mesh handling.
2. A framework for **multiphysics coupling**, with functionalities enabling the transfer of solution fields and data from one core model to the other, either in the same domain or across multiple domains.
3. A seamless **user interface** through several advanced input/output (I/O) capabilities, with a focus on importing data coming from the post-processing of experimental results, imaging techniques, or other numerical simulations, such as with the help of the `VTK` library.

The main code components falling into these three categories, their classes, and their interactions are schematized in Fig. 2.

2.2. Software functionalities

All life^x executables are classified as follows:

`apps`: Generic applications that are not model-specific, such as tools for printing mesh statistics or converting between compatible file formats.

`examples`: Problems and solvers that define specific model or geometric parameters, such as boundary conditions, initial conditions, domain, and so on.

`tests`: Executables used for automatic testing (run via CTest), automatically run on continuous integration (CI) services at each `git push` on GitLab remote. Tests also include

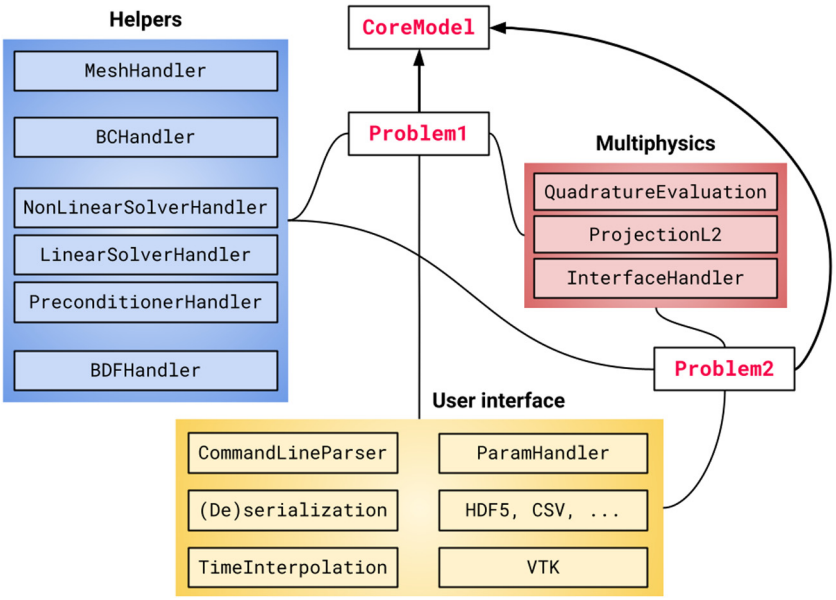

Fig. 2. Overview of main `life`^{x} components. The main classes and their interactions are shown, grouped into three categories: abstract numerical helpers (blue), multiphysics coupling (red), and user interface (yellow). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

a number of *tutorials*, which can be used as prototypes for building new applications. All tests and tutorials are used to determine the overall code coverage; that is, a metric that determines the number of lines of code that are successfully validated by the testing procedure.

Each `life`^{x} executable is typically associated with a set of common attributes, such as user-specified command line flags, the name of a parameter file (that is, a file containing all configurations, parameters, and settings used to run the executable, organized in a tree-like subsection structure), an *execution mode* flag that specifies whether to generate a new parameter file or actually execute the app, an output directory containing all output files, `MPI` rank and size used for parallel computations. Upon running, such attributes are shared among instances of all classes. Moreover, all main classes are designed so as to expose their own specific parameters, such as geometry, physical parameters, discretization schemes, numerical settings, and I/O options, from the parameter file, each within its own subsection path.

The following three main classes define the minimal kernel interface common to all `life`^{x} modules and executables:

`Core`: A class implemented following the *singleton* design pattern [16] that stores attributes that are global and common to all other classes, such as those listed directly above.

`CoreModel`: An abstract class that inherits `Core` and extends it with pure virtual methods that define the interface exposed by each core model or numerical solver throughout `life`^{x}. Classes in the `CoreModel` hierarchy expose a set of parameters that configure their behavior. Such parameters are exposed to the user through the parameter file (see Section 2.2.3). A sample code snippet is provided and discussed in Section 2.3.

`lifex_init`: A lifespan handler that takes care of properly initializing all attributes and dependencies needed by each run, such as the instance of the singleton `Core` and `MPI`; an instance of this class is typically constructed at the very beginning of the `main()` function and destroyed at the program's end.

More specific high-level data structures are introduced below.

2.2.1. Abstract numerical helpers

An enormous part of `life`^{x} consists of abstract wrappers and helpers: most of these classes explicitly invoke or refer to `deal.II` design and features [9], with the goal of exposing a higher-level interface to them and facilitating the implementation of advanced numerical schemes for a given problem. The main classes are described below.

`MeshHandler`: A wrapper around `deal.II` distributed meshes. The user can select whether to import a mesh with hexahedral or tetrahedral elements; depending on that choice, this class owns an instance of a `distributed` or a `fullydistributed` triangulation from `deal.II`; the latter is a recent introduction that adds support to tetrahedral meshes [3], whose functionalities at the time of writing are still to be consolidated. The `MeshHandler` class interacts closely with `MeshInfo`, which parses information from the input mesh like volume and surface tags to be used, for example, to impose different boundary conditions on different parts of the boundary or to differentiate material properties in different sub-regions. Helper functions implemented in the `geometry/mesh_info` and `geometry/finders` modules allow the computation of (sub)domain volumes and boundary surfaces or to locate, for example, the closest degree of freedom (DoF), mesh vertex, or boundary face to a given input point.

`BCHandler`: A helper class to impose different types of boundary conditions. Dirichlet boundary conditions can be either applied directly to an FE vector or imposed as linear constraints to the linear system arising from an FE discretization. For vector problems, normal or tangential fluxes can also be imposed. A helper method to assemble Neumann and Robin-like contributions to a local system's right-hand side is also provided.

`LinearSolverHandler`: For a sparse, distributed linear system, this class provides a simple interface that enables the user to select at run time which linear solver to use and all of its options (for instance, maximum number of iterations, tolerances, stopping criteria, and history log), as parsed from

the parameter file. Many common solvers are included, such as `CG`, `GMRES`, `BiCGStab`, `MinRes`, `FGMRES`, but in principle any solver exposed by `deal.II` (including those from `PETSc` and `Trilinos`) is supported. Furthermore, the complete suite of solvers from `PETSc` remains accessible via the `-options_file` command line flag, forwarded from `life`^{x} to `PETSc`.

`PreconditionerHandler`: Analogously to `LinearSolverHandler`, this class exposes parameters that are used for the preconditioning of linear systems. It supports many preconditioner types, such as algebraic multi-grid (AMG), block Jacobi, additive Schwarz (SOR, SSOR, block SOR, block SSOR, ILU, ILUT), and can easily be extended to support more.

`BDFHandler`: For time-dependent problems, semi-implicit backward difference formula (BDF) time discretization schemes [17] are implemented in this class, which deals with storing the information to advance the problem from one time step to the next. This class stores and exposes the BDF solution and its extrapolation and can easily be extended to different time-advancing schemes.

`NonLinearSolverHandler`: For solving non-linear problems, a family of Newton methods is provided. An abstract implementation requires the user to specify an *assemble function*, which assembles the Jacobian matrix and the residual vector, and a *solve function* that assembles the preconditioner and solves the linear system associated with each non-linear iteration; the two functions must return the norms of residual, solution, and Newton increment to be used as possible stopping criteria. The *frozen Jacobian* (or *Jacobian lagging*) approach [18], which consists of reassembling the Jacobian only once every n time steps, can be toggled to increase computational efficiency. Two specializations for the quasi-Newton method with the Jacobian matrix approximated via finite differences [19] and for the inexact Newton method [20] are also supplied. Moreover, each non-linear solution scheme can be equipped with proper acceleration strategies (static relaxation, Aitken extrapolation [21], and Anderson acceleration [22]) to accelerate convergence. In addition to the non-linear solver handler, the user can benefit from the use of *automatic differentiation* (with support for the Sacado and ADOL-C interfaces exposed by `deal.II`), demonstrated on `Tutorial04_AD` and `Tutorial07_AD`, which enables implementing the computation of exact derivatives (up to machine precision) of complicated functions very easily.

2.2.2. Multiphysics coupling

The complexity of multiphysics, multiscale, and multidomain problem can be relieved with the help of three hierarchies of classes that all serve the purpose of transferring solution fields and data either from one core model to another or across internal interfaces.

In order to keep the code as general as possible, we assume that different core models can be solved using arbitrarily independent discretization schemes, such as different FE degrees or mesh resolutions. This improves the capturing of all physical phenomena involved, even though their dynamics can be characterized by vastly different spatial and temporal scales.

We note that problems involving more than one physical model (possibly on multiple domains sharing a common interface, such as in the case of fluid–structure interaction) can be generally solved using either monolithic or partitioned algorithms [23], as schematized in Fig. 3. In the former case, a global system involving all unknowns from all problems is assembled and solved at each time step; in the latter, each sub-problem is solved independently, and coupling conditions are imposed, for example by using explicit schemes or sub-iterating with a fixed-point scheme until a convergence of coupling conditions is reached. Both choices are possible in `life`^{x} and illustrated by a number of examples and tests.

`QuadratureEvaluation`: This class provides a high-level interface for the evaluation of arbitrary analytic functions or more complex data structures at a given quadrature point. User-defined classes deriving from `QuadratureEvaluation` can easily be implemented for scalar, vector, or tensor fields. Furthermore, the `QuadratureEvaluationFEM` hierarchy of classes is implemented to enable the coupling of multiple FE models solved in the same domain. Different problems can be discretized using different FE degrees, and the integrals arising from the weak formulation can be approximated using quadrature formulas of different types and degrees of accuracy. The `QuadratureEvaluationFEM` classes provide an interface similar to that of `FEValues` from `deal.II`: such objects are constructed using the `DoFHandler` associated with the FE field to be evaluated and the quadrature rule used for the target problem. By re-initializing such objects on each mesh cell, the input field can be evaluated at the corresponding quadrature points. `life`^{x} provides specializations to automatically evaluate the FE solutions, gradients, and divergence of a given solution vector.

`ProjectionL2`: Instead of the exact numerical evaluation allowed by `QuadratureEvaluation` classes, a smoothed L^2 projection can be considered. Given a function $f(\mathbf{x})$, this class computes a FE solution $f_h(\mathbf{x})$ that satisfies $(\varepsilon\nabla f_h, \nabla\varphi_i)_\Omega + (f_h, \varphi_i)_\Omega = (f, \varphi_i)_\Omega$ for each basis function φ_i in the chosen FE space. The numerical solution to this problem clearly involves a mass matrix: its lumping can be toggled, and the regularization parameter ε can be tuned to prevent numerical oscillations, for example in the case of coarse meshes [25]. The solution f_h obtained can thus easily be evaluated at the quadrature nodes associated with the target problem.

`InterfaceHandler`: Consider two subdomains Ω_1 and Ω_2 sharing a common interface Σ with conforming discretizations, and let u_1 and u_2 be FE functions defined on the two subdomains, typically representing solutions to differential problems defined on the two subdomains. Suppose that the problem defined on Ω_1 (Ω_2) involves conditions on Σ that depend on u_2 (u_1) [23,26]. `InterfaceHandler` builds the *interface maps*; that is, two mappings of DoFs between the local interface Σ and the global domains Ω_1 and Ω_2. This is of critical importance in parallel simulations, where the parallel partitioning on both domains can be different, as in the example in Fig. 4. Finally, for each subdomain, this class manages the extraction of interface data on Σ from the other subdomain and its application as a boundary condition on Σ. This class deals only with conforming meshes; extensions to non-conforming discretizations, such as the `INTERNODES` technique [27], are still under development.

The last case to be considered is transferring solutions between multiple core models solved using the same FE discretization but with different mesh resolutions. For nested hexahedral grids, the `VectorTools` namespace of `deal.II` already provides functions that perform precisely the interpolation needed. This procedure is hardly generalizable as it depends heavily on how

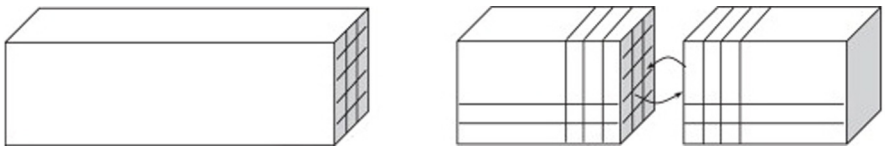

Fig. 3. Possible solution schemes for a geometrically coupled problem: monolithic (left) vs. partitioned (right) solution scheme. Reprinted from [24]. The original image is licensed under a CC BY 3.0 License.

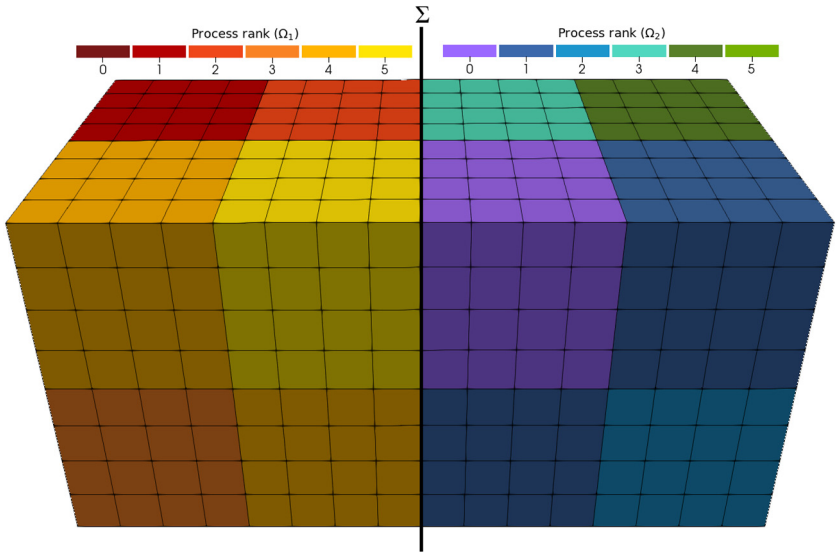

Fig. 4. Example of handling two domains Ω_1 (left) and Ω_2 (right) sharing a common interface Σ with conforming mesh discretizations. The `InterfaceHandler` is able to deal properly with non-conforming parallel partitioning.

the different meshes have been generated and on the mesh element type. For instance, transfer operators built on radial basis function (RBF) interpolators could be used in the case of non-conforming discretizations [28,29] but have yet to be implemented.

Clearly, the two approaches can be combined to couple different models solved with both different FE approximations and mesh resolutions.

2.2.3. User interface

`CommandLineParser`: `life`^{x} makes use of the lightweight parser clipp for parsing command line arguments. All executables expose a set of command line options that can be printed using the `-h` (or `--help`) flag:

```
./executable_name -h
```

`ParamHandler`: Each `life`^{x} executable defines a set of parameters that are required in order to be run. They involve problem-specific parameters (such as coefficients, geometry, time interval, and boundary conditions), numerical parameters (such as types of linear/non-linear solvers, tolerances, and maximum number of iterations), I/O options, and so on. In the event an application has sub-dependencies such as a linear solver, the related parameters are also included, typically in a proper subsection path. Parameters are organized in a tree-like structure following the functionalities exposed by the `ParameterHandler` class from `deal.II`. The first step before running any executable is to generate the default parameter file(s) via the `-g` (or `--generate-params`) flag:

```
./executable_name -g -f filename.ext
```

At the user's option, in order to guarantee a flexible interface with external file processing tools, the parameter file extension `ext` can be chosen among three different interchangeable file formats `prm`, `json` or `xml`, sorted from the most human-readable to the most machine-readable.

An excerpt of a `prm` file follows:

```
subsection Problem
  subsection Mesh and space discretization
    # Parameter description goes here.
    set Element type = Hex
    # ...
  end
  # ...
  subsection Linear solver
    set Type = GMRES

    subsection GMRES
      set Max. number of temporary vectors = 100
      # ...
    end
  end
  # ...
  subsection Preconditioner
    set Type = AMG

    subsection AMG
      set W-cycle = true
      # ...
    end
end
```

Listing 1: Example of parameter file in `prm` format. The tree-like subsection structure is emphasized.

A parameter file can easily be set up using any text editor, without need to recompile the source code. Finally,

omitting the -g flag in the command above, an existing parameter file is read and the simulation subsequently run.

The ParamHandler class of lifex extends the deal.II class by two main functionalities:

verbosity control: By default, only parameters declared to have a *standard* verbosity are printed. In order to customize the user experience, the verbosity of each parameter can be decreased (*minimal*) or increased (*full*) from the source code. A parameter file containing a minimal (full) set of parameters can be generated by passing the optional flag minimal (full) to the -g flag:

```
./executable_name -g [minimal,full] \
                  -f filename.ext
```

If the -g is provided without any further specification, the intermediate level of verbosity is assumed.

multiple default values: In principle, each application could be run to simulate different scenarios or simply with different predefined sets of parameters; lifex offers the possibility of providing multiple default parameter files out of the box. The ParamHandler class can read user-provided files in json format by specifying a list of parameter names and their (new) default values, which will be appended to the complete set of parameters and written to a ready-to-use file (see, for example, the time_interpolation test).

Utilities for parsing lists of values are also provided in the param_handler_helpers module for convenience of use.

(De-)serialization: lifex includes a checkpointing system that allows for all aspects of a simulation to be serialized to file. This allows recovering a simulation state after an unexpected failure, restarting after maximum computational wall time has been reached, or simply initializing a simulation with custom input data. Convenient tools for (de-)serializing distributed meshes and solution vectors are provided in the io/serialization module, with an interface to deal.II-compatible binary files, and their use is demonstrated in the serialization test.

CSV readers and writers: The simplicity of use of comma-separated value (CSV) files makes it a widely chosen option to process data organized into fields. Many utility functions and classes are present in lifex to read and write CSV files by converting number and text values into STL containers or deal.II data structures (vectors, matrices, and so on). This enables easily post-processing simulation results, for example by exporting point-wise variables at each time step.

TimeInterpolation: Many applications require resampling discrete sets of data at arbitrary points, such as time-dependent variables that need to be interpolated in correspondence with the time steps performed by the numerical simulation. The TimeInterpolation class provides methods based on linear interpolation, cubic splines, smoothing cubic splines, trigonometric interpolation (discrete Fourier transform), and linear and spline interpolation of the derivative of the input data.

VTKFunction and VTKPreprocess: Many physical problems are characterized by coefficients derived from experimental data or imaging techniques, such as segmented geometries of organs from magnetic resonance imaging (MRI) or computer tomography (CT) scans [30,31], or from post-processing of other numerical simulation steps [32]. The VTK toolkit defines some of the most common data formats to deal with data defined over volumes (vtkUnstructuredGrids) or surfaces (vtkPolyData). Moreover, it is also used in sophisticated pipelines for surface processing and mesh generation [33]. lifex provides a class named VTKFunction, inherited from dealii::Function, that imports a VTK file containing a cell or point data field and evaluates it at an arbitrary point, possibly associated with a computational mesh. Three possible evaluation methods are available: closest point, linear projection, and signed distance. Finally, the VTKPreprocess class exploits VTKFunction to interpolate input VTK data onto FE vectors, which are serialized to file for later importing and reuse in numerical simulations.

2.3. Sample code snippet

The following code illustrates a sample code snippet with comments, containing the minimal interface exposed by the vast majority of all lifex classes; that is, those inherited from CoreModel. In particular, the declare_parameters and parse_parameters methods are *pure virtual* and must be overridden, whereas the run method is virtual and has an empty definition by default. An example of how to locally adjust the verbosity of some parameters is also shown. Finally, this sample class makes use of a LinearSolverHandler, for which we also declare and parse related parameters.

```
namespace lifex
{
  class Problem : public CoreModel
  {
  public:
    // Specify the subsection path where to
    // declare current parameters.
    Problem(const std::string &subsection_path)
    : CoreModel(subsection_path)

    // Specify a "relative" subsection.
    // Subpaths are separated by a "/".
    , linear_solver(
        prm_subsection_path + " / Linear solver",
        /* ... */)
    {}

    virtual void
    declare_parameters(ParamHandler &params) const
    override
    {
      // Navigate subsections and declare parameters.
      params.enter_subsection_path(prm_subsection_path
    );
      {
        // Problem-dependent parameters.
        // ...

        params.set_verbosity(VerbosityParam::Full);
        {
          // If -g full is *not* specified,
          // the parameters declared here will
          // be hidden from the parameter file.
          // ...
        }
        params.reset_verbosity();
      }
      params.leave_subsection_path();

      linear_solver.declare_parameters(params);
    }

    virtual void
    parse_parameters(ParamHandler &params) override
```

```
    {
      // Actually parse parameter file.
      params.parse();

      // Analogously to declare_parameters,
      // navigate subsections, read parameters,
      // and possibly store them into class members.
      // ...

      linear_solver.parse_parameters(params);
    }

    virtual void
    run() override
    {
      // Create mesh.
      // Setup system.
      // Assemble system.
      // Solve system.
      // Output solution.
    }

  private:
    LinearSolverHandler linear_solver;
    // ...
  };
}
```

3. Illustrative examples

`life`^{x} is capable of solving complex multiphysics problems. The functionalities described in the previous section are pointed out in a series of *tutorials* that are found in the source code as tests. The tutorials are sorted by increasing complexity and involve different kinds of scalar or vector equations and coupled problems, solved either monolithically or partitioned. Here, we provide a summary of the tutorials available and the corresponding PDEs solved.

`Tutorial01`: Linear elliptic equation:

$$-\Delta u = f, \quad \text{in } (-1,1)^3.$$

`Tutorial02`: Linear parabolic equation:

$$\frac{\partial u}{\partial t} - \Delta u + u = f, \quad \text{in } (-1,1)^3 \times (0,T].$$

`Tutorial03`: Non-linear elliptic equation:

$$-\Delta u + u^2 = f, \quad \text{in } (-1,1)^3.$$

`Tutorial04`: Non-linear parabolic equation:

$$\frac{\partial u}{\partial t} - \Delta u + u^2 = f, \quad \text{in } (-1,1)^3 \times (0,T].$$

`Tutorial04_AD`: The same as `Tutorial04`, with the Jacobian matrix assembled via automatic differentiation.

`Tutorial05` Parabolic system of equations, solved monolithically:

$$\begin{cases} \dfrac{\partial u}{\partial t} - \Delta u + u^2 = f, & \text{in } (-1,1)^3 \times (0,T], \\ \dfrac{\partial v}{\partial t} - \Delta v + uv = g, & \text{in } (-1,1)^3 \times (0,T]. \end{cases}$$

`Tutorial06`: The same as `Tutorial05`, solved using an explicit partitioned scheme and exploiting the `QuadratureEvaluationFEM` capabilities.

`Tutorial07`: Cahn-Hilliard equation:

$$\begin{cases} \dfrac{\partial c}{\partial t} - \Delta \mu = 0, & \text{in } (0,1)^3 \times (0,T], \\ \mu - \dfrac{\mathrm{d}f}{\mathrm{d}c}(c) + \lambda \Delta c = 0, & \text{in } (0,1)^3 \times (0,T]. \end{cases}$$

For further details about the mathematical and numerical formulations of all these problems, such as boundary and initial conditions, please refer to the `life`^{x} documentation.

We present below three examples that showcase the main features of `life`^{x}. All the results shown are new, original contributions. First, we prove that the abstract helpers for the advanced numerical schemes described in Section 2 do not affect parallel performance, as the speedup is almost approximately linear up to thousands of cores; then, we present a multidomain problem where two Stokes problems are solved on two cubes sharing a common face with proper interface conditions, proving that monolithic and partitioned schemes for domain decomposition problems can easily be implemented with a negligible computational overhead due to the parallel transfer of solutions across the interface; finally, an advanced, fully implicit numerical solver for the Cahn–Hilliard equation demonstrates the ease of implementation and the enormous flexibility available to users when dealing with complex multiphysics problems.

3.1. Scalability study

We perform a strong scaling test on `Tutorial06`, where the following equations are solved:

$$\begin{cases} \dfrac{\partial u}{\partial t} - \Delta u + u^2 = f, & \text{in } \Omega \times (0,T] = (-1,1)^3 \times (0,T], \\ \dfrac{\partial v}{\partial t} - \Delta v + uv = g, & \text{in } \Omega \times (0,T], \\ u = u_{\text{ex}}, & \text{on } \partial\Omega \times (0,T], \\ v = v_{\text{ex}}, & \text{on } \partial\Omega \times (0,T], \\ u = u^0, & \text{in } \Omega \times \{0\}, \\ v = v^0, & \text{in } \Omega \times \{0\}, \end{cases}$$

where f, g, u^0, and v^0 are chosen such that the exact solution is

$$\begin{cases} u_{\text{ex}}(\mathbf{x},t) = t\cos(\pi x_0)\cos(\pi x_1)\cos(\pi x_2), \\ v_{\text{ex}}(\mathbf{x},t) = e^t \|\mathbf{x}\|^2. \end{cases}$$

The two equations are discretized in time using the `BDFHandler` of order 1 for u and 3 for v, decoupled using an explicit partitioned scheme, and linearized using the `NonLinearSolverHandler` class. Finally, the FE space discretization consists of linear (quadratic) elements for u (v). The solution u appearing in the second equation is evaluated using the capabilities of `QuadratureEvaluationFEM`. The mesh size consists of 2,097,152 cells (average cell diameter: $h \approx 0.027$) and 19,121,282 DoFs (2,146,689 for u, 16,974,593 for v), the time step chosen is equal to $\Delta t = 0.1$, and the simulation is run until $T = 1$.

The scalability test was run on the GALILEO100 supercomputer available at `CINECA` (Intel CascadeLake 8260, 2.40 GHz). We recorded the total simulation time and partial times spent in the assembly and linear-solving phases; the speedup for the three quantities shown in Fig. 5 confirms that on such a benchmark problem, the main `life`^{x} data structures scale approximately linearly up to 4096 cores. The linear solver performances slowly degrades beginning at about 512 cores, which is likely due to the limited problem size. Table 1 reports the summary of the computational costs for the different phases of a run of `Tutorial06` on 1024 cores.

Table 1
Summary of computational costs of a run of `Tutorial06` on 1024 cores.

Section	No. calls	Wall time	% of total
Solver for u: solve time step	11	31.515 s	3.05%
Solver for u: non-linear solver	11	29.460 s	2.85%
Solver for u: preconditioner assembly + linear solver	33	25.907 s	2.51%
Solver for u: system assembly	44	3.365 s	0.33%
Solver for u: linear solver	33	1.799 s	0.17%
Solver for v: solve time step	11	988.561 s	95.79%
Solver for v: system assembly	11	956.339 s	92.67%
Solver for v: `QuadratureEvaluationFEM` initialization	11	0.000 s	0.000%
Solver for v: `QuadratureEvaluationFEM` re-initialization	22,528	0.020 s	0.000%
Solver for v: `QuadratureEvaluationFEM` evaluation	443,418,624	278.760 s	27.32%
Solver for v: preconditioner assembly + linear solver	11	23.601 s	2.29%
Solver for v: linear solver	11	13.897 s	1.35%
Total wallclock time		1031.991 s	100%

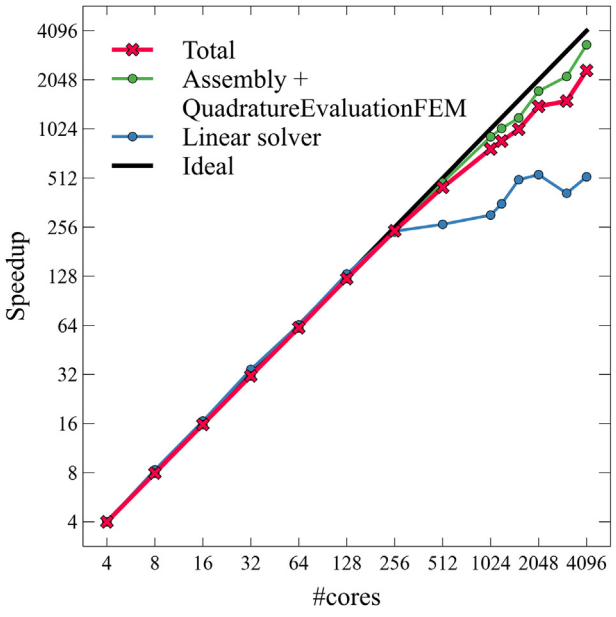

Fig. 5. Parallel speedup of `life`^{x}, demonstrated on `Tutorial06`. The speedup was computed on total time (red), time spent in assembling the linear system (including the evaluation of the `QuadratureEvaluationFEM` field) at each time step (green), and time spent in solving the linear system at each time step through the `LinearSolverHandler`, together with the `PreconditionerHandler` wrappers (blue). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Some interesting conclusion can be drawn. First, the evaluation of u at quadrature nodes of the FE space used for the discretization of the equation for v is invoked ≈ 443 million times, which makes a substantial contribution to the assembly phase (about 27% of the total time): nevertheless, as Fig. 5 shows, the assembly phase still scales almost perfectly linearly, which proves that the implementation of the `QuadratureEvaluation` hierarchy of classes introduces a computational overhead that scales almost ideally in parallel. Moreover, the additional overhead from applying the `BDFHandler`, `NonLinearSolverHandler`, `LinearSolverHandler`, and `PreconditionerHandler` wrappers is negligible and does not affect the solver's overall parallel performance. This shows that `life`^{x} can reach an ideal parallel speedup while the abstract numerical helpers and multiphysics coupling interface enable a significant reduction in the total number of lines of code compared to a naive implementation based only on `deal.II`, thus letting the user focus on the plain discrete formulation of the problems of interest rather than on technical numerics and implementation details.

3.2. Multidomain problems

The `multidomain_stokes` example was run to demonstrate the parallel performance of the multidomain capabilities of `life`^{x}, with particular reference to the `InterfaceHandler` class.

The problem being solved is the Stokes model:

$$\begin{cases} -\mu\Delta\mathbf{u} + \nabla p = 0, & \text{in } \Omega, \\ \nabla\cdot\mathbf{u} = 0, & \text{in } \Omega, \\ \mathbf{u} = [1,0,0]^T, & \text{on } \Gamma_{\text{in}}, \\ \mathbf{u} = [0,0,0]^T, & \text{on } \Gamma_{\text{sides}}, \\ \mu\nabla\mathbf{u}\cdot\boldsymbol{\nu} - p\mathbf{n} = 0, & \text{on } \Gamma_{\text{out}}, \end{cases}$$

where

$$\begin{aligned} \Omega &= (-0.5, 1.5)\times(-0.5, 0.5)\times(-0.5, 0.5), \\ \Gamma_{\text{in}} &= \{x = -0.5\}, \\ \Gamma_{\text{out}} &= \{x = 1.5\}, \\ \Gamma_{\text{sides}} &= (\partial\Omega\backslash(\Gamma_{\text{in}}\cup\Gamma_{\text{out}}))^o, \end{aligned}$$

and $\boldsymbol{\nu}$ denotes the outward unit normal. Ω is split into the two subdomains Ω_0 and Ω_1 across the interface Σ, which is defined as

$$\begin{aligned} \Omega_0 &= (-0.5, 0.5)\times(-0.5, 0.5)\times(-0.5, 0.5), \\ \Omega_1 &= (0.5, 1.5)\times(-0.5, 0.5)\times(-0.5, 0.5), \\ \Sigma &= \{x = 0.5\}. \end{aligned}$$

Denoting the solution on the subdomain Ω_i by $\mathbf{u}_i$, p_i for $i = 0, 1$, the multidomain formulation of the problem is as follows:

$$\begin{cases} -\mu\Delta\mathbf{u}_i + \nabla p_i = 0, & \text{in } \Omega_i, \\ \nabla\cdot\mathbf{u}_i = 0, & \text{in } \Omega_i, \\ \mathbf{u}_0 = [1,0,0]^T, & \text{on } \Gamma_{\text{in}}, \\ \mathbf{u}_i = [0,0,0]^T, & \text{on } \Gamma_{\text{sides}}\cap\partial\Omega_i, \\ \nabla\mu\mathbf{u}_1\boldsymbol{\nu} - p\boldsymbol{\nu} = 0, & \text{on } \Gamma_{\text{out}}, \\ \mathbf{u}_0 = \mathbf{u}_1, & \text{on } \Sigma, \\ -\mu\nabla\mathbf{u}_0\boldsymbol{\nu}_0 - p_0\boldsymbol{\nu}_0, = \mu\nabla\mathbf{u}_1\boldsymbol{\nu}_1 + p_1\boldsymbol{\nu}_1 & \text{on } \Sigma, \end{cases}$$

where the two conditions on Σ denote the continuity of velocity and stresses across the interface.

This problem can be solved by using either a fixed-point or a monolithic scheme. The goal of this section is to demonstrate the performance of the `InterfaceHandler` class. We thus ran a simulation using 6,714,692 DoFs on each subdomain Ω_i for $i = 0, 1$ (6,440,067 for the velocity and 274,625 for the pressure block), resulting in a total number of DoFs at the interface Σ equal to 49,923. The simulation requires 40 fixed-point iterations to meet the prescribed tolerance of 10^{-6} on the increment norm of the interface data $\mathbf{u}_1|_\Sigma$.

Table 2
Summary of computational costs of a run of the example `multidomain_stokes` on 1024 cores.

Section	No. calls	Wall time	% of total
`InterfaceHandler`: build interface maps	1	24.1 s	0.17%
`InterfaceHandler`: extract interface values	80	2.38 s	0%
`InterfaceHandler`: apply Dirichlet interface conditions	80	0.875 s	0%
Problem in Ω_0: preconditioner assembly + linear solver	40	6.73e+03 s	48%
Problem in Ω_0: system assembly	40	42.2 s	0.3%
Problem in Ω_1: preconditioner assembly + linear solver	40	7.05e+03 s	51%
Problem in Ω_1: system assembly	40	41.2 s	0.3%
Total wallclock time		1.39e+04 s	100%

Table 2 reports the summary of the computational costs for the different phases of a run of the `multidomain_stokes` example on 1024 cores. As the number of interface DoFs is much smaller than the total number of volume DoFs, the time spent in setting up interface maps and transferring the solutions between the two subdomains Ω_0 and Ω_1 is negligible with respect to the phases of assembling and solving the associated linear systems. This demonstrates that the overall computational cost associated with a multidomain simulation, such as in a fluid–structure interaction (FSI) framework, is largely dominated by the time spent in assembling and solving the associated linear system(s), whereas the overhead introduced by the `InterfaceHandler` class is negligible.

As a consequence, the speedup results shown in Section 3.1 still hold for multidomain problem, solved either monolithically or partitioned, as long as the parallel partitioning of Ω_0 and Ω_1 is fairly load-balanced.

3.3. User interface flexibility

Finally, to demonstrate the ease of implementing new solvers for complex multiphysics problems exploiting the capabilities of `life`^{x}, we present the development of a spinodal decomposition model of a binary fluid undergoing shear flow using the advective Cahn–Hilliard equation, a stiff, non-linear, parabolic equation characterized by the presence of fourth-order spatial derivatives [34]. Spinodal decomposition consists of the separation of a mixture of two or more components to the bulk regions of both, which occurs, for example, when a high-temperature mixture of two or more alloys is rapidly cooled.

The equation was discretized by using FEs in mixed form, thus splitting it into a system of two parabolic–elliptic equations:

$$\begin{cases} \dfrac{\partial c}{\partial t} - \Delta\mu = 0, & \text{in } \Omega \times (0,T] = (0,1)^3 \times (0,T], \\ \mu - \dfrac{\mathrm{d}f}{\mathrm{d}c}(c) + \lambda \Delta c = 0, & \text{in } \Omega \times (0,T], \\ \nabla c \cdot \boldsymbol{\nu} = 0, & \text{on } \partial\Omega \times (0,T], \\ \nabla \mu \cdot \boldsymbol{\nu} = 0, & \text{on } \partial\Omega \times (0,T], \\ c = c_0(\mathbf{x}), & \text{in } \Omega \times \{0\}, \end{cases}$$

where $f(c) = 100c^2(1-c)^2$ and the initial condition is given by $c_0(\mathbf{x}) = 0.63 + +0.01 \sin(2000\pi x_1 x_2 x_3)$.

The problem is discretized in time using a fully implicit scheme through the `BDFHandler` and linearized using the `NonLinearSolverHandler` class; at each time step, the Jacobian matrix is computed exploiting automatic differentiation and the associated linear system is solved using GMRES with an AMG preconditioner. Fig. 6 shows the steady state solution over the computational domain.

Despite the complexity of the FE formulation of the above problem and the sophisticated numerical schemes adopted for its discretization, the effort involved in implementing such a new solver from scratch in `life`^{x} requires writing approximately 350 lines of `C++` code (excluding comments and blank lines). This is a strikingly small number, especially given the vast flexibility the user has in selecting many modeling and numerical features, such as choosing the FE degree, setting up the computational domain and physical parameters, changing the time discretization parameters (including the BDF order), selecting the type of linear solver and preconditioner and related parameters, and so on.

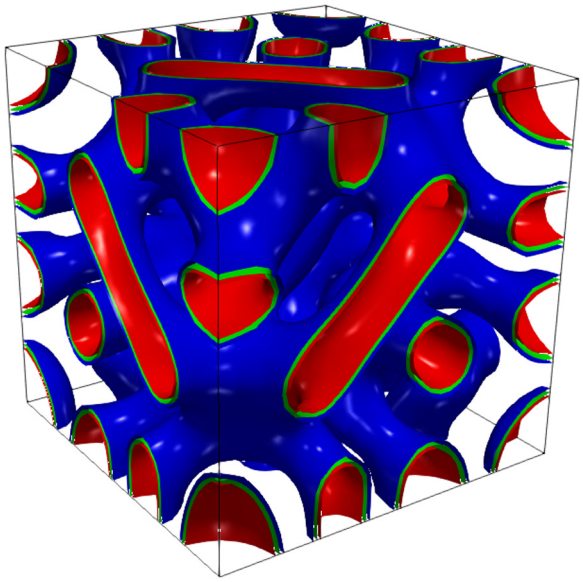

Fig. 6. Solution of the Cahn–Hilliard equation implemented in `Tutorial07`. Isosurfaces corresponding to values of the solution c equal to 0.35 (red), 0.5 (green), and 0.65 (blue) are shown. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4. Impact

The impact and wide applicability of `life`^{x} are demonstrated by the high number of journal articles, preprints, conference abstracts, and PhD theses that have already cited it. Computational studies carried out with `life`^{x} have appeared in a variety of fields, mostly originating from but not limited to cardiovascular modeling: cardiac electrophysiology [35–38], cardiac mechanics, electromechanics, and blood circulation [32,39–43], fluid dynamics [31,44–46], FSI [23], poromechanics coupled with blood perfusion [47,48], and hemodynamics in patients affected by COVID-19 [49].

Other notable computational models relying on `life`^{x} involve a comprehensive and biophysically detailed electromechanics of the entire human heart [50] (see Fig. 7) and an integrated description of electrophysiology, mechanics, and fluid dynamics in the human left heart [51,52]. Other studies oriented toward

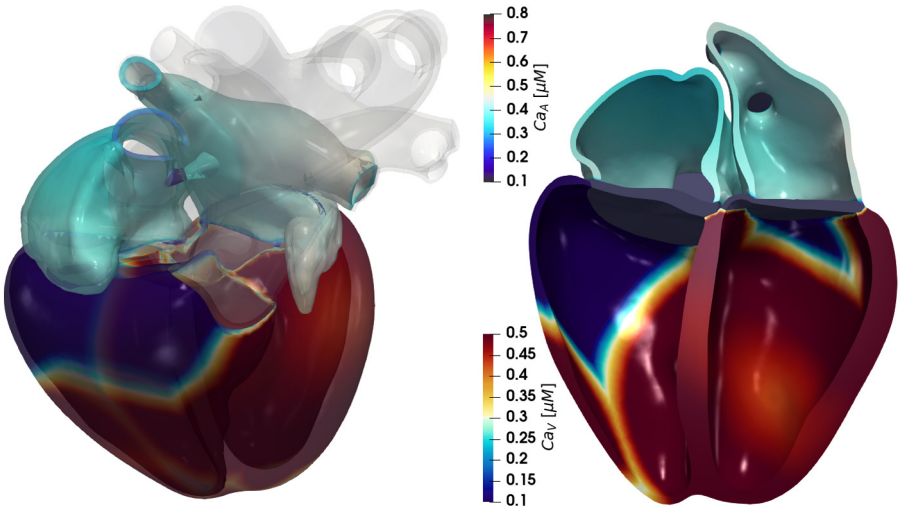

Fig. 7. Snapshot of a cardiac electromechanics simulation on a whole-heart geometry during ventricular systole. The color map shows the intracellular concentration of calcium ions in the myocardium. The computational mesh was generated from the Zygote Solid 3D Heart Model [57].

numerical methods have addressed the development of a high-order, matrix-free solver for cardiac electrophysiology [53] and reduced order methods for real-time simulations [54–56].

`life`^{x} provides a fast and stable environment with a gentle learning curve and enables obtaining unmatched results in terms of model reliability, numerical accuracy, and computational efficiency. A comprehensive and up-to-date list of publications making use of `life`^{x} can be found at https://lifex.gitlab.io/lifex/publications.html.

5. Conclusions

`life`^{x} is a parallel C++ library for simulations of multiphysics, multiscale, and multidomain problems based on the `deal.II` FE core. `life`^{x} offers several distinctive features, such as abstract helpers for advanced numerical schemes, a convenient framework to deal with several kinds of coupled problems, and a friendly interface that allows for the implementation of new complex FE solvers or sophisticated numerical schemes with all parameters selectable at run time with an economical number of lines of code.

The abstraction layer and the software functionalities presented in Section 2 show either high parallel speedup or negligible computational overheads. Overall, `life`^{x} shows a very high computational efficiency and seamless parallel performance; it is thus an invaluable tool that can be run on diverse architectures, ranging from laptop computers to HPC facilities and cloud platforms.

On the one hand, `life`^{x} provides a robust and friendly interface enabling easily accessible and reproducible *in silico* experiments, without any compromise in terms of computational efficiency and numerical accuracy. On the other, because it is conceived as a research library, `life`^{x} can be exploited by scientific computing experts to address new modeling and numerical challenges in an easily approachable development framework.

Future directions for `life`^{x} will include expanding its developer and user bases, maintaining an active and friendly community that welcomes new contributions, and making new advanced features openly available to the wider public (see, for example, [58,59]). In support of these goals, there are a number of technical areas that will open the way to many upcoming developments. First, when moving towards larger numbers of cores or DoFs, new bottlenecks are typically encountered, which will be addressed by carefully profiling all parts of the code base and exploiting more efficient algorithms, such as those based on matrix-free methods [53]. Similarly, large computer clusters will likely benefit from GPUs and similar devices in the near future, so targeting new architectures and porting algorithms to different programming models will be of utmost importance [9]. Other possible areas of improvement will target new advanced numerical techniques, such as higher-order or adaptive time advancing schemes, support of multidomain problems discretized over non-conforming meshes (exploiting, for instance, the INTERNODES technique or RBF interpolators [27–29]), and *hp*-adaptive discretizations to further increase numerical accuracy with a lower computational footprint [9].

We expect `life`^{x} to attract a sizable community of users and developers. Any contribution is highly appreciated, from code commits to bug reports and suggestions for improvement.

As we approach the exascale era that will be dominated by high-end supercomputers, numerical simulations are expected to be one of the main computational workloads [60]; against this background, `life`^{x} offers transparency, accessibility, reproducibility, and reusability of *in silico* experiments, within a flexible, high performance software tool.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 740132, iHEART - An Integrated Heart Model for the simulation of the cardiac function; P.I. Prof. A. Quarteroni). We acknowledge the CINECA award `MathBeat` under the ISCRA initiative, for the availability of high performance computing resources and support. The `life`^{x} logo was designed by S. Pozzi. `life`^{x} would not have been possible without a large and loyal team working on software development and review, contributing code and fixes or reporting

bugs and suggestions: N. Barnafi, L. Bennati, M. Bucelli, L. Cicci, S. Di Gregorio, M. Fedele, I. Fumagalli, S. Pagani, R. Piersanti, F. Regazzoni, M. Salvador, S. Stella, E. Zappon, and A. Zingaro, among many others.

Special thanks go to Profs. A. Quarteroni, L. Dede', L. Formaggia, P. Gervasio, A. Manzoni, C. Vergara, and P. Zunino for many stimulating and inspiring discussions and to L. Paglieri for endless patience and support.

`life`^{x} shares the enthusiasm, passion, experience, and dedication to scientific computing brought by several people who contributed to the `LifeV` library [61]. The name itself was inspired by `LiFE` (Library of Finite Elements), conceived by Prof. Fausto Saleri.

References

[1] Groen D, Zasada SJ, Coveney PV. Survey of Multiscale and Multiphysics Applications and Communities. Comput Sci Eng 2014;16(2):34–43. http://dx.doi.org/10.1109/MCSE.2013.47.

[2] Keyes DE, McInnes LC, Woodward C, Gropp W, Myra E, Pernice M, et al. Multiphysics simulations: Challenges and opportunities. The Int J High Perform Comput Appl 2013;27(1):4–83. http://dx.doi.org/10.1177/1094342012468181.

[3] Arndt D, Bangerth W, Blais B, Fehling M, Gassmöller. R, Heister. T, et al. The `deal.II Library, Version 9.3`. J Numer Math 2021;29(3):171–86. http://dx.doi.org/10.1515/jnma-2021-0081, URL https://www.dealii.org/.

[4] Alnaes MS, Blechta J, Hake J, Johansson A, Kehlet B, Logg A, et al. The FEniCS Project Version 1.5. Arch Numer Softw 2015;3. http://dx.doi.org/10.11588/ans.2015.100.20553.

[5] Scroggs MW, Baratta IA, Richardson CN, Wells GN. Basix: a runtime finite element basis evaluation library. J Open Source Softw 2022;7(73):3982. http://dx.doi.org/10.21105/joss.03982.

[6] Anderson R, Andrej J, Barker A, Bramwell J, Camier J-S, Cerveny J, et al. MFEM: A modular finite element methods library. Comput Math Appl 2021;81:42–74. http://dx.doi.org/10.1016/j.camwa.2020.06.009, Development and Application of Open-source Software for Problems with Numerical PDEs.

[7] Permann CJ, Gaston DR, Andrš D, Carlsen RW, Kong F, Lindsay AD, et al. MOOSE: Enabling massively parallel multiphysics simulation. SoftwareX 2020;11:100430. http://dx.doi.org/10.1016/j.softx.2020.100430.

[8] Bungartz H-J, Lindner F, Gatzhammer B, Mehl M, Scheufele K, Shukaev A, et al. preCICE – A fully parallel library for multi-physics surface coupling. Comput & Fluids 2016;141:250–8. http://dx.doi.org/10.1016/j.compfluid.2016.04.003, Advances in Fluid-Structure Interaction.

[9] Arndt D, Bangerth W, Davydov D, Heister T, Heltai L, Kronbichler M, et al. The deal.II finite element library: Design, features, and insights. Comput Math Appl 2021;81:407–22. http://dx.doi.org/10.1016/j.camwa.2020.02.022, Development and Application of Open-source Software for Problems with Numerical PDEs.

[10] Quarteroni A, Valli A. Numerical approximation of partial differential equations. vol. 23, Springer Science & Business Media; 2008, http://dx.doi.org/10.1007/978-3-540-85268-1.

[11] Wilson G, Aruliah DA, Brown CT, Hong NPC, Davis M, Guy RT, et al. Best Practices for Scientific Computing. PLOS Biol 2014;12(1):1–7. http://dx.doi.org/10.1371/journal.pbio.1001745.

[12] Balay S, Abhyankar S, Adams MF, Benson S, Brown J, Brune P, et al. PETSc Web page. 2022, URL https://petsc.org/.

[13] Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, et al. An Overview of the Trilinos Project. ACM Trans Math Software 2005;31(3):397–423. http://dx.doi.org/10.1145/1089014.1089021.

[14] Africa PC. `mk`: environment modules for scientific computing software. 2022, http://dx.doi.org/10.5281/zenodo.6947700, Zenodo.

[15] Africa PC. `life`^{x}`-env`. 2022, http://dx.doi.org/10.5281/zenodo.6947962.

[16] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: Elements of reusable object-oriented software. USA: Addison-Wesley Longman Publishing Co., Inc.; 1995.

[17] Forti D, Dedè L. Semi-implicit BDF time discretization of the Navier–Stokes equations with VMS-LES modeling in a high performance computing framework. Comput & Fluids 2015;117:168–82. http://dx.doi.org/10.1016/j.compfluid.2015.05.011.

[18] Brown J, Brune P. Low-rank quasi-Newton updates for robust Jacobian lagging in Newton methods. In: Proceedings of the 2013 international conference on mathematics and computational methods applied to nuclear science and engineering. 2013, p. 2554–65, URL https://jedbrown.org/files/BrownBrune-LowRankQuasiNewtonRobustJacobianLagging-2013.pdf.

[19] Gill PE, Murray W. Quasi-Newton methods for unconstrained optimization. IMA J Appl Math 1972;9(1):91–108. http://dx.doi.org/10.1007/BF01585529.

[20] Eisenstat SC, Walker HF. Choosing the forcing terms in an inexact Newton method. SIAM J Sci Comput 1996;17(1):16–32. http://dx.doi.org/10.1137/0917003.

[21] Küttler U, Wall WA. Fixed-point fluid–structure interaction solvers with dynamic relaxation. Comput Mech 2008;43(1):61–72. http://dx.doi.org/10.1007/s00466-008-0255-5.

[22] Walker HF, Ni P. Anderson Acceleration for Fixed-Point Iterations. SIAM J Numer Anal 2011;49(4):1715–35. http://dx.doi.org/10.1137/10078356X.

[23] Bucelli M, Dede' L, Quarteroni A, Vergara C. Partitioned and monolithic algorithms for the numerical solution of cardiac fluid-structure interaction. 2021, URL https://www.mate.polimi.it/biblioteca/add/qmox/78-2021.pdf.

[24] Borgdorff J, Mamonski M, Bosak B, Kurowski K, Belgacem MB, Chopard B, et al. Distributed multiscale computing with MUSCLE 2, the Multiscale Coupling Library and Environment. J Comput Sci 2014;5(5):719–31. http://dx.doi.org/10.1016/j.jocs.2014.04.004.

[25] Quarteroni A, Sacco R, Saleri F. Numerical mathematics. vol. 37, Springer Science & Business Media; 2010, http://dx.doi.org/10.1007/b98885.

[26] Quarteroni A, Valli A. Domain decomposition methods for partial differential equations. Oxford University Press; 1999, URL http://infoscience.epfl.ch/record/140704.

[27] Deparis S, Forti D, Gervasio P, Quarteroni A. INTERNODES: an accurate interpolation-based method for coupling the Galerkin solutions of PDEs on subdomains featuring non-conforming interfaces. Comput & Fluids 2016;141:22–41. http://dx.doi.org/10.1016/j.compfluid.2016.03.033, Advances in Fluid-Structure Interaction.

[28] Deparis S, Forti D, Quarteroni A. A Rescaled Localized Radial Basis Function Interpolation on Non-Cartesian and Nonconforming Grids. SIAM J Sci Comput 2014;36(6):A2745–62. http://dx.doi.org/10.1137/130947179.

[29] Salvador M, Dede' L, Quarteroni A. An intergrid transfer operator using radial basis functions with application to cardiac electromechanics. Comput Mech 2020;66(2):491–511. http://dx.doi.org/10.1007/s00466-020-01861-x.

[30] Paliwal N, Ali RL, Salvador M, O'Hara R, Yu R, Daimee UA, et al. Presence of Left Atrial Fibrosis May Contribute to Aberrant Hemodynamics and Increased Risk of Stroke in Atrial Fibrillation Patients. Front Physiol 2021;12. http://dx.doi.org/10.3389/fphys.2021.657452.

[31] Fumagalli I, Fedele M, Vergara C, Dede' L, Ippolito S, Nicolò F, et al. An image-based computational hemodynamics study of the Systolic Anterior Motion of the mitral valve. Comput Biol Med 2020;123:103922. http://dx.doi.org/10.1016/j.compbiomed.2020.103922.

[32] Regazzoni F, Salvador M, Africa P, Fedele M, Dede' L, Quarteroni A. A cardiac electromechanical model coupled with a lumped-parameter model for closed-loop blood circulation. J Comput Phys 2022;457:111083. http://dx.doi.org/10.1016/j.jcp.2022.111083.

[33] Fedele M, Quarteroni A. Polygonal surface processing and mesh generation tools for the numerical simulation of the cardiac function. Int J Numer Methods Biomed Eng 2021;37(4):e3435. http://dx.doi.org/10.1002/cnm.3435.

[34] Liu J, Dede' L, Evans JA, Borden MJ, Hughes TJ. Isogeometric analysis of the advective Cahn–Hilliard equation: Spinodal decomposition under shear flow. J Comput Phys 2013;242:321–50. http://dx.doi.org/10.1016/j.jcp.2013.02.008.

[35] Vergara C, Stella S, Maines M, Africa PC, Catanzariti D, Demattè C, et al. Computational electrophysiology of the coronary sinus branches based on electro-anatomical mapping for the prediction of the latest activated region. Med Biol Eng Comput 2022. http://dx.doi.org/10.1007/s11517-022-02610-3.

[36] Piersanti R, Africa PC, Fedele M, Vergara C, Dede' L, Corno AF, et al. Modeling Cardiac Muscle Fibers in Ventricular and Atrial Electrophysiology Simulations. Comput Methods Appl Mech Engrg 2021;373:113468. http://dx.doi.org/10.1016/j.cma.2020.113468.

[37] Pagani S, Dede' L, Frontera A, Salvador M, Limite LR, Manzoni A, et al. A Computational Study of the Electrophysiological Substrate in Patients Suffering From Atrial Fibrillation. Front Physiol 2021;12. http://dx.doi.org/10.3389/fphys.2021.673612.

[38] Stella S, Vergara C, Maines M, Catanzariti D, Africa PC, Dematté C, et al. Integration of Activation Maps of Epicardial Veins in Computational Cardiac Electrophysiology. Comput Biol Med 2020;127:104047. http://dx.doi.org/10.1016/j.compbiomed.2020.104047.

[39] Piersanti R, Regazzoni F, Salvador M, Corno A, Vergara C, Quarteroni A, et al. 3D–0D closed-loop model for the simulation of cardiac biventricular electromechanics. Comput Methods Appl Mech Engrg 2022;391:114607. http://dx.doi.org/10.1016/j.cma.2022.114607.

[40] Salvador M, Regazzoni F, Pagani S, Dede L, Trayanova N, Quarteroni A. The role of mechano-electric feedbacks and hemodynamic coupling in scar-related ventricular tachycardia. Comput Biol Med 2022;142:105203. http://dx.doi.org/10.1016/j.compbiomed.2021.105203.

[41] Salvador M, Fedele M, Africa PC, Sung E, Dede' L, Prakosa A, et al. Electromechanical modeling of human ventricles with ischemic cardiomyopathy: numerical simulations in sinus rhythm and under arrhythmia. Comput Biol Med 2021;136:104674. http://dx.doi.org/10.1016/j.compbiomed.2021.104674.

[42] Dedè L, Quarteroni A, Regazzoni F. Mathematical and numerical models for the cardiac electromechanical function. Atti Accad Naz Lincei Cl Sci Fis Mat Natur Rend Lincei 2021;32(2):233–72. http://dx.doi.org/10.4171/rlm/935.
[43] Quarteroni A, Dedè L, Regazzoni F. Modeling the cardiac electromechanical function: A mathematical journey. Bull Amer Math Soc 2022;59(3):371–403. http://dx.doi.org/10.1090/bull/1738.
[44] Zingaro A, Bucelli M, Fumagalli I, Dede' L, Quarteroni A. Modeling isovolumetric phases in cardiac flows by an Augmented Resistive Immersed Implicit Surface Method. 2022, http://dx.doi.org/10.48550/ARXIV.2208.09435.
[45] Zingaro A, Fumagalli I, Fedele M, Africa PC, Dede' L, Quarteroni A, Corno AF. A geometric multiscale model for the numerical simulation of blood flow in the human left heart. Discrete Contin Dyn Syst - S 2022;15(8):2391–427. http://dx.doi.org/10.3934/dcdss.2022052.
[46] Fumagalli I, Vitullo P, Vergara C, Fedele M, Corno A, Ippolito S, et al. Image-based computational hemodynamics analysis of systolic obstruction in hypertrophic cardiomyopathy. Front Physiol 2022;2437. http://dx.doi.org/10.3389/fphys.2021.787082.
[47] Barnafi Wittwer NA, Gregorio SD, Dede' L, Zunino P, Vergara C, Quarteroni A. A Multiscale Poromechanics Model Integrating Myocardial Perfusion and the Epicardial Coronary Vessels. SIAM J Appl Math 2022;82(4):1167–93. http://dx.doi.org/10.1137/21M1424482.
[48] Gregorio SD, Fedele M, Pontone G, Corno AF, Zunino P, Vergara C, et al. A computational model applied to myocardial perfusion in the human heart: From large coronaries to microvasculature. J Comput Phys 2021;424:109836. http://dx.doi.org/10.1016/j.jcp.2020.109836.
[49] Dedè L, Regazzoni F, Vergara C, Zunino P, Guglielmo M, Scrofani R, et al. Modeling the cardiac response to hemodynamic changes associated with COVID-19: a computational study. Math Biosci Eng 2021;18(4):3364–83. http://dx.doi.org/10.3934/mbe.2021168.
[50] Fedele M, Piersanti R, Regazzoni F, Salvador M, Africa PC, Bucelli M, et al. A comprehensive and biophysically detailed computational model of the whole human heart electromechanics. 2022, http://dx.doi.org/10.48550/ARXIV.2207.12460.
[51] Bucelli M, Zingaro A, Africa PC, Fumagalli I, Dede' L, Quarteroni A. A mathematical model that integrates cardiac electrophysiology, mechanics and fluid dynamics: application to the human left heart. 2022, http://dx.doi.org/10.48550/ARXIV.2208.05551.
[52] Bucelli M, Gabriel MG, Gigante G, Quarteroni A, Vergara C. A stable loosely-coupled scheme for cardiac electro-fluid-structure interaction. 2022, http://dx.doi.org/10.48550/ARXIV.2210.00917.
[53] Africa PC, Salvador M, Gervasio P, Dede' L, Quarteroni A. A matrix-free high-order solver for the numerical solution of cardiac electrophysiology. 2022, http://dx.doi.org/10.48550/ARXIV.2205.05136.
[54] Cicci L, Fresca S, Pagani S, Manzoni A, Quarteroni A. Projection-based reduced order models for parameterized nonlinear time-dependent problems arising in cardiac mechanics. Math Eng 2023;5(2):1–38. http://dx.doi.org/10.3934/mine.2023026.
[55] Regazzoni F, Salvador M, Dedè L, Quarteroni A. A machine learning method for real-time numerical simulations of cardiac electromechanics. Comput Methods Appl Mech Engrg 2022;393:114825. http://dx.doi.org/10.1016/j.cma.2022.114825.
[56] Regazzoni F, Quarteroni A. Accelerating the convergence to a limit cycle in 3D cardiac electromechanical simulations through a data-driven 0D emulator. Comput Biol Med 2021;135:104641. http://dx.doi.org/10.1016/j.compbiomed.2021.104641.
[57] Inc. ZMG. Zygote solid 3D heart generation II. 2014, Development Report.
[58] Africa PC, Piersanti R, Fedele M, Dede' L, Quarteroni A. An open tool based on `life`[x] for myofibers generation in cardiac computational models (Software). 2022, http://dx.doi.org/10.5281/zenodo.5810268, Zenodo.
[59] Africa PC, Piersanti R, Fedele M, Dede' L, Quarteroni A. An open tool based on `life`[x] for myofibers generation in cardiac computational models. 2022, http://dx.doi.org/10.48550/ARXIV.2201.03303.
[60] Alowayyed S, Groen D, Coveney PV, Hoekstra AG. Multiscale computing in the exascale era. J Comput Sci 2017;22:15–25. http://dx.doi.org/10.1016/j.jocs.2017.07.004.
[61] Bertagna L, Deparis S, Formaggia L, Forti D, Veneziani A. The LifeV library: engineering mathematics beyond the proof of concept. 2017, http://dx.doi.org/10.48550/ARXIV.1710.06596.