CARDIAX: A JAX-based platform for Rapid Cardiac Functional Simulations

Benjamin J. Thomas[a], Christian Goodbrake[a], Kenneth Meyer[a], and Michael S. Sacks[a,b,c,1]

[a] *James T. Willerson Center for Cardiovascular Modeling and Simulation*

[a]*The Oden Institute for Computational Engineering and Sciences*

[b]*Department of Biomedical Engineering*

[c]*Department of Mechanical Engineering*

The University of Texas at Austin, Austin, TX, USA

Keywords:

[1]Corresponding Author: msacks@oden.utexas.edu

ABSTRACT

Computational pipelines of increasing sophistication are actively being developed for both basic studies and patient-specific cardiac simulations for direct clinical utilization. A major barrier continues to be simulation execution times, limiting levels of scale integration and performing such simulations in clinically relevant time frames. While machine-learning based surrogate models are an active research area, they remain early in development. As an alternative approach we have developed CARDIAX, a JAX-based traditional finite element platform that leverages GPU technologies for accelerated cardiac simulations. A high-fidelity cardiac model was developed in the CARDIAX framework to simulate a complete pressure-volume loop in a normal and infarcted heart. The cardiac model was based on an extant comprehensive dataset acquired from a single ovine heart, which accounted for realistic biventricular geometry and fiber orientations, myocardial mechanical behavior including physiologically realistic compressibility behavior, and realistic kinematic constraints. We were able to run simulations considerably faster in the JAX framework using GPUs compared to similar CPU based software platforms making at least a 10 fold improvement in runtime. In a cardiac disease model, CARDIAX demonstrated a mean increase in active stress of $\sim$30 kPa in the presence of an apical infarction compared to the normal case. To better conduct objective performance benchmarking, we developed and utilized a standard benchmark using an cuboidal test problem with a hyperelastic isotropic material model applied to demonstrate the performance characteristics of each software platform evaluated. CARDIAX exhibited well over a ten fold speed gain across a range of mesh sizes. Regarding scalability, for larger problem sizes with 100,000 degrees of freedom CARDIAX demonstrated more than 35 times faster execution time. We conclude that use of GPU-based software technologies can be effective platforms for high speed cardiac mechanics simulations.

1. INTRODUCTION

Myocardial infarction (MI) is a leading cause of death affecting nearly 1 million people annually, resulting in the death of nearly 30% of those affected. High accuracy computational simulations of MI can play a pivotal role in improving our understanding of the cascade of events that lead to progressive heart failure, as well as guide future therapies. A major direction in healthcare is digital twin technology, a modeling approach based on continuously updated patient-specific data to better predict a future state of the patient. To successfully create a digital twin, the computational model must be able to update quickly and accurately, and be easily adaptable at the scale needed for clinical use [1]. One example application would be a digital twin for a MI patient, taking into account the location and severity of infarction; however, simulation times are one of the major barriers preventing this capability.

Depending on the specific goal of the digital twin, different levels of fidelity and complexity would be required in the model. Many groups have developed very high quality models for solving cardiac mechanics and multiphysics problems using a variety of approaches where the primary goal was validation of the model. Some groups focus on the multiphysics capabilities, combining the electrophysiology with the mechanics [2]. These models use ion channel dynamics to track levels of calcium which trigger contractile forces which can be further coupled through stretch dependent ion channels [3]. In general, it is preferred to simulate the full effects of the diseased state, but this remains computationally prohibitively expensive for clinical applications. This is especially the case for solving inverse problems to obtain patient specific characteristics. One way to remedy the computational cost issue is through simplification of the equations and related reduced order approaches, but these often results in noticeable loss of accuracy.

Another long-standing issue is the development of high performance computing (HPC) codes that that provide substantial performance but limit flexibility. These popular open source packages are typically built on C++ code with various levels of control for the user (e.g. Lifex [4]). A popular alternative is FEniCS (now FEniCSx) [5, 6] which allows the use of Python through the Unified Form Language, making it more user friendly and facilitates code recycling. However, it is often necessary to perform low-level modifications for require modifications of the C++ source code, losing much of its intrinsic advantages. Moreover, these packages rely on parallel processing on CPU cores, yet GPUs have been shown to outperform CPUs in highly parallelized tasks such as matrix vector multiplication.

In recent years, GPU-based technology has made major advances thanks to the rapid rise of machine learning (ML) and artificial intelligence (AI) applications. Thus, there remains significant advantages to using current GPU based ML/AI software methods (e.g. just-in-time (JIT) compilation, automatic differentiation, vectorized parallelization) in a traditional forward model

framework. In particular, GPUs provide substantial speed gains over CPUs in solving the large linear systems encountered in the finite element method. The JAX library [7] offers a convenient, fully customizable approach to exploit. In particular, the recognition of the ability to use JAX for high speed finite element approaches has been recently demonstrated with JAX-FEM [8]. This open-source software demonstrated the high speed and functionality of traditional finite element approaches in the JAX environment.

In the present work, we extended JAX-FEM to develop CARDIAX, a cardiac mechanics specific framework in the JAX environment. The code was refactored to be more modular, clearly separating the components based on their functionality. There are two main advantages of this Python/JAX approach. The first is with regards to development time because the code is entirely in Python, so the user can see exactly where errors occur and modify the code as they wish. The second is the efficient integration with GPUs, taking full advantage of their high parallelization. We evaluated the performance on a biventricular model by solving a pressure-volume (PV) loop over the cardiac cycle with a healthy and apically infarcted heart. Comparisons to similar work in the literature for the biventricular model are difficult to make directly because of the variations in geometry, formulation, and available data. To have a more concrete performance metric, we developed a unit cube model using a neo-Hookean hyperelastic material model in compression to serve as common benchmarked. This benchmark provided a controlled test across the different software platforms wherein all model components (e.g. mesh) and tolerances were the identical, allowing focus on the solution time alone.

2. METHODS

2.1. Overview

Herein we describe the development and application of CARDIAX, a JAX-based software platform for rapid finite element based cardiac mechanics simulations. A detailed biventricular model was used, based on a comprehensive, high-fidelity ovine heart data source [9]. We then compared the performance of CARDIAX with our previous ABAQUS-based biventricular model, based on the same ovine heart dataset [10]. We further compared our results to related cardiac simulations performed in FEniCS [11] from the literature. To better compare CARDIAX's performance, we utilized a single common benchmark problem that consisted of a single cube composed of an isotropic hyperelastic material. CARDIAX, JAX-FEM, and FEniCSx were then applied to this problem under the same solvers and settings.

2.2. Cardiac model development

2.2.1. Ovine heart data and model construction

An extant ovine heart database was used to develop the CARDIAX model, as extensively described in Liu et al. [10] and Soares et al. [9]. This extensive dataset from a single ovine heart collected using in vivo, ex situ, and ex vivo sources. The dataset included biventricular geometry, pressure-volume data, epicardial monophasic action potentials, and diffusion tensor magnetic image resonance data (DTMRI). The employment of a complete dataset from a single heart avoided registration issues associated with multiple datasets from different animals, as well as the use of prescribed mathematical models for quantities such as fiber architecture. To align the DTMRI measurements to the finite element mesh precisely, the data was interpolated onto the cells of the mesh. Simulations were conducted that incorporated myocardial wall compressibility in both the normal and infarcted heart. The effect of myocardial infarction on the levels of active contraction was examined as done in the Liu et al. [10]. Other kinematic variables such as volume change were also used to validate the simulations.

2.3. Formulation and Implementation

2.3.1. Boundary Conditions

As an improvement to our previous ABAQUS-based model, we made the boundary conditions more physiologically realistic by applying a spring-like boundary condition to the base, Γ_{Base}, to represent the attachment of the ventricles to the atria (Figure 1). A similar boundary condition was then applied to the epicardium, Γ_{Epi}, with a less stiff spring to model the heart surrounded by the pericardium. These boundary conditions are similar to [12, 13], except our pressure-volume simulations are quasi-static, so we neglect the time component and use an isotropic stiffness.

As in the previous model [10], the full pressure field was applied to the left ventricle endocardial surface with a corresponding pressure of 20% applied to the right ventricle endocardial surface. The pressures $p_{LV}(t)$ and $p_{RV}(t)$ are time dependent, changing over the cardiac cycle with $p_{RV}(t) = 0.2p_{LV}(t)$ (Figure 1). The pressures remain constant over the surface, so for a fixed time point, the pressure scales the boundary integral.

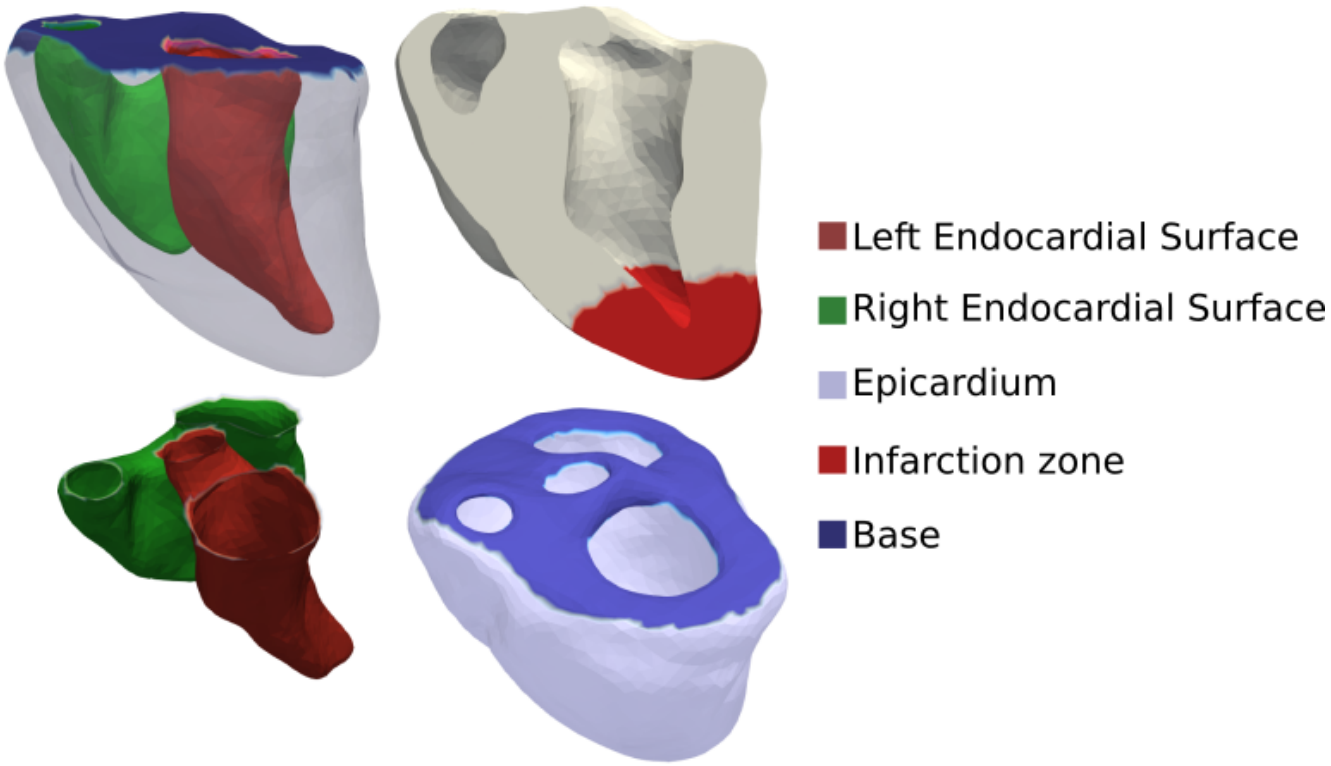

Figure 1: The mesh on the top left shows the boundaries where surface integrals are computed in solving the PV loop. The mesh to the right show the region of the infarction. This represents the tissue that will not contract in the infarct simulations. The bottom meshes show the biventricular model from a top view.

2.3.2. *Model Mathematical Formulation*

We define the governing PDE for passive cardiac mechanics. This is formally the conservation of linear momentum in the referential configuration

$$\nabla \cdot \mathbf{S} + \mathbf{b}_0 = 0 \quad \text{in } \Omega_0 \tag{1}$$

where $\mathbf{S}$ is the second Piola-Kirchhoff stress tensor and $\mathbf{b}_0$ is the body force per unit volume in the reference configuration Ω_0. Due to their small effects we ignore body forces by setting $\mathbf{b}_0 = 0$. As usual, our goal is to solve for the displacements $\mathbf{u}$. Per conventional finite element approaches, we use a weak form of the PDE obtained by multiplying by a test function $\mathbf{v}$ (virtual displacements) and integrating by parts. The weak form is given by

$$\int_{\Omega_0} \mathbf{S} : \left(\mathbf{F}^T \nabla \mathbf{v}\right) dx + p_{LV} \int_{\Gamma_{LV}} J\mathbf{F}^{-T}\mathbf{n} \cdot \mathbf{v} ds + p_{RV} \int_{\Gamma_{RV}} J\mathbf{F}^{-T}\mathbf{n} \cdot \mathbf{v} ds \\ - \int_{\Gamma_{Epi}} k_{Epi}\mathbf{u} \cdot \mathbf{v} ds - \int_{\Gamma_{Base}} k_{Base}\mathbf{u} \cdot \mathbf{v} ds = 0 \tag{2}$$

Here, $\mathbf{F} = \nabla\mathbf{u} + \mathbf{I}$ is the deformation gradient tensor which is the sum of the gradient of displacements $\nabla\mathbf{u}$, plus the identity tensor $\mathbf{I}$, $J = \det(\mathbf{F})$ is the Jacobian determinant, $\mathbf{n}$ is the vector normal to the surface, and k_{epi} and k_{Base} are the stiffness coefficients. The first term represents the contribution of the internal stress. The second and third term are the pressure boundary conditions on the left and right endocardial surfaces. The last two terms are spring-like boundary terms to represent the connection of the epicardium to the pericardium and the attachment of the base to the atrium respectively.

2.3.3. Passive and Active Constitutive Model

We take similar approaches to that used in Liu et al. [10] by assuming the total myocardial stress to be the sum of the passive and active stresses using a hyperelastic framework. For the passive component, the hyperelastic strain energy is split into deviatoric and volumetric components.

$$\Psi = \Psi_{dev}(\tilde{\mathbf{E}}) + \Psi_{vol}(J) \tag{3}$$

where $\mathbf{E} = \frac{1}{2}(\mathbf{F}^T\mathbf{F} - \mathbf{I})$ is the Green-Lagrange strain tensor

$\tilde{\mathbf{E}} = J^{-2/3}\mathbf{E}$ is the deviatoric component of $\mathbf{E}$, and $J = \det(\mathbf{F})$. This explicit compressible formulation allows control of how much the volumetric component contributes to the strain energy [10, 14]. We again chose $\Psi_{dev}(\tilde{\mathbf{E}})$ to be a transversely-isotropic Fung-based model [15]

$$\Psi_{dev}(\tilde{\mathbf{E}}) = \frac{c}{2}\left[\exp\left(\alpha Q(\tilde{\mathbf{E}})\right) - 1\right] \tag{4}$$

where

$$Q(\tilde{\mathbf{E}}) = A_1\tilde{E}_{11}^2 + A_2(\tilde{E}_{22}^2 + \tilde{E}_{33}^2 + 2\tilde{E}_{23}^2) + A_3(\tilde{E}_{12}^2 + \tilde{E}_{13}^2) \\ \tilde{E}_{ij} = \mathbf{v}_i \cdot \tilde{\mathbf{E}}\mathbf{v}_j \quad \mathbf{v}_i \in \{\mathbf{f}_0, \mathbf{s}_0, \mathbf{n}_0\} \tag{5}$$

the values $\mathbf{f}_0$, $\mathbf{s}_0$, and $\mathbf{n}_0$ are the local fiber, sheet, and normal myofiber directions in the undeformed configuration. The myofiber directions are obtained from the eigenvectors e_I, e_{II}, and e_{III} which correspond to the fiber, sheet, and normal directions [10]. The volumetric portion of the strain energy was modeled using

$$\Psi_{vol}(J) = \frac{K}{2}\left(\frac{J^2 - 1}{2} - \ln(J)\right) \tag{6}$$

with K controlling the volume preserving penalty level, and is physically the bulk modulus in the small deformation limit.

From the above we first determine the passive stress $\mathbf{S}_{pass}$ using

$$\mathbf{S}_{pass} = \frac{\partial \Psi}{\partial \mathbf{E}}. \tag{7}$$

The active component of the stress tensor is modeled by the contraction of the myofibers and is given by [10, 16]

$$\mathbf{S}_{act} = T_{Ca}\ g(x,t) \frac{1+\beta(\lambda_f - 1)}{\lambda_f^2} (\mathbf{f}_0 \otimes \mathbf{f}_0) \tag{8}$$

$$\lambda_f = \|\mathbf{F}\mathbf{f}_0\|_2 \tag{9}$$

where T_{Ca} is the magnitude of the active stress, $g(x,t)$ is an element based function used to turn off active contractions in selected regions, and λ_f is the local fiber stretch. The total stress then becomes

$$\mathbf{S} = \mathbf{S}_{pass} + \mathbf{S}_{act} \tag{10}$$

2.3.4. Myocardial Compressiblity

It is known that the vasculature takes up about 15-20 % [17, 18] of the volume of ventricular myocardium. During systole the contractile forces of the myocardium are thought to compress the vasculature, pushing blood of this space in both the arterial and venous components. We have previously demonstrated the importance of myocardial compressiblity in realistic cardiac mechanics simulations [10, 19]. Specifically, we noted that a compressible myocardial model provided a more accurate representation of ventricular kinematics and more realistic computed active contraction levels. The present approach adopts this method where the time-changing compressibility of the myocardial tissue was based on the level of active contraction by changing the value of K in Equation 6.

Taking our previous approach [10, 19], we model bulk myocardial wall compressiblity as a time-varying modulus $K = K(t)$ that is dependent upon $T_{Ca}(t)$ using

$$K(t) = K_{inc} - \gamma T_{Ca}(t) \tag{11}$$

where γ is tuned so the $K(t)$ ranges from 10.0 to 0.64 GPa at end systole [10]. We have shown this basic approach accurately captures the observed wall compressibility that occurs in end systole. Due to the compressibility that now occurs in the model, volume locking was not an issue since we are mostly away from the nearly incompressible regime over the course of the simulation.

2.3.5. The Pressure-Volume Inverse Problem

We next define the inverse problem to solve for $T_{Ca}(t)$. As in previous studies [10, 11], we aim to solve for the amount of additive active stress controlled by T_{Ca} required to obtain a specified

volume V for a given LV pressure p_{LV}. Therefore, we are given data points as PV pairs $\{p_{LV}, V\}$ and want to search for the correct value of T_{Ca}. While the residual of the PDE (Equation 2) can be written as $\mathcal{R}(\mathbf{u}, T_{Ca}, p_{LV}) = 0$, we are concerned with solving for displacements given the additive stress and pressure, $\mathcal{S}(T_{Ca}, p_{LV}) = \mathbf{u}$. After obtaining the solution, the observable $\tilde{V} = \mathcal{Q}(\mathbf{u})$, is computed from the displacements.

To compute the volume in a quick and accurate manner, we have implemented the shoelace method [20]. The shoelace method requires a closed, orientable boundary, which for us is the surface of the left ventricle. The surface of the left ventricle is closed by capping the mitral and aortic valves. Once we have the closed triangular surface, we add an arbitrary point, usually chosen to be the origin, to make all of the triangles into a collection of tetrahedra, $\mathcal{T}$. The volume of the enclosed surface is then computed by

$$\tilde{V} = \frac{1}{6} \sum_{\tau \in \mathcal{T}} \text{sign}(\tau) |(v_1 \times v_2) \cdot v_3| \tag{12}$$

The error between the computed and measured volume is written as

$$\mathcal{L}(T_{Ca}, p_{LV}, V) = \mathcal{Q}(\mathcal{S}(T_{Ca}, p_{LV})) - V \tag{13}$$

with the error function being parameterized by the data. At a specific point in the cardiac cycle, the values are fixed, and only T_{Ca} can vary

$$\mathcal{L}(T_{Ca}; p_{LV}^i, V^i) = \mathcal{Q}(\mathcal{S}(T_{Ca}; p_{LV}^i)) - V^i \tag{14}$$

where we are optimizing a function from $\mathbb{R} \to \mathbb{R}$. The loss is written this way to take advantage of Newton's Method for roots. Updates are computed by taking the derivative of the loss

$$\frac{d\mathcal{L}}{dT_{Ca}}(T_{Ca}) = \left(\frac{d\mathcal{Q}}{d\mathbf{u}}(\mathcal{S}(T_{Ca})) \right)^T \cdot \frac{d\mathbf{u}}{dT_{Ca}}(T_{Ca}) \tag{15}$$

Computing $\frac{d\mathcal{Q}}{d\mathbf{u}}$ is straightforward; however, the term $\frac{d\mathbf{u}}{dT_{Ca}}(T_{Ca})$ must be computed through the adjoint method because of the solution operator, $\mathbf{u} = \mathcal{S}(T_{Ca})$. Thus, we differentiate the residual

$$\frac{\partial \mathcal{R}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial T_{Ca}} = -\frac{\partial \mathcal{R}}{\partial T_{Ca}} \tag{16}$$

and plug back into Equation 15

$$\frac{d\mathcal{L}}{dT_{Ca}}(T_{Ca}) = -\left(\frac{d\mathcal{Q}}{d\mathbf{u}}(\mathcal{S}(T_{Ca})) \right)^T \cdot \left(\frac{\partial \mathcal{R}}{\partial \mathbf{u}} \right)^{-1} \frac{\partial \mathcal{R}}{\partial T_{Ca}} \tag{17}$$

and solve the adjoint equation for $\mathbf{p}$

$$\left(\frac{\partial \mathcal{R}}{\partial \mathbf{u}} \right)^T \mathbf{p} = -\frac{d\mathcal{Q}}{d\mathbf{u}}(\mathcal{S}(T_{Ca})) \tag{18}$$

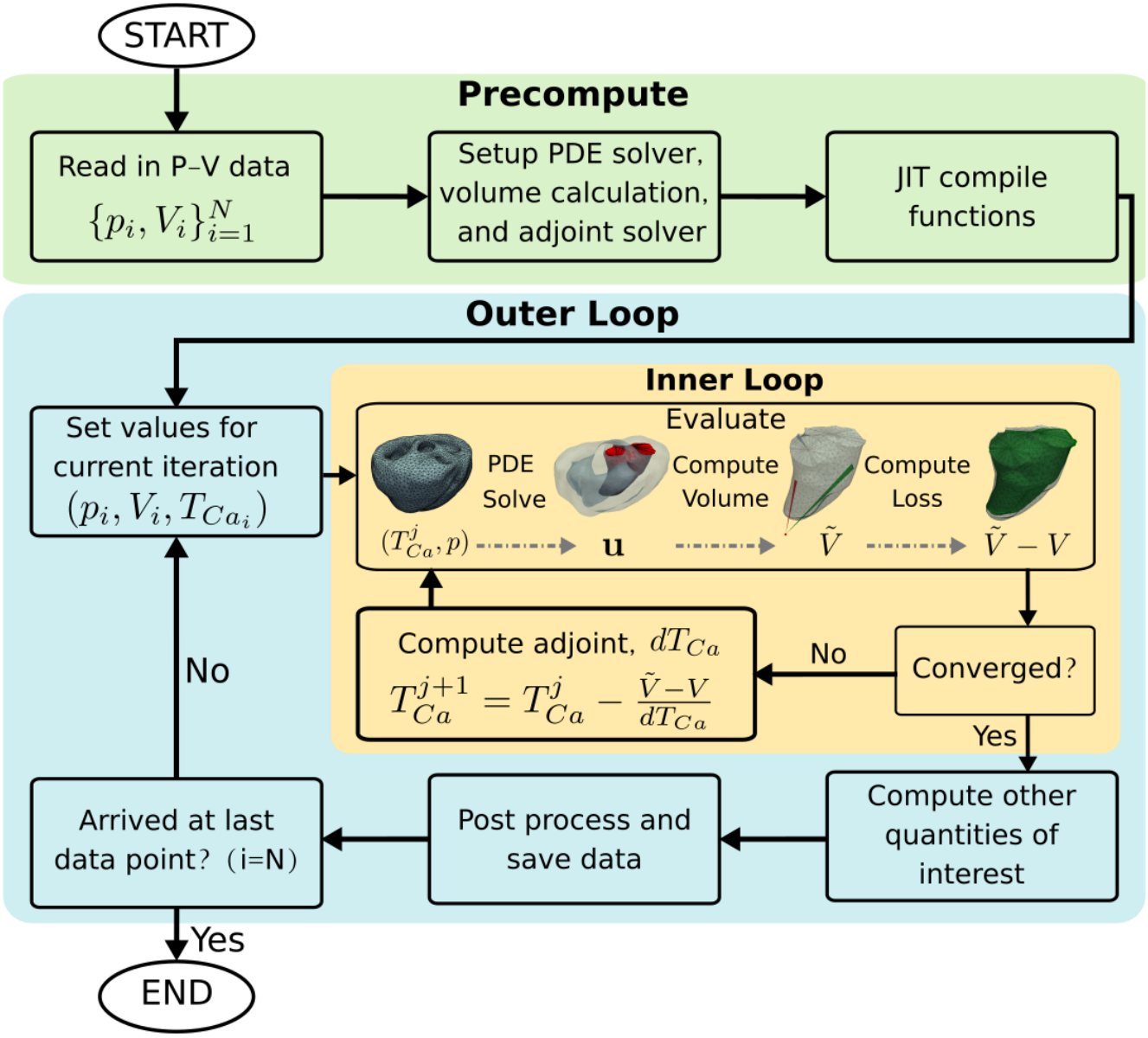

Figure 2: To solve the inverse problem, we follow the control flow described above. First, functions are created and compiled. Then the outer loop iterates over the data points, solving for values of T_{Ca}. At each iteration of the outer loop, we also iterate over the inner loop using the "Evaluate" function to compute derivatives through the PDE.

The gradient is then represented as

$$\frac{d\mathcal{L}}{dT_{Ca}}(T_{Ca}) = \left(\frac{\partial \mathcal{R}}{\partial T_{Ca}}\right)^T \mathbf{p} \tag{19}$$

Once derivatives are computed, we then update T_{Ca}

$$T^i_{Ca} \leftarrow T^i_{Ca} - \mathcal{L}(T^i_{Ca})/\mathcal{L}'(T^i_{Ca}) \tag{20}$$

and this is repeated until convergence (Figure 2).

2.4. CARDIAX: Development and Implementation

2.4.1. Overview

JAX is a very natural library to use when creating a finite element software platform, as demonstrated in JAX-FEM [8]. In their work, Xue et al. showed how GPUs can surpass CPUs in the larger DoF regime thanks to a number of advantages. The main solver used is Newton's Method with consistent tangents, made possible by automatic differentiation. The assembly of the matrix system was computed with vectorized functions. The solvers take advantage of GPU architecture to perform faster matrix-vector products. Their results were then validated against FEniCSx to ensure the code works as intended. Along with the standard finite element functionality, the adjoint method was also implemented to perform inverse problems more easily. This allowed JAX-FEM to tackle a wide range of problems encountered in computational mechanics.

We wanted to expand the capabilities of JAX-FEM to cover the problems faced in cardiac mechanics. CARDIAX has enhanced the code through reorganizing (refactoring), more substantially utilizing JIT compilation, and leveraging the Equinox ecosystem. To emphasize, JIT compilation is the primary way to reduce computational time in these simulations. The more code that can be JIT compiled, which is typically memory bound, the faster the code can execute. The implemented changes have allowed for more efficient development and less computational time to solve these types of mechanics problems.

2.4.2. Modularization

The code was modularized to follow a style similar to FEniCS/FEniCSx; whereas the previous code was more functional in style. To start, we define a `FiniteElement` class that contains mesh information, defines the quadrature rule, and tracks local DoFs. The `Problem` class defined the PDE over the function space where the `FiniteElement` class is given as an input. We use `Problem` to collect the local DoFs for the global system, compute the residual, and build the linearized stiffness matrix. Lastly, the `Solver` class allows the user to tune its parameters dynamically and call the `solve` method when appropriate. All components fit seamlessly together, minimizing information and allowing the user full control over the simulation as desired.

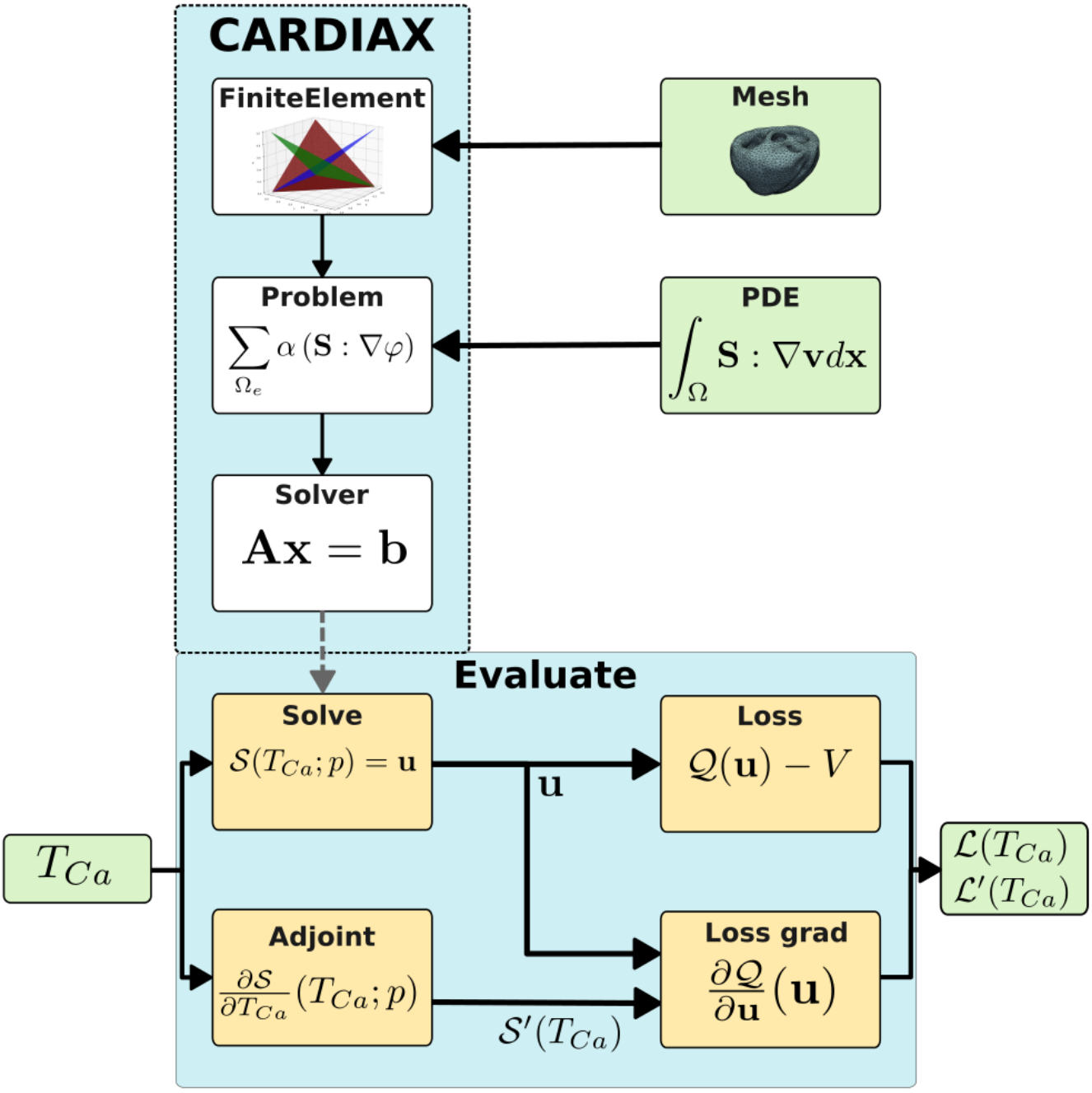

Figure 3: The above describes how the CARDIAX package is used for solving inverse problems. The mesh is given to `FiniteElement` to create the desired function space. Together with the function space and user defined PDE, the `Problem` class is created for the assembly of the linear system. Then `Solver` is made to solve the linear system. Given the defined loss, which for us is the volume mismatch, we can create the "Evaluate" function to compute values and derivatives of the PDE-constrained loss function.

Another benefit that comes with modularization is the flexibility it provides to solve various problems. A common roadblock that comes up in modeling nearly incompressible materials is the issue of volume locking. While volume locking was not an issue in our case due to the myocardial compressibility keeping us away from the stiff regime, we are able to make the problem more stable through reduced integration. This is done by defining two separate finite element fields and kernels for computing the strain energy, one for the deviatoric component and one for the volumetric. To let `Problem` know these are the same displacement variables, we bind the two fields, which tells the solver to add these contributions together. By doing so, we can define a field with a reduced integration order providing the stability needed to prevent volume locking.

2.4.3. The Equinox Ecosystem

The Equinox ecosystem [21] gives a very clean and efficient way to extend the capabilities of the JAX library. The main reason for needing to extend the functionality of JAX is that the core of JAX's functionality is concerned with fundamentals and flexibility. The Equinox ecosystem extends this to specific applications relevant for scientific computing. More specifically, Lineax [22] is one package in the ecosystem designed to solve linear systems in a JAX-compatible way by allowing the user to define custom linear operators.

For finite element systems, the matrix system is fixed once the PDE is defined, so after the `Problem` class is initialized, we define a custom object, `SparseMatrixLinearOperator`, to represent the global stiffness matrix. The sparsity pattern is fixed, so we only have to fill in the data of the matrix. By using the `sparsify` functionality in JAX, we can now use coordinate list (COO) sparse representation of the matrix. Another major improvement offered by Lineax is the linear system solvers. Lineax allows for more solvers than standard JAX, is more numerically stable, and provides information about when and how the solver terminates. The information provided from the solver can be invaluable when debugging the system because it gives insights to when the solve fails, allowing the user to make the appropriate corrections. The linear system solve can now be JIT compiled with the `SparseMatrixLinearOperator` and residual vector as inputs. Since we are focused on solving nonlinear problems in cardiac mechanics, the linear solve is called many times, so reducing its execution time is very impactful.

2.4.4. Inverse Problem Solution

To solve the inverse problem, as mentioned in Section 2.3.5, we must compute the gradients through the adjoint method. The adjoint method is implemented by wrapping a custom derivative rule around the `solve` method. The new modular structure of CARDIAX allows the user to adjust the solver class after the derivative rule has been created, which is especially important for sequential inverse problems, where one would like to use the previous solution as the initial

guess for the next problem. The adjoint solve also uses the same JIT compiled function as the standard solve because we pass the transpose and a different target vector. The advantage of this implementation is the ease of use on the user's side. The user only defines a function that computes the error for a quantity of interest from the solution vector, in our case the volume. Calling the gradient on this function will give the `Evaluate` function (Figure 3). Passing the desired parameters as input, the loss and gradient will be calculated and returned to the user, which makes custom optimization easy to implement as done for the PV loop. The solution can then be used as the initial guess for the next iteration to minimize solve time.

Data: $\{p_{LV}^i, V^i\}_{i=1}^N$
Result: $\{\mathbf{u}^i, T_{Ca}^i\}_{i=1}^N$
Initialize `FiniteElement`, `Problem`, and `Solver` classes
Define solve for $\mathcal{R}(\mathbf{u}, T_{Ca}, p_{LV})$
Define error function, $\mathcal{L}(T_{Ca}, p_{LV}, V)$
Set $T_{Ca}^0 = 0$, $\mathbf{u}^0 = \mathbf{0}$
for $i \in \{1, 2, ..., N\}$ **do**
  Set $p_{LV} \leftarrow p^i$, $V \leftarrow V^i$
  Initial guess $\mathbf{u}_0^i \leftarrow \mathbf{u}^{i-1}$, $T_{Ca_0}^i \leftarrow \mathrm{T}_{Ca}^{i-1}$
  for $j \in \{1, 2, ..., \mathit{iters}\}$ **do**
    $\mathbf{u}_j^i \leftarrow \mathcal{R}(\mathbf{u}_{j-1}^i, T_{Ca_{j-1}}^i; p_{LV})$ when solved
    $\ell \leftarrow \mathcal{L}(T_{Ca_{j-1}}^i; p_{LV}, V)$
    if $\ell < \epsilon$ **then**
      break
    else
      $T_{Ca_j}^i \leftarrow T_{Ca_{j-1}}^i - \ell / \mathcal{L}'(T_{Ca_{j-1}}^i)$
    end
  end
end

Algorithm 1: Pseudocode for solving an inverse problem

2.4.5. Benchmarking using a common test case

While comparing previous computational outcomes in different studies is useful, such comparisons can be confounded by subtle variations in geometry, material models, objective functions, and many other modeling factors. To provide for a more direct comparison between each computational approach, it is best to evaluate them against an identical problem on the same platform. Towards this end, we developed the following benchmark problem. The domain prescribed is a unit cube where displacements are fixed to zero on the bottom face. The top face

has a traction applied downward, compressing the cube. The material model was chosen to be neo-Hookean model. For the neo-Hookean model, the strain energy is defined by

$$\Psi = \frac{\mu}{2}(J^{-2/3}I_{\mathbf{C}} - 3) + \frac{\kappa}{2}(J-1)^2 \tag{21}$$

$$\mu = \frac{E}{2(1+\nu)} \tag{22}$$

$$\kappa = \frac{E}{3(1-2\nu)} \tag{23}$$

where $I_C = \text{trace}(\mathbf{F}^T\mathbf{F})$ is the first invariant of the right Cauchy-Green strain tensor. Note that in the small strain limit, E approaches the Young's modulus, and ν approaches Poisson's ratio, and the parameters μ and k become related to the Lame parameters. The parameter values used to run the benchmark can be found in Appendix 7.2.

To also test the scalability of the code, we ran simulations on the neo-Hookean cube, but only scale the number of degrees of freedom by refining the mesh. The mesh sizes range from a cube with 5 elements per side to 50 elements, which correspond to a range of 10^2 to 10^5 DoFs. All implementations solved the problem with Newton's Method using a consistent tangent. To solve the linear system for the increment of displacements, all codes used bicgstab with jacobi preconditioning with a tolerance of $1\text{e}{-10}$. The solve was completed when the residual reached below the tolerance of $1\text{e}{-6}$. The three simulations were cross validated to make sure they arrived to the same solution. We also note that since FEniCSx first compiles the unified form language as input, we allowed the JAX based codes (JAX-FEM and CARDIAX) to be solved once to JIT compile then solved again to obtain an equivalent performance metric, and thus only report only the live computational time. The JIT compile time is considered offline because it can be stored on the device if running multiple times. Speed improvements for FEniCSx comes from utilizing MPI to distribute the problem across different cores of the CPU while the JAX-based codes perform better based on how much of the code is compiled on the GPU.

3. RESULTS

3.1. Basic Performance

For the healthy heart model we began at the initial pressure of 16 mmHg with no active stress. Along with the initial pressure solve, the code is compiled onto the GPU. To complete the initial solve and the process of JIT compilation took less than five minutes. The simulation was then performed at 100 data points over the cardiac cycle using different pressure-volume pairs. All of the active stress values were solved in 53 minutes from start to finish. For the healthy heart the maximum value for the T_{Ca} that was solved was 186.8 kPa in the end systolic phase. Due to the effects of variations in fiber strain the average active fiber stress increased further up to 196 kPa. This gave a spatial dependence to the stress that mimicked the fiber strains which averaged to -0.13 throughout the left ventricle. By modeling a change in compressibility of the heart, we see that the volumetric change occurred much more in the center of the left ventricle reaching values of approximately 0.87 whereas it stayed closer to unity near the epicardial surface.

3.2. Effect of Infarction

When apical infarction occurs the myocardial tissues in the affected zone can no longer contract. The lack of contraction means the other tissue must compensate, increasing the overall levels of active contraction seen in (Figure 4). The magnitude of T_{Ca} increased from 186.8 to 195.5 kPa at end systole when the healthy tissue became infarcted. Even though the peak stress did not increase substantially, the average value of T_{Ca} over the cardiac cycle went from 61.9 to 83.2 kPa. However, due to the nonlinear effect that fiber stretch has on active stress, we must look at the myocardial wall strains for a complete picture.

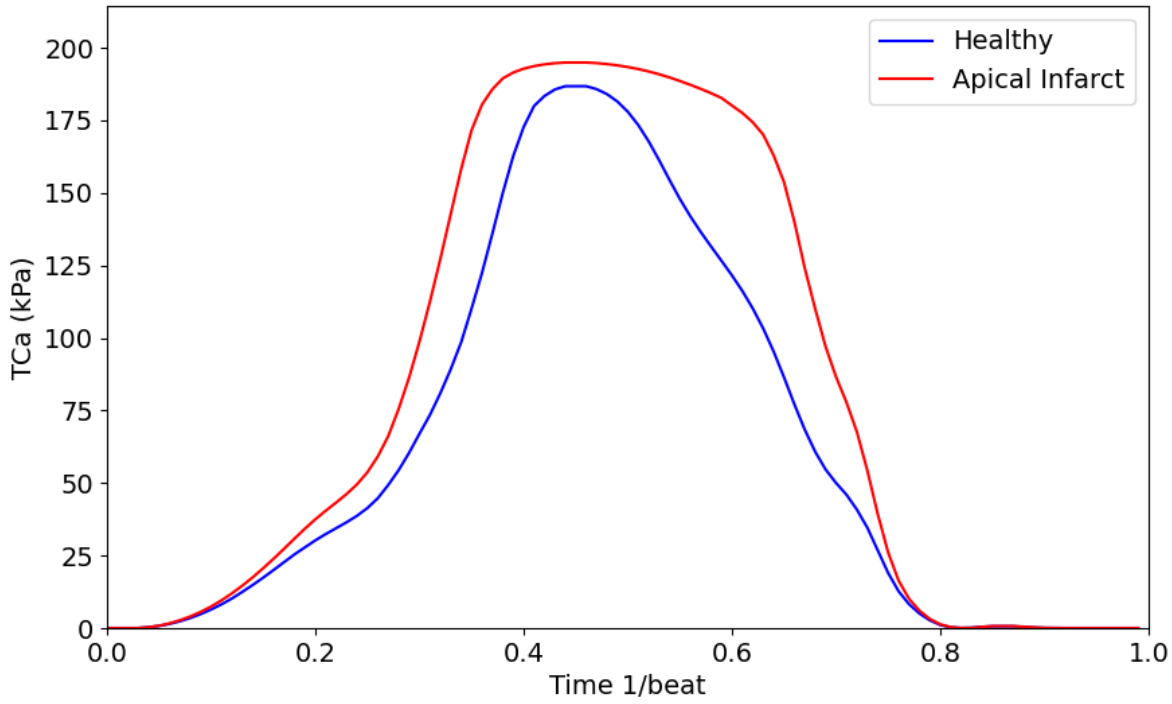

Figure 4: Impact of apical infarction on the value of T_{Ca}. The red curve represents the values for the model with an apical infarction while the blue curve represents the values for a healthy heart model.

Compared to the healthy simulation, the amount of tissue compressed and the level of compression is higher in the apical infarct because the infarction zone has no myofibers contracting with average values of J at end systole being 0.93 in the healthy tissue and 0.81 for the remaining healthy tissue in the infarcted heart. In particular when looking at the transmural variation in J, near the epicardial surface for the healthy heart J is close to 1; whereas the infarcted heart is closer to 0.95. Then it decreases in the transmural direction for both dropping to 0.92 and 0.80 for the healthy and infracted respectively (Figure 5). It is most compressive near the surface of the ventricle and stays nearly incompressible at the epicardium.

Following similar patterns as the volume change, the fiber strains for the infarcted heart have larger magnitudes with more compression in the healthy region and stretching in the infarcted zone (Figure 6). With the fiber strains, we can now see the values of the active stress given by the scalar quantity in Equation 8. The average fiber stress experienced by the healthy model is 196 kPa versus the 228 kPa experienced by the tissue in the infarcted model.

3.3. Benchmarking

3.3.1. Inverse Problem Enhancement

After the code was JIT compiled, the average time for completing one of the inner loops is approximately 30 seconds. The compilation time is done in tandem with the first inverse

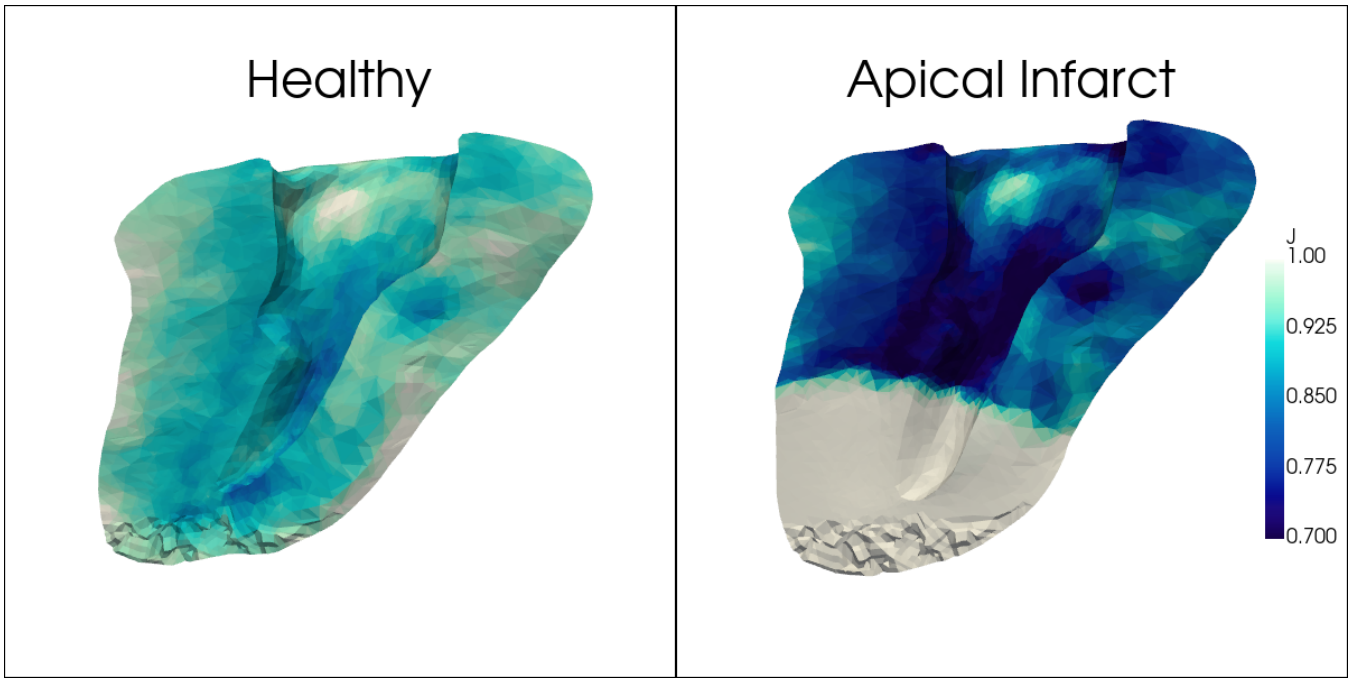

Figure 5: The spatial variation of J is shown above when modeling the tissue as compressible. The left side shows is the healthy heart while the right has an apical infarction. The region of the infarct is identified by the white tissue near the apex that lacks contraction.

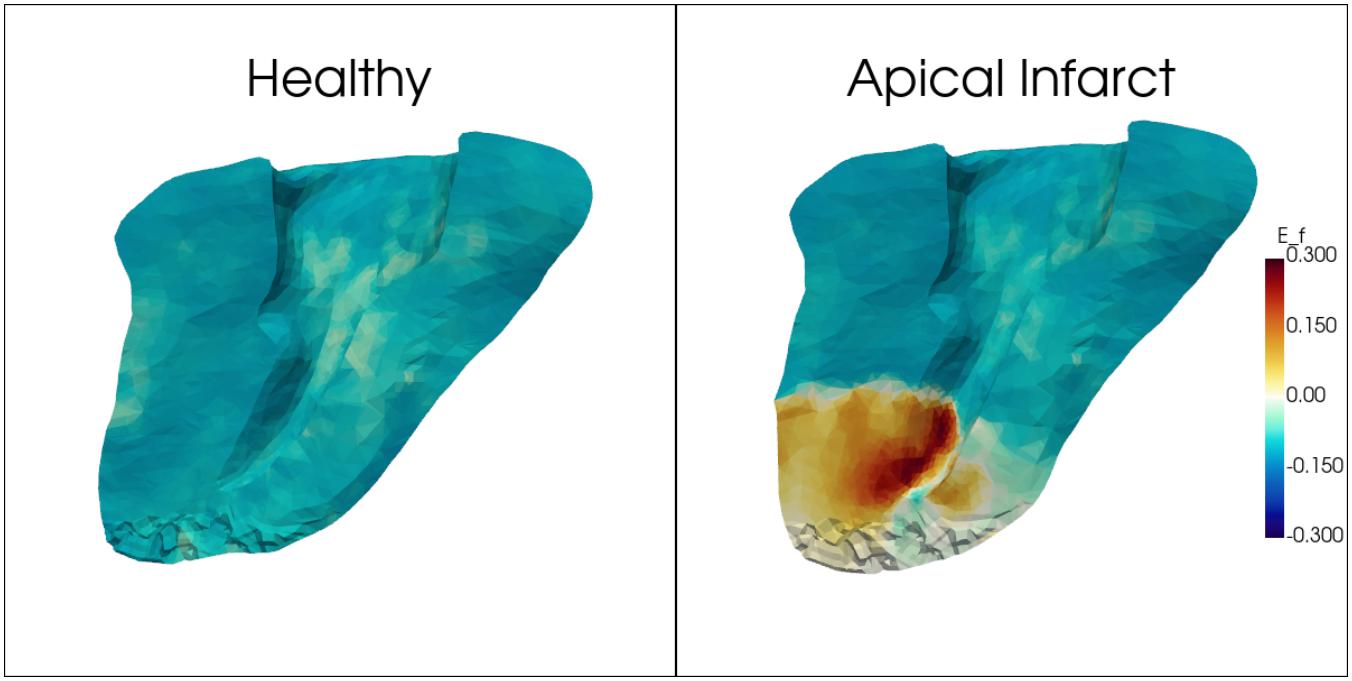

Figure 6: The spatial variation of fiber strain comparing the healthy and infarcted heart. The strain in the infarcted heart is more compressive in the healthy tissue and stretches in the infarcted region.

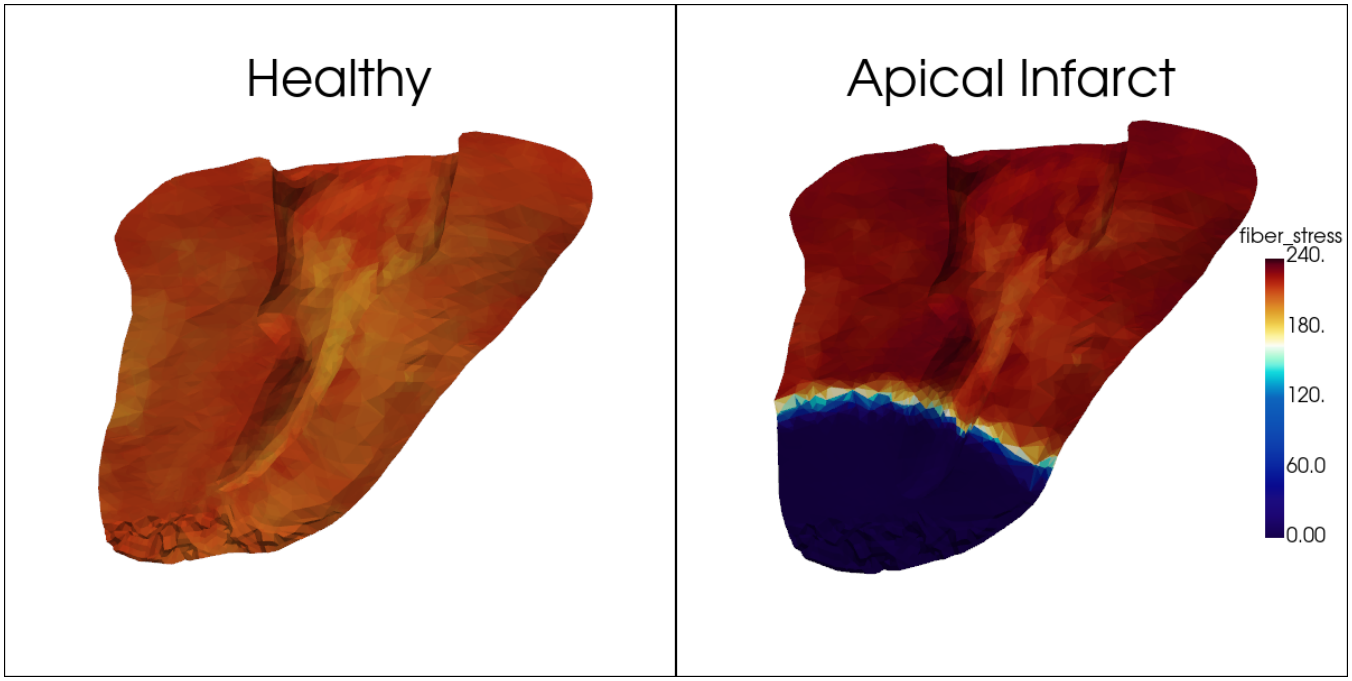

Figure 7: The fiber stress is shown in the healthy and infarcted heart. The left is the healthy heart with a mostly uniform stress distribution while the right one shows the elevated stress of the healthy tissue when effected by apical infarction. The blue region is where no fiber stress is applied, representing the dead tissue.

problem of the system, which solves the end diastolic state, taking 5.33 minutes to solve and compile. The inner loop is solving for the T_{Ca} value at a fixed time point with a given pressure and volume. Solving the entire PV loop amounts to chaining together 100 of these inner loop inverse problems, taking 48 minutes in total. This is about 30x faster compared to the simulations run by Liu et al (Table 1), and we were able to perform 5x the amount of inner loop problems.

Along with comparisons to ABAQUS simulations, similar work was done in FEniCS. While both models aimed at finding the fiber stress, the FEniCS model solved for active strain using a mixed formulation for displacement and pressure. While there were other variations, both simulations used quadratic tets for displacements and optimized via the adjoint method to match volumes. The time to compute the inverse problems in FEniCS, ABAQUS, and CARDIAX are reported in Table 1. The times for CARDIAX include the JIT compilation and offline computation time. CARDIAX was able to solve the inverse problem the fastest out of the three software packages, even outperforming a mesh with less than 20% of the elements.

3.3.2. Cube tests

The results of the benchmark problem were obtained using a single *Nvidia A100 GPU* for running CARDIAX and JAXFEM and a *Intel(R) Core(TM) i9-9920X CPU @ 3.50GHz* for the FEniCSx simulations. We see vast improvements in computational speed across all DoF ranges when CARDIAX is JIT compiled (Figure 9). We focus on the FEniCSx solutions obtained with 8 pro-

Table 1: Runtime comparisons between FEniCS and CARDIAX. The FEniCS simulations were performed over 25 data points throughout the cardiac cycle, while the CARDIAX simulations used 100 data points.

Software	Platform	Adjoints	Data points used	Elements	Total runtime
FEniCS	CPU	True	25	4377	1.35 hours
FEniCS	CPU	True	25	35016	15.6 hours
ABAQUS	CPU	False	20	24841	24.37 hours
CARDIAX	GPU	True	100	24841	0.89 hours
CARDIAX	GPU	True	100	41305	1.78 hours

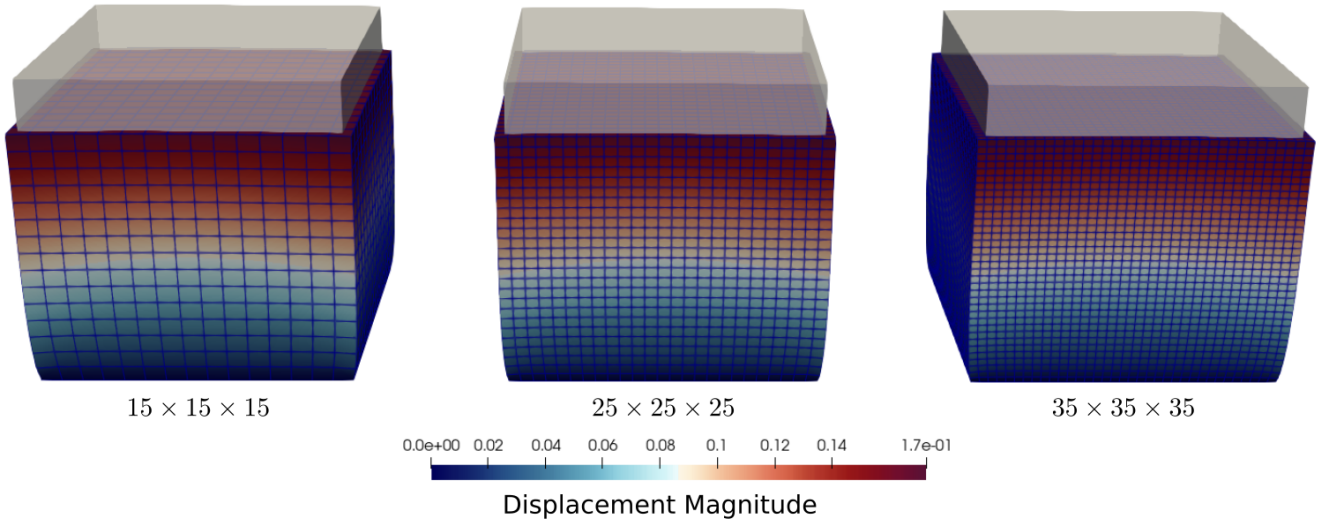

Figure 8: A sample of mesh sizes used as a benchmark for the code. From left to right, the cubes have 15^3 elements (12288 DoFs), 25^3 elements (52728 DoFs), and 35^3 elements (139968 DoFs).

cessors because it outperformed the other MPI runs. Looking at a moderate size of 52,728 DoFs, FEniCSx was capable of solving in 16.10 seconds. The two JAX codes were able to solve in 27.2 seconds with JAX-FEM and solved in 0.46 seconds with CARDIAX once compiled. On the larger DoF side, JAX-FEM solved in 291.13 seconds while CARDIAX took 11.76 seconds compiled and 104.63 seconds to solve while compiling. In both cases, we see that the offline solve that includes compile time still executes faster to obtain a solution.

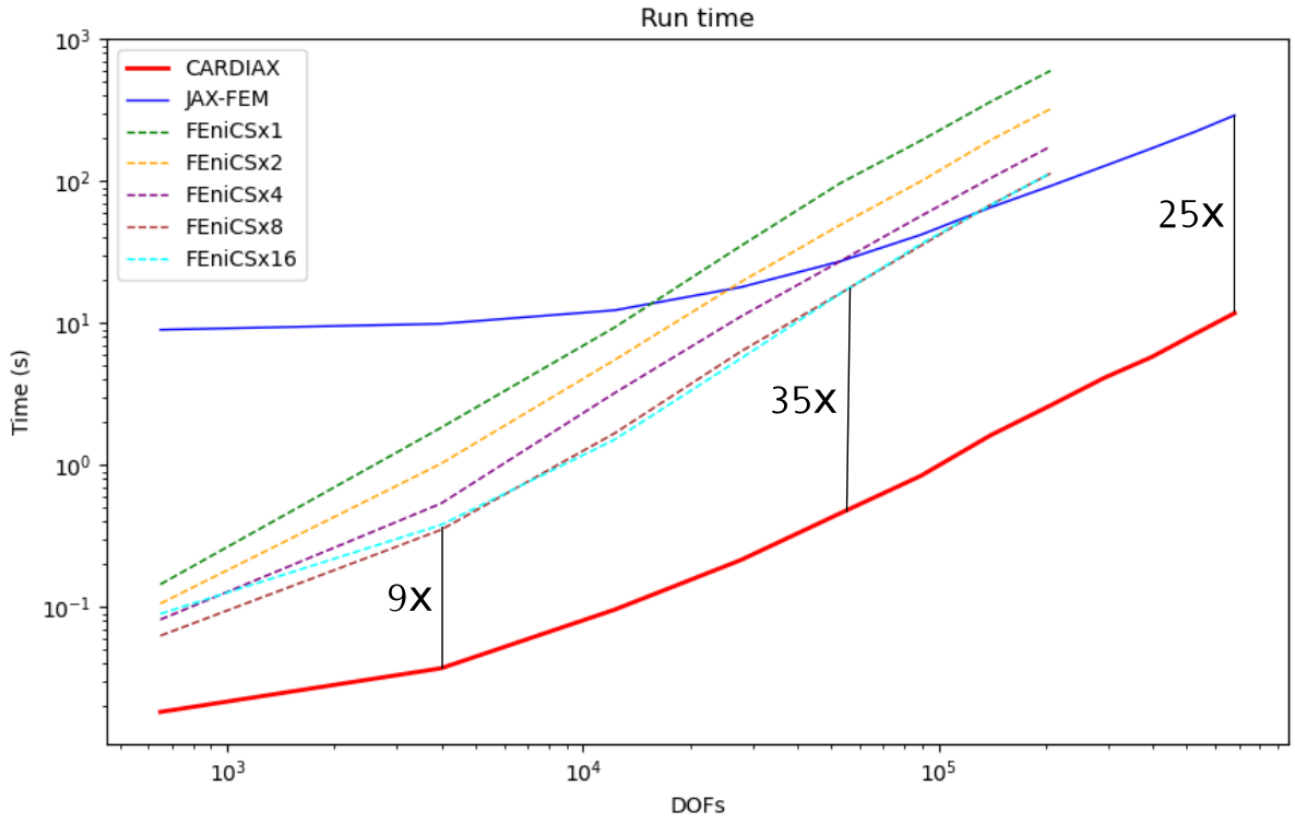

Figure 9: A comparison between CPU and GPU based finite element codes for solving a Neohookean cube in compression. The solid lines represent the GPU based codes, JAXFEM and CARDIAX. The dotted lines represent the FEniCSx comparison where "xN" represents a run with N cores. The left figure shows absolute solve time after compilation time and the figure on the right shows the solve time relative to CARDIAX.

4. DISCUSSION

4.1. Goals of the study

Computational models are becoming more influential in clinical decision making as the models have become more accurate and reliable. One of the major limitations to wider clinical use is the inability to obtain results in a reasonable timeframe. Machine learning shows promise to solve problems where one would like to solve many scenarios in rapid fashion. Along with a many-query problem, we would still like to take advantage of this technology to solve a single, high-fidelity problem. We have shown how GPUs allowed us to drive simulation time for a PV loop to under an hour, making the potential for clinical digital twins more feasible.

4.2. Validation of the Biventricular Model

This work focused on the simulation speed with CARDIAX while maintaining the accuracy determined by Liu et al. with the ABAQUS model. In the healthy and infarcted models, we can see the change in compressibility through the wall of the heart. The volume is mostly preserved near the epicardium and becomes more compressible the closer the tissue is to the endocardium. We see higher levels of compression in the infarcted heart compared to the healthy one since the same volume must be ejected with less active muscle mass. The healthy heart had values of $T_{Ca} = 186$ kPa which had to increase to $T_{Ca} = 195$ kPa for the infarcted heart. The absolute difference is not much, but the magnitude of the stress is also affected by the levels of strain. The increase in the strain makes the average levels of stress 196 kPa in the healthy versus 228 kPa in the infarcted heart. The gap in active stress between the models show good agreement with a 30 kPa increase between the healthy and infarcted models in CARDIAX and a 34 kPa increase in the ABAQUS model. The discrepancies in the values obtained in these simulations are due to the boundary conditions applied to the model. Since the ABAQUS model used Dirichlet conditions on the base rather than springs, we expected the overall stress to be higher. Thus, the higher magnitude of active stresses were expected.

4.3. Improving Simulation Development and Execution

We have shown the capacity for CARDIAX to improve simulation speeds. In a controlled benchmark, CARDIAX was able to solve forward problems 10x faster than FEniCSx for smaller problems, improving to more than 30x faster for larger problems. We have demonstrated the enhancements made to JAX-FEM, maintaining a minimum of 25x improvement in execution time across a range of problem sizes. To determine the ability of CARDIAX to solving cardiac mechanics, we solved for active stress to match a measured pressure-volume loop. Similar studies were done by Liu et al. [10] and Finsberg et al. [11] in ABAQUS and FEniCS respectively. While there were variations between the formulation and implementation in each model, CARDIAX

performed about 15x faster than the FEniCS simulations for similar estimations. CARDIAX has shown improvements in the time required to solve, and its ability to scale the inverse problem to more DoFs. It is also simpler to code because of the functional nature of JAX. By defining the appropriate loss or error function, the user can take the gradient of the function, and the adjoint is used where necessary. The user also maintains some level of control of the adjoint solve, allowing them to change settings if they know certain parts of the problem are stiffer than others.

Another important feature to be emphasized in both JAX-FEM and CARDIAX is that the code is entirely in Python. A downfall for solving inverse problems in ABAQUS is the use of multiple languages where Fortran is used for material models and Matlab is used to wrap around the FE solves. A similar but less impeding issue with FEniCSx is the compilation of the code to execute. For new users, it increases the difficulty of debugging because NaNs cannot be tracked. JAX has a feature to track where NaNs may appear on execution, so the user is able to run the code without compiling to make sure it is implemented correctly. Then compilation can be turned on for the main workflows.

4.4. Limitations and Future Extensions

We have demonstrated CARDIAX as an alternative for extant open source and commerical finite element codes. CARDIAX implicitly uses the Bubnov-Galerkin method where the trial and test functions come from the same function space. FEniCSx allows the flexibility to implement whatever scheme is desired. Another potential bottleneck is the inability to run CARDIAX on multiple compute nodes. It takes advantage of using all the memory a single GPU nodes has the capacity for, but currently it cannot compute across multiple nodes. While there are potential remedies like `mpi4jax`, it will take time to fully implement. Two other limitations are the inability to solve problems with a mixed formulation and a clean way to solve time dependent problems. Mixed formulations appear in cases such as the two field approach for incompressibility with displacement and pressure. These problems mix trial and test functions between the two variables, which is currently not handled. The need to solve time dependent problems is motivated by electromechanics where the bidomain or monodomain models must be solved. Another JAX package, Diffrax, is a differential equation solver that can potentially be integrated to give a suite of time steppers that can be used to solve time-dependent problems.

An important direction is to extend the capability of CARDIAX to also solve PDEs using isogeometric analysis (IGA). Many of the domains where problems are solved in cardiac mechanics would benefit from using splines. The power of splines comes from their ability to couple different geometries of the problem through the parametric domain; whereas isoparametric methods are solved for one geometry. They also can describe the mesh with fewer nodes making them a very expressive geometric descriptor. This was already shown to be beneficial, especially on a

complex problem such as contact mechanics [23].

Lastly, CARDIAX has the potential to have major impact on the development of machine learning methods for computational mechanics. Since the software is written entirely in JAX, all of the machine learning extensions that have been created for JAX are capable of being integrated. We would recommend the use of Equinox for machine learning with CARDIAX because the library is already utilized throughout the codebase. By using the `filter_jit` function, neural networks should be able to fit seamlessly into any part of the finite element framework. It also offers various extensions that are helpful for scientific computing such as Lineax, Diffrax, and Optimistix.

4.5. Conclusion

We have presented CARDIAX as a software platform to leverage modern GPU technology to enhance the time savings for solving cardiac mechanics. The speed gains were demonstrated on a PV loop which requires solving many inverse problems sequentially. We also showed the benefits through a simple benchmark problem comparing with FEniCSx and JAXFEM. The code will continue to be improved upon by incorporating machine learning with the NNFE method and IGA through Lagrange Extraction. These developments will help us lay the foundation for making patient-specific disease modeling a useful tool in the clinic.

5. Conflicts of Interest

None.

6. Acknowledgments

The authors gratefully acknowledge the funding provided by the Platform for Advanced Scientific Computing of the Swiss Federation to MSS, the National Institutes of Health R01 HL073021 to MSS and the Oden Institute CSEM Fellowship to BJT.

References

References

[1] S. A. Niederer, M. S. Sacks, M. Girolami, K. Willcox, Scaling digital twins from the artisanal to the industrial 1 (5) 313–320. doi:10.1038/S43588-021-00072-5.

[2] M. Fedele, R. Piersanti, F. Regazzoni, M. Salvador, P. C. Africa, M. Bucelli, A. Zingaro, L. Dede', A. Quarteroni, A comprehensive and biophysically detailed computational model of the whole human heart electromechanics 410 115983. doi:10.1016/j.cma.2023.115983.

[3] N. A. Trayanova, Whole-heart modeling: applications to cardiac electrophysiology and electromechanics, Circ Res 108 (1) (2011) 113–28. doi:10.1161/CIRCRESAHA.110.223610.
URL http://www.ncbi.nlm.nih.gov/pubmed/21212393

[4] P. C. Africa, life: A flexible, high performance library for the numerical solution of complex finite element problems 20 101252. doi:10.1016/j.softx.2022.101252.

[5] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, G. N. Wells, The fenics project version 1.5 Vol 3 ¡strong¿Starting Point and Frequency: ¡/strong¿Year: 2013¡/p¿. doi:10.11588/ANS.2015.100.20553.

[6] I. A. Baratta, J. P. Dean, J. S. Dokken, M. Habera, J. S. Hale, C. N. Richardson, M. E. Rognes, M. W. Scroggs, N. Sime, G. N. Wells, Dolfinx: The next generation fenics problem solving environment. doi:10.5281/ZENODO.10447666.

[7] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: composable transformations of Python+NumPy programs (2018).
URL http://github.com/jax-ml/jax

[8] T. Xue, S. Liao, Z. Gan, C. Park, X. Xie, W. K. Liu, J. Cao, JAX-FEM: A differentiable GPU-accelerated 3D finite element solver for automatic inverse design and mechanistic data science 291 108802. doi:10.1016/j.cpc.2023.108802.
URL https://www.sciencedirect.com/science/article/pii/S0010465523001479

[9] J. S. Soares, D. S. Li, E. Lai, J. H. Gorman, R. C. Gorman, M. S. Sacks, Modeling of myocardium compressibility and its impact in computational simulations of the

healthy and infarcted heart., Functional imaging and modeling of the heart : ... International Workshop, FIMH ..., proceedings. FIMH 10263 (2017) 493–501. doi:10.1007/978-3-319-59448-4_47.

[10] H. Liu, J. S. Soares, J. Walmsley, D. S. Li, S. Raut, R. Avazmohammadi, P. Iaizzo, M. Palmer, J. H. Gorman, R. C. Gorman, et al., The impact of myocardial compressibility on organ-level simulations of the normal and infarcted heart, Scientific reports 11 (1) (2021) 1–15.

[11] H. Finsberg, C. Xi, J. L. Tan, L. Zhong, M. Genet, J. Sundnes, L. C. Lee, S. T. Wall, Efficient estimation of personalized biventricular mechanical function employing gradient-based optimization 34 (7). doi:10.1002/cnm.2982.

[12] A. Gerbi, L. Dede', A. Quarteroni, A monolithic algorithm for the simulation of cardiac electromechanics in the human left ventricle 1 (1) 1–37. doi:10.3934/mine.2018.1.1.

[13] S. Marchesseau, H. Delingette, M. Sermesant, N. Ayache, Fast parameter calibration of a cardiac electromechanical model from medical images based on the unscented transform 12 (4) 815–831. doi:10.1007/s10237-012-0446-z.

[14] G. A. Holzapfel, Nonlinear solid mechanics: a continuum approach for engineering science (2002).

[15] V. Y. Wang, H. Lam, D. B. Ennis, B. R. Cowan, A. A. Young, M. P. Nash, Modelling passive diastolic mechanics with quantitative mri of cardiac structure and function 13 (5) 773–784. doi:10.1016/j.media.2009.07.006.

[16] P. J. Hunter, A. D. McCulloch, H. E. ter Keurs, Modelling the mechanical properties of cardiac muscle, Prog Biophys Mol Biol 69 (2-3) (1998) 289–331.
URL http://www.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=m&form=6&dopt=r&uid=0009785944

[17] C. M. Wacker, F. Wiesmann, M. Bock, P. Jakob, J. J. Sandstede, A. Lehning, G. Ertl, L. R. Schad, A. Haase, W. R. Bauer, Determination of regional blood volume and intra-extracapillary water exchange in human myocardium using feruglose: First clinical results in patients with coronary artery disease 47 (5) 1013–1016. doi:10.1002/mrm.10125.

[18] F. C. Yin, C. C. Chan, R. M. Judd, Compressibility of perfused passive myocardium, Am J Physiol 271 (5 Pt 2) (1996) H1864–70.
URL http://www.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=m&form=6&dopt=r&uid=0008945902

[19] R. Avazmohammadi, J. S. Soares, D. S. Li, T. Eperjesi, J. Pilla, R. C. Gorman, M. S. Sacks, On the in vivo systolic compressibility of left ventricular free wall myocardium in the normal and infarcted heart., Journal of biomechanics 107 (2020) 109767. doi:10.1016/j.jbiomech.2020.109767.

[20] E. L. Allgower, P. H. Schmidt, Computing volumes of polyhedra 46 (173) 171–174. doi:10.1090/s0025-5718-1986-0815838-7.

[21] P. Kidger, C. Garcia, Equinox: neural networks in JAX via callable PyTrees and filtered transformations, Differentiable Programming workshop at Neural Information Processing Systems 2021 (2021).

[22] J. Rader, T. J. Lyons, P. Kidger, Lineax: unified linear solves and linear least-squares in JAX and equinox abs/2311.17283. arXiv:2311.17283, doi:10.48550/ARXIV.2311.17283.

[23] K. Meyer, C. Goodbrake, M. S. Sacks, A neural network finite element trileaflet heart valve model incorporating multi-body contact 41 (4). doi:10.1002/cnm.70038.

7. Appendix

7.1. Parameters for Biventricular Model

Parameter	c	α	β	A_1	A_2	A_3	K	γ
Value	1522.083	2.125	1.4	12	8	26	10^7	51

Table 2: Parameters used in the material model

7.2. Parameters for Benchmark

Parameter	E	ν	Traction
Value	10	0.3	2

Table 3: Parameters used in the PDE

7.3. Major Components of CARDIAX

1. `FiniteElement`: defines the discrete finite element evaluations on the given mesh. Contains information such as quadrature points for the function and derivative on the element. Methods are included to move the cell information to the local DoF system.

2. `Problem`: handles the global system, evaluating the weak form of the problem, and obtaining the linearization of the PDE. The methods used here are now all JIT compiled at solve time to compute linear system. It handles the collection of local to global information. This class also contains debugging functionality to view the system being solved, giving insights to adapting the solver.

3. `Solver`: solves the final linear system obtained by problem. The `SparseMatrixLinearOperator` defined in the solver defines the discretized linear approximation of the operator governing the system. Thus, the dirichlet boundary conditions are defined in the action of the operator. The method of solving the linear system can be changed as well as tolerances and initial guesses. These changes are done in an object oriented way, so the functions don't have to be recompiled for majority of changes.

7.4. Example of Code used for inverse Problem

```
1    from cardiax.fe import FiniteElement
2    from cardiax.problem import Problem
3    from cardiax.solver import Newton_Solver
4
```

```
    # Read in mesh
    mesh = meshio.read("my_mesh.msh")
    # Create desired function space
    fe = FiniteElement(mesh, vec=3, dim=3, ele_type="tetra10", gauss_order=2)
    # Define Boundary Conditions
    location_fns = ...
    # Create Problem object from PDE
    problem = VHL_Fung_Compressible(fe, location_fns=location_fns)

    # Set parameter constants and mesh data
    # like fiber directions
    problem.set_params(...)

    # Create solver object for Newton's method
    solver = Newton_Solver(problem, initial_guess=np.zeros((problem.
num_total_dofs_all_vars)))

    # Wrap solver.solve to do adjoint calculation
    adjoint = solver.ad_wrapper()

    def error(sol, V_true):
        vol = calc_volume(sol)
        return vol - V_true

    def composed(T_Ca, pressures, V_true):
        sol = fwd_pred([T_Ca, pressures])
        return error(sol, V_true), sol

    val_and_grad = jax.value_and_grad(composed, has_aux=True)

    # Perform outer loop
    for p, v in zip(pressures_list, volumes_list):
        problem.set_params(p)
        composed.set_volume(v)

        # Solve inner loop
        while np.abs(loss) > tol:
            solver.initial_guess = sol

            vals = val_grad(TCa_full, pressures, v)
            loss = vals[0][0]
            sol = vals[0][1]
            dTCa = vals[1].sum()
            TCa_full = update_TCa(dTCa)
```