FAST PLANAR LINKAGE MECHANISM SOLVER AND AUTOMATIC DIFFERENTIATION FOR GRADIENT-BASED OPTIMIZATION AND PATH SYNTHSIS

AMIN HEYRANI NOBARI

1. Introduction. The longstanding challenge in engineering design has been the analysis and synthesis of various kinematic mechanisms, a topic that has engaged countless engineers and scientists over time [17]. The understanding of the design of intricate kinematic systems is still elusive, and typically is restricted to certain tasks. The design process often relies on a combination of trial and error, specialized knowledge, and heuristics to discover effective designs. However, the emergence of computational design strategies in recent years has prompted a shift towards inverse kinematic design via optimization techniques [17, 3, 4, 28, 11, 1, 13, 19, 5, 7, 29, 8, 9, 2, 21, 14].

In the realm of computational approaches, the application of data-driven strategies for inverse design has garnered increasing interest. This trend has spurred a surge in research centered on the utilization of statistical machine learning and deep learning models in inverse kinematics [29, 8, 9, 14, 12] and in engineering design more generally [22]. Yet, the effectiveness of many computational and data-driven methodologies is hindered by the computational power required for mechanism simulations. Some methods demand millions of simulations for efficient inverse kinematic design [12], and others are confined to certain types of basic mechanisms, like 4-bars or 6-bars, among others [4, 28, 11, 1, 13, 19, 5, 7, 29, 8, 9, 14].

The primary constraint stems from the vastness of the design space – the sheer variety of problem requirements and mechanism variations is virtually boundless, which makes the exploration of more sophisticated mechanisms a daunting task, potentially necessitating billions or even trillions of simulations. Therefore, to facilitate the investigation of this immense design space for inverse kinematic synthesis, it is crucial to have computational tools equipped with faster solvers.

In this work we focus on the problem of kinematic synthesis of planar linkage mechanisms, however, unlike most of the work done more recently, we do not limit our work to specific topologies of mechanisms (such as 4-bars, 6-bars, *etc.*) and go beyond the well understood and further investigated mechanisms. By exploring a wide range of simple and complex 1-DOF (degrees of freedom) planar linkage mechanisms, we attempt to provide a more general picture of the problem and address the limitations at the most generalizable scale possible. In these kinds of mechanisms, the problem of inverse kinematic synthesis can have different types of goals, such as coupler path synthesis, motion generation, and signal generation [18]. The path synthesis problem can be described as the problem of designing linkage mechanisms that can generate a particular path that is described by a finite series of point coordinates. Motion generation can be thought of as the generalized version of the path synthesis where aside from point coordinates, the orientation of an attached rigid body (such as a robot arm) is also prescribed. Finally, the function generation problem refers to the problem of generating a specific series of output crank angles (or slider positions) at given angles (or positions) at the actuator, essentially transforming the signal from the actuator (angle or position) to a different signal at the output crank (or slider). See Figure 1 for more details. This work is created with a primary focus on the "Path Synthesis" problem, although with some tweaking the findings of this work could be

adapted to be used for other types of problems such as "Function Generation" and "Motion Generation" [18].

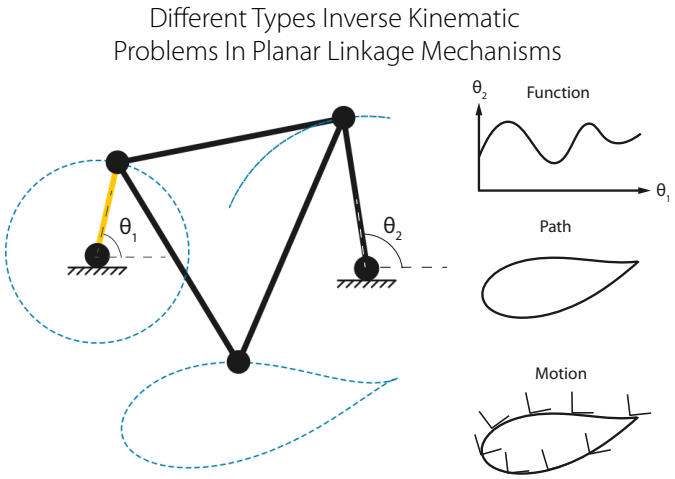

FIG. 1. *Different types of problems in inverse kinematics of planar linkage mechanisms. Note that the yellow arm is the actuator arm*

As mentioned before, one of the main limiting factors for computational approaches in inverse kinematic design is the need for a significant number of simulations. In fact the current state-of-the-art method for path synthesis uses reinforcement learning (RL) for path synthesis, however the RL agent needs to be retrained for each target curve to find a solution [12]. In this work, the authors require 2 million simulations of mechanisms with up to 11 joints for every target curve. Other approaches based on genetic programming [17] and genetic algorithms have a similar limitation since they require even larger amounts of simulations to find feasible solutions for path synthesis problems sometimes requiring billions of simulations. This can be rather time-consuming using conventional solvers currently being used by most researchers. Given this, in this work, we develop highly optimized solvers for planar linkage mechanisms based on the solver proposed by Bächer *et al.* [3]. We take this solver and optimize the code for the solution by vectorizing the process for multiple timesteps and vectorizing the process for multiple timesteps and multiple mechanisms simultaneously to take advantage of the highly optimized underlying linear algebra packages in Julia. Furthermore, we further speed up the already accelerated solver even more by implementing GPU-based solvers for the problem which speed up the process even more. We demonstrate that these optimizations provide multiple orders of magnitude improvement over a naive solver and speed up the simulation process massively. We hope that with the advent of these faster solvers even existing methods can be accelerated massively leading to faster or better optimizers that not only perform inverse kinematic synthesis faster but are also capable of exploring more complex mechanisms. We detail our work and the implementation of the fast solvers in the sections that follow.

Finally, we take advantage of the automatic differentiation tools developed for Julia, specifically the ForwardDiff.jl package, to develop a gradient-based optimizer for planar linkage mechanisms to allow for the refinement of planar linkage mechanisms for path synthesis problems. The goal of this optimizer is to take candidate solutions

that are capable of producing paths very similar to a given target path and adjust the mechanism to achieve as close a match to the target curve as possible. Existing gradient-based methods [3] are limited to exceedingly minor changes when it comes to path matching because these methods introduce a penalty to the objective of the solver to prevent mechanisms from falling into a locking configuration, however, this makes the optimizer limited to more smooth paths and paths that are produced by mechanisms which are sufficiently away from locking configuration, as otherwise the penalty factor in the optimizer becomes the dominating factor in the gradient [3]. In this work, we propose a novel approach for dynamically changing the weight of the penalty term in the optimizer's objective to enable more effective gradient-based optimization for planar linkage mechanisms. the details of how we achieve this are provided in the sections that follow.

2. Background & Related Works. In this section, we will discuss some of the prior works which we utilize in our approach and discuss the overall topic of kinematic synthesis briefly.

2.1. Computational Inverse Kinematics. Computational approaches in the inverse kinematics problem fall into three primary categories: a) Numerical-atlas approaches, b) Optimization-based approaches, and c) Data-driven approaches. Below we briefly discuss each approach in the context of how our work can benefit each category of approaches.

Numerical atlas-based approaches:. The initial strategy involves building a repository of mechanisms, which subsequently generate paths that function like a "numerical atlas". This atlas can be used to find the nearest paths to any designated path and employ the corresponding mechanism from the current database as a solution. Additionally, integrating this retrieval step with local optimization of the mechanism can bring us closer to the desired path [19, 6, 26]. In the majority of cases, the numerical atlas is generally restricted to a particular mechanism or a few types of mechanisms, such as a four-bar or six-bar mechanism. This is primarily due to the computational constraints of simulating numerous complex mechanisms, and the creation of a large numerical atlas of intricate mechanisms would require excessive simulations. Searching for viable mechanisms in an expansive design space entails simulating countless mechanisms, only to discover that they are locked or degenerate and thus unsuitable for inclusion in the numerical atlas. As a result, these methods often only apply to a limited range of simpler mechanisms. Furthermore, these basic mechanisms, with a few joints, can only generate a limited variety of paths. For instance, it is known that a four-bar mechanism can precisely match at most five points of a path (and even this is not always feasible) [23]. This demonstrates that even with a substantial atlas, the array of feasible paths that can be traced is limited. However, our work can address this issue by creating highly efficient solvers that facilitate the creation of extensive numerical atlas databases, accommodating a higher level of complexity and more joints.

Optimization-based approaches:. The second computational strategy is referred to as the optimization-based approach. This category encompasses various works that utilize a range of optimization algorithms to determine the most appropriate mechanisms for a specific target path. While some scholars employ genetic algorithms or genetic programming techniques to produce mechanisms that can follow desired paths [17, 13], others opt for optimization using Fourier descriptors [27, 31]. An additional approach involves gradient-based optimization [3] to modify an existing machine, although this method requires an already approximate solution to yield any

benefits. Apart from a few exceptions [17, 3], most of these methods are limited to altering existing mechanisms through optimization [3], or they are limited to a specific subset of problems. For example, Lipson *et al.* [17], focused on solving the straight-line problem using genetic programming. The performance of population-based optimization approaches, such as genetic algorithms, is mostly reliant upon the volume of simulations that can be done in practical time. That is the size of the populations and the number of generations the algorithm can be iterated over will largely determine the level of success these approaches will have. And with conventional solvers, the volume of simulation is highly limited. In our work, we develop extremely fast solvers which potentially increase the volume of simulations by multiple orders of magnitude allowing for a much larger size of population and more generations to be evaluated within the same amount of time which can hugely improve the performance of such approaches. Beyond this, we also develop a gradient-based optimization scheme that can refine mechanisms with better performance compared to the existing state-of-the-art gradient-based methods such as the one proposed in [3].

Data-driven approaches:. In recent times, the popularity of machine learning-based methods has surged, leading to a number of published works that adopt a data-driven perspective. Most of these studies incorporate the previously mentioned "numerical atlas" and optimization strategies within data-driven frameworks. For instance, Deshpande and colleagues combined the numerical atlas method with optimization in their research [7, 8, 9]. They utilize variational autoencoders (VAEs)[15] and clustering-based search techniques to identify suitable candidates capable of generating a desired coupler curve. In their subsequent studies, they apply VAEs and conditional VAEs[25] to synthesize mechanisms.

The datasets employed in such studies are typically small and restricted to certain types of mechanisms (such as four-bar, six-bar, etc.). For example, a dataset of 6818 linkage mechanisms is used in [10]. These models could substantially benefit from larger datasets comprising millions of mechanisms. Other data-driven studies have attempted to generate mechanisms conditioned on paths [29], but these are again limited to four-bar mechanisms.

In contrast to these "numerical atlas" adaptations, some researchers have endeavored to translate the optimization approach into a machine learning framework. One such study applied deep Q learning [20] and Lipson's T and D operators [2]. Although these reinforcement learning (RL) based approaches are not confined to specific mechanisms, they require retraining for each new target shape. More recently, Fogelson *et al.*, proposed a new RL-based approach that was able to beat all existing approaches [12] in accuracy, however, as mentioned before the limitation of these approaches is that they need retraining for every new target shape and specifically in this latest work each new target requires 2 million simulations to be done. What is evident is that machine learning approaches show great promise, however, at the moment, the same limitations that we saw in optimization and numerical atlas approaches can be observed in this category of approaches as well. As such our work can significantly benefit this type of approach as well.

2.2. Simulation of Kinematics. There has been substantial work done in solving 1-DOF mechanisms, however, as mechanisms get more complex, solving them becomes costly and the complexity of the closed-form analytical equations becomes gargantuan. As a result, algorithms-based and numerical approaches to solving such systems are typically employed [30, 24]. The literature on this topic is extensive and beyond the scope of this report, however, there are a few relevant works that

we will discuss here as they set up the context for future discussion. Broadly, two different approaches can be considered in simulating mechanisms beyond analytical approaches [24]. One approach is the numerical approach to solving kinematic systems. An example of such an approach is Lipson's simulator, used for the genetic programming approach in [16]. In most numerical approaches planar mechanisms are solved using numerical algorithms used for solving systems of non-linear equations (such as Newton-Raphson or Broyden's method), these approaches are capable of simulating very complex systems, however, in many complex systems the solution is not unique and these simulators only produce one of the possible results [30, 24]. Despite this, these numerical approaches are the most general solvers that can handle all types of mechanisms which graphical approaches might not be capable of, however, in the context of inverse kinematic design this matters less as we see. The other approach to solving linkage mechanisms is to take a graphical approach and solve planar mechanisms from a purely kinematic approach. One such simulation approach which focuses purely on kinematics is the one proposed by Bächer *et al.* [3]. The simulator proposed by Bächer *et al.*solves a linkage system iteratively by starting from known values such as the position of the ground joints and the current position of the actuator arm (which can be determined based on the velocity profile of the motor) and solving for any joints that can be solved with the available information (taking into account initial positions of the joint). At every iteration, more joints will be solved until at the final iteration where all joints are solved. This approach is illustrated in Figure 3. Unlike the numerical solvers, these solvers cannot handle all kinds of mechanisms and are limited to dyadic loops only, however, systems with complex kinematic loops either can be converted to equivalent dyadic mechanisms and if they cannot be converted to such mechanisms they must have a non-unique solution or "Branch Defects" which from a design perspective makes them undesirable as their kinematics are not predictable and as such their use in real-world applications require special considerations. Therefore, while these approaches are limited to mechanisms with simple kinematic loops consisting of dyadic loops from a designs perspective they offer a near-complete representation of the design space, furthermore, an advantage of using them is that the gradients of simulation can be obtained in a similar manner which enables gradient-based optimization (*e.g.*, editing existing mechanisms to fit certain constraints [3]). We adopt the approach of Bächer *et al.*and discuss the details of our work in the following sections. For a more in-depth view of simulation methods, readers are referred to [24].

3. Methodology.

In this section, we will discuss the details of our methodology for both accelerating simulations and performing gradient-based optimization. First, we will discuss the solver we use and our implementations of it, then we will discuss how we approach the gradient-based optimization and our contributions there.

3.1. Accelerating Graphical Solvers.

As mentioned before we adopt the solver proposed by Bächer *et al.* [3] and introduce some improvements to the implementation to speed up the simulation process. In this approach, we take any given mechanism with simple kinematic loops and rather than performing a dyadic decomposition [24] to identify four-bar loops, take an iterative approach to find the solutions by modeling mechanisms as graphs. We can solve for any joint which has two known neighbors (*i.e.*, two joints with currently known positions at a given timestep that have linkages connecting to the node we are trying to solve). To solve for a joint with two known neighbors, we can use the initial positions of the joints and the current positions of the known neighbors to solve for the unknown joint. Take the example

illustrated in Figure 2. In this example, joints 1 and 2 are solved at time step t, given this and the fact that we know the initial positions of the joints we can solve for the angle θ the linkage between joints 1 and 3 takes in this time step using the following equation:

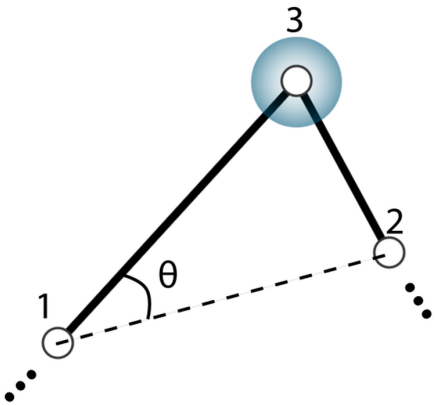

FIG. 2. *This figure shows the simple case of solving for the angle theta for an unsolved joint 3 using the solutions for joints 1 and 2.*

$$\theta = \text{sign}\left(\left[(X_{0,1} - X_{0,3}) \times (X_{0,1} - X_{0,2})\right].\hat{e}_z\right) \times \cos^{-1}\left(\frac{\|X_{0,1} - X_{0,3}\|_2^2 + \|X_1 - X_2\|_2^2 - \|X_{0,2} - X_{0,3}\|_2^2}{2\,\|X_{0,1} - X_{0,3}\|_2\,\|X_1 - X_2\|_2}\right) \tag{3.1}$$

Where X_0 is an $N \times 2$ matrix of initial positions ($X_{0,1}$, for example, is a 2D vector indicating the initial position of joint 1) and X is an $N \times 2$ matrix of the positions of joints at the current timestep. As evident this is simply applying the cosine rule to find the angle the linkage takes. Then once this angle theta is determined we can determine the position of our unknown joint 3 using the following equation:

$$X_3 = X_1 + R(\theta)\frac{(X_2 - X_1)\,\|X_{0,3} - X_{0,1}\|_2}{\|X_2 - X_1\|_2} \tag{3.2}$$

Where $R(\theta)$ is the 2D rotation matrix for a given angle theta. One important thing to note here is that the value of the term inside the inverse cosine function in (3.1) can become larger than 1 or smaller than -1 which leads to no solution. This happens when a mechanism is locking or degenerating. Determining the exact timesteps that the mechanism locks at would require solving massive impractical non-linear equations as such in most cases the practical solution to identifying locking mechanisms is by refining the timesteps of the simulation to ensure that the mechanism does not lock at any point. This is one of the reasons why fast solvers are necessary as many high-fidelity simulations are needed just to identify feasible mechanisms.

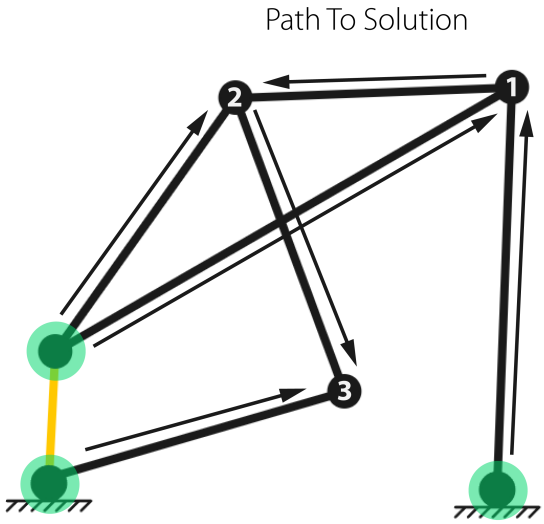

FIG. 3. *Here we illustrate the path the solver takes to find the solution. At first, the solver starts with the known joints (*i.e.*, fixed and actuated joints) and at every step nodes with two known neighbors can be found, in this mechanism illustrated, the path to the solution has 3 steps. The numbers of the joints indicate the order in which the solution is found and the arrows indicate which two neighboring joints are needed to solve the given joint. Known nodes are highlighted in green.*

Now that we established how we can solve for joints with two known neighbors all that is left to do is to find a path to solving a given mechanism for each timestep. Starting from joints with always-known positions at all timesteps (actuator arm and fixed joints), we start by identifying joints with two known neighbors these joints are what we can solve for first at every timestep. Then we repeat the process but with the joints, we identified in the prior step considered as known joints and determine which joints can be solved for given the solution to these joints. We repeat this process until all the joints in the mechanism have been solved. This process is similar to a breadth-first search in graphs and it is visualized for a simple mechanism in Figure 3. In doing this a path to the solution of all joints is found which can be used to find solutions at different timesteps. Note that this process only needs to be done once for every mechanism to find the path to the solution and to solve all the timesteps we would only traverse this already established path. The process for finding the path to the solution is described in Algorithm 3.1.

Once the path to the solution is found we can then run the simulation for all the necessary timesteps. We can perform this task naively by iterating through each timestep separately in a loop, however, as we mentioned in (3.1) the solution to each timestep does not rely on prior timesteps therefore these calculations can be made in parallel. As a baseline, we can look at the naive algorithm described in Algorithm 3.2. One obvious way to accelerate this simulator is simply run the timestep for loop in parallel using multiple threads. This will immediately speed up the simulations significantly depending on the number of threads available. However, using high-level Julia code to do this will not allow us to optimize the simulation speed to the maximum speed it can get to as low-level optimization is necessary for such a thing. The good news is that the underlying linear algebra packages implemented in Julia use highly optimized code which has been optimized as much as possible over decades of

Algorithm 3.1 Path Algorithm

Require: initialNodes fixed joints and motor
Require: activeList
  for all $i \in$initialNodes **do**
    activeList.insert(neighbors(initialNodes(i)))
  end for
  while !activeList.empty() **do**
    k = activeList.pop front()
    vn = visitedNeighbors(k)
    if vn.size()¿ 1 **then**
      i = vn(1), j = vn(2)
      addRule(i,j,k) Add the solution dyadic $(i, j) \to k$ to the list of operation steps
      assignNextIndex(k), setVisited(k)
      activeList.append(unvisitedNeigbors(k))
    else
      activeList.push back(k)
    end if
  end while

research on matrix and vector operations. Therefore what we need to do to accelerate the solver to the maximum extent possible is to vectorize the process of finding the solution for all time steps in one go using only vector and matrix operations. In this way, we replace the entire for loop in Algorithm 3.2 for timesteps with a single function or a few lines of code that compute the solution for all timesteps at once. For the sake of brevity, we do not provide the specific code in the main body of the report but rather include it in Appendix A. By doing this we only use built-in linear algebra operations (*i.e.*matrix multiplication and vector operations) which are highly optimized in the backend of Julia. In this way, we take advantage of the low-level optimization done for linear algebra packages and with minimal effort achieve great code efficiency.

Algorithm 3.2 Naive Solver

Require: X_0 initial positions of joints
Require: initialNodes, NTimesteps
Require: X = zeros(NTimesteps,NJoints,2)
  Path ← Use Algorithm 3.1 to find the path to the solution
  for $t \leftarrow 1$ to NTimesteps **do**
    Compute X[t,initialNodes] fixed joints and actuator position at timestep t
    for all step ∈ path **do**
      (i,j,k) ← step
      X[t,i] ← Use (3.1) to compute i using j and k
    end for
  end for
  return X

Beyond vectorizing the simulation for all timesteps in one mechanism, we can take this vectorization to another level, and vectorize the process for a batch of mechanisms. That is to say that we simulate not just for all the timesteps at the same time

but rather solve for all the timesteps and all the mechanisms in a batch at the same time. However, to do this we need to take into account a few considerations. Vectorized processes require all the mechanisms to have the same number of joints so as to make the matrices and vectors representing all the mechanisms the same size so that we can batch them into higher dimensional arrays and apply built-in linear algebra operations in batch. However, different mechanisms can have different sizes, therefore, we need to somehow resize the mechanisms without changing their kinematics. Luckily, this is a fairly simple thing to achieve. When batching mechanisms we can simply add fixed joints that are not connected to anything, and therefore do not alter the kinematics of the mechanism. Using this approach we simply add as many fixed joints to mechanisms in a batch to make sure that they are all the same size (*i.e.*, the maximum size, and in this work we limit the size to mechanisms with 20 joints). By doing this we make sure that all the mechanisms in a batch are of the same size. As it can be seen in Appendix A in the vectorized solver we still compute the path for a mechanism and traverse through the path to the solution, which is different for each mechanism. However, if we wish to solve a batch of mechanisms this is not going to be possible as batch operations must be applied at the same time, and this cannot be done in different paths simultaneously. Therefore, the only way we can simulate a batch of mechanisms is if they all have the same path to the solution. This however is clearly not the case for different mechanisms, but this does not have to be the case in general. To address this all we have to do is sort the joint ordering for all mechanisms in a batch such that the path to the solution for all mechanisms is to simply start with the first joint and compute the solution one joint at a time in order. Therefore, to overcome this second challenge we simply first sort all the mechanisms in a batch as well. For the sake of brevity, we do not go into the algorithm and code details here and include the Julia code for both preprocessing a batch and simulating a batch in Appendix B.

At this point, we have discussed the three main ways that we can accelerate the solver; 1) Multi-Threding, 2) Vectorizing over timesteps, and 3) Vectorizing over timesteps and batches of mechanisms. In the sections that follow we will experiment with these approaches and demonstrate that the proposed methods truly do accelerate the simulation of the mechanisms significantly. Furthermore, we will also implement similar solvers that run on the GPU and demonstrate that by utilizing the high throughput of modern GPUs we can further accelerate the simulations.

3.2. Computing Gradient And An Optimization Scheme For Path Synthesis. So far we have discussed ways to accelerate the solver, however, as we discussed in the background section there is great interest in gradient-based optimization methods for path synthesis as well. Moreover, the ForwardDiff.jl package in Julia allows us to easily obtain gradients and jacobians of our solver without any additional effort. This means that it is worthwhile to investigate gradient-based optimization using our fast solvers as the accelerated solver not only speeds up the forward simulation process it also enables that much faster gradient calculations for the solver, effectively accelerating both simulation and optimization at the same time. However, simply applying ForwardDiff.jl to the solver would not be doing this project justice as there are many challenges in the gradient-based optimization of planar linkage mechanisms that are currently unaddressed with the majority of the work in literature focusing on the path synthesis problem from the perspective of design space exploration through genetic algorithms, RL, or deep learning enhanced numerical atlas approaches. The main reason these existing methods do not use an optimization-based refinement in

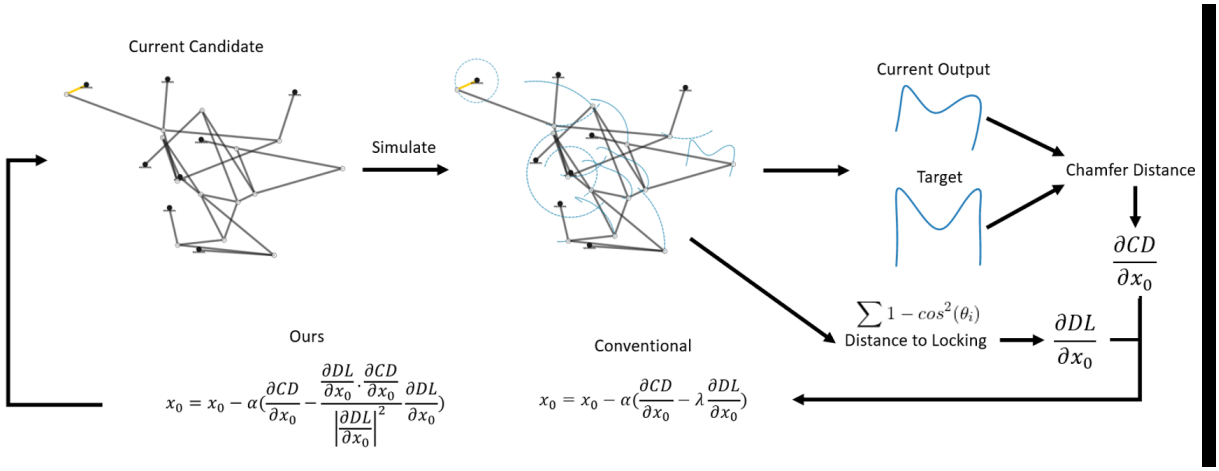

FIG. 4. *Here we illustrate the overall gradient decent approach and how we perform this compared to the conventional approach of applying a weighted penalty to the objective.*

their work is because of a few notable challenges which have yet to be addressed by researchers in the community. In this project, we aim to address three major challenges in gradient-based optimization for path synthesis which make the application of gradient-based optimization rather difficult. One of the challenges is slow solvers and gradient calculations which make the iterative process of optimization rather slow. This challenge is already addressed thanks to our significantly faster solver. However, two other challenges exist that we still need to address. These issues are the problem of picking a good objective function for comparing target paths and traced paths and the other is how to prevent mechanisms to fall into locking configurations during optimization. Both of these matters boil down to obtaining a reasonable objective function to allow us to match our target path as well as possible while still remaining in the feasible space. We will discuss the details of our implementation in the sections that follow.

3.3. Shape Matching Objective. The main objective of path synthesis is to generate mechanisms that trace a path that matches any arbitrary path with any arbitrary number of points. This means that if we simulate a mechanism for say 200 timesteps but our objective curve is a hand-drawn path with only a handful of points we cannot simply compare the points of the path generated by a mechanism and our target directly. Furthermore, the simulated curve has information regarding velocity, that is to say, if the mechanism is moving faster in a given part of the path the distance between the points will be larger, however, in path synthesis the only objective is to match a given target curve purely from a geometric perspective without any dynamic considerations. In fact, in practice, the dynamics can be controlled by adjusting the speed of the actuator as needed. As such, we need to come up with an objective function that unlike the work by Bächer *et al.* [3] does not rely on direct editing of coupler path already traced by a machine but rather is generalized to any arbitrary shape we wish to optimize for. This can be done using single-directional or bi-directional chamfer distance. The general form of the equation for chamfer distance between two sets of points S_1 and S_2 is as follows:

$$d_{CD}(S_1, S_2) = \sum_{x \in S_1} \min_{y \in S_2} \|x - y\|_2^2 + \sum_{y \in S_2} \min_{x \in S_1} \|x - y\|_2^2 \tag{3.3}$$

Where the first term measures the distance from S_1 to S_2 and the second term the other way around. As such this is usually referred to as the bi-directional chamfer distance. However, in path synthesis, we may be interested in only capturing the desired path as part of the mechanism's output curve. That is because even if the mechanism only traces the desired path in part of its motion the problem of synthesizing that path is solved. Therefore, in some instances, it may make sense to include only the term that measures the distance from the target curve to the traced path. However, for the results and discussions presented in this report, we will use the bi-directional chamfer distance and not make any further modifications to this objective function.

3.4. Preventing Locking In Mechanisms. As we saw in (3.1) the solution to any of the joints in the mechanisms requires the computation of an inverse cosine, and if a mechanism is set up such that during the motion of the actuator at some timestep the value inside the inverse cosine function is above 1 or below -1, the simulation fails, which indicates that the mechanism locks in at time between the last timestep and the timestep without a solution. This presents a challenge in gradient-based optimization as now a highly non-linear and complex constraint is added to the optimization problem which if ignored will easily render gradient-based optimization useless. Furthermore, the complexity of the constraint is beyond its non-linear nature, as the algorithm for the solution traverses a specific path through the graph representing the mechanisms. This means that if at one point in the path to the solution, a locking joint is identified the steps in the solution path after this point also cannot be solved for since they rely on the solution of prior joints in the path to be solved. This means that even if we can measure how much the cosine value of a given joint that locking is above 1 or below -1 we cannot say anything about how using gradients of this constraint violation for healing this issue will affect the solution of all the other joints. Therefore, a straightforward measure of constraint violation in cases of failure is not practical in this case. However, we may yet be able to perform gradient-based optimization by adding a penalty term to the objective of the optimization. Looking at prior works Bächer *et al.* [3] propose a penalty strategy by adding an objective to increase what the authors call distance to locking. This metric is measured by the following equation:

$$d_i(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) = 1 - \cos(\theta(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k))^2 \tag{3.4}$$

where $\cos(\theta(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k))$ is the value of the cosine of the angle calculated for joint i using the solutions of joints j and k using (3.1). To prevent this penalty from becoming too restrictive the authors use a cross-entropy penalty with a threshold ε and define the penalty objective as:

$$f_i(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) = -\frac{1}{2} \log^2 \left(\frac{1}{\varepsilon} d_i(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) \right) \tag{3.5}$$

by doing this they prevent the penalty from being too restrictive until the distance to locking dips below a specific threshold ε. However, this penalty will practically

encourage the optimization to specifically achieve a distance to locking equal to ε. And this is not exactly optimal. Furthermore, we still have to choose a fairly large ε. Bächer *et al.* [3] simply add the sum of this equation for all joints to their objective for minimization with some weight applied to it with respect to the main objective and solve for the minimal value of the sum of the two objectives in other words to minimize the following optimization problem:

$$\Delta\overline{X_0} = \arg\min_{\Delta X_0} d_{CD}(S_{target}, X_0) + \lambda \sum_i f_i(X_0) \tag{3.6}$$

Where $d_{CD}(S_{target}, X_0)$ is the chamfer distance between the path traced by a mechanism with initial joint positions X_0 and the target curve with points in the set S_{target} and $f_i(X_0)$ is the value of cross-entropy penalty function for joint i in the mechanism with initial joint positions X_0 and λ is the weight for the penalty objective and $\Delta\overline{X_0}$ is the optimal changes applied to the initial joint positions X_0 to optimize the mechanism for path synthesis. In the end, although this penalty approach makes the optimization process clear-cut, it still limits the path synthesis potential with optimization as it will always directly work against the main objective of reducing chamfer distance. As such we introduce an alternative to this by suggesting a different strategy for preventing locking, which is similar in nature to this approach but gives the optimization more freedom. We propose using simple gradient descent with a step size of α however instead of just optimizing for minimal chamfer distance we linearly orthogonalize the chamfer distance gradient with respect to distance to locking. By doing this we make an assumption that for small enough α the objective can be deemed linear, hence making it so that if we orthogonalize the gradient of chamfer distance with respect to distance to locking we move only in the direction that reduces chamfer distance without changing the distance to locking for the mechanism. This assumption of course in practice is not going to be accurate as we still have to choose a sufficiently large α to make the optimization practical, however by doing this we ensure that if the distance to locking is to increase during the optimization the rate at which it will happen during the gradient descent is minimal. Therefore, the update rule we propose for the gradient descent can be described as:

$$X_0^{k+1} = X_0^k - \alpha \left(\frac{\partial CD}{\partial X_0} - \frac{\frac{\partial DL}{\partial X_0} \cdot \frac{\partial CD}{\partial X_0}}{\left| \frac{\partial DL}{\partial X_0} \right|^2} \frac{\partial DL}{\partial X_0} \right) \tag{3.7}$$

Where $CD = d_{CD}(S_{target}, X_0)$ and $DL = \sum_i f_i(X_0)$ and X_0^k is the intial joint positions for the mechanism after k steps of gradient decent. We will later demonstrate how this approach is effective for refining mechanisms for path synthesis. The overall workflow of our approach is illustrated in Figure 4.

4. Results & Discussion. In this section, we run experiments on different variants of the solver and demonstrate how the approaches we have developed for accelerating the solver actually significantly improve the speed of the solver to a noticeable extent. Finally, we will demonstrate the gradient-based optimization results for a few examples to show the efficacy of the proposed approach for the refinement of mechanisms.

4.1. Acceleration Of The Solver. To measure how well each of the proposed implementations of the solver performs we conduct an experiment for simulating 10,000 mechanisms with 6 to 20 joints in them using each of the solvers described in prior sections. We solve all of these mechanisms for 200 timesteps and with the actuator moving at constant velocity. We also make a GPU implementation of the batch solver and test the simulation speeds on the GPU as well. As for hardware we use an Intel i9-12900K processor for the CPU versions of the solver and an RTX 3090Ti for the GPU implementations of the solver. Furthermore, it is important to note that we were unable to get the GPU version to work properly on Julia, as such we implemented the GPU versions on Python using Pytorch, and the results presented here for the GPU version of the solver are based on experiments on python. The results of our experiments are presented in Figure 5.

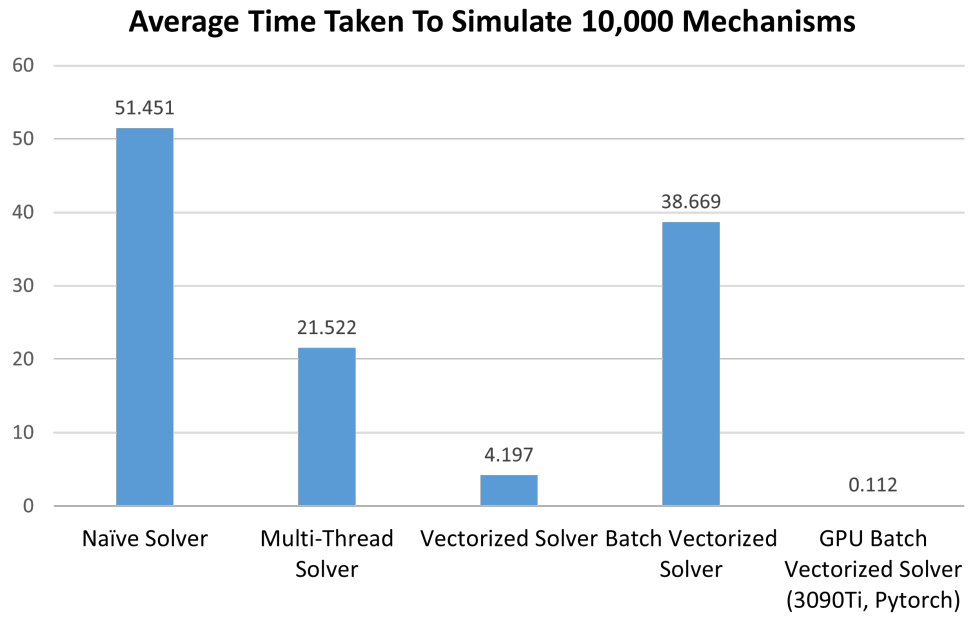

FIG. 5. *The results of running the solvers for 10,000 mechanisms. The time reported is the average across 10 runs of each solver.*

As evident in Figure 5 we see that the naive solver is the slowest solver. However, we can see that a simple approach of multi-threading does not really yield a significant improvement in the results and only accelerates the process by slightly more than double despite the i9-12900k's 24 threads. This is the main reason it is important for us to utilize the lower-level optimized linear algebra packages to maximize the performance of the solver. As evident the vectorized solver is more than 10 times faster than the naive solver and 5 times faster than the multi-threaded solver. Interestingly the batch-vectorized solver actually ends up being slower than the multi-threaded solver on the CPU. This is simply because we resize all of the mechanisms to the maximum size of 20 joints which essentially makes the overall solver much slower as the CPU simply does not have enough throughput to truly perform all of the computations at once despite the highly optimized linear algebra packages. However,

the same cannot be said of the GPU. As it is shown in Figure 5, the GPU can simulate all 10,000 mechanisms using the batch vectorized solver in roughly 100 milliseconds, which is a mindblowing 500 times faster than the naive solver and 50 times faster than the vectorized solver on the CPU. This is simply made possible by the immense throughput of the GPU and the highly optimized CUDA libraries that Pytorch uses in its backend. Furthermore, we see that despite the increased cost of the batch solver resizing all mechanisms to the maximum size the GPU has the necessary throughput to handle the larger number of computations and perform all the simulations in one go. One thing to note is that when we tested the GPU using just a timestep vectorized solver the results we got were much slower as each simulation had to be done on the GPU one at a time, despite the GPU having the capacity for much more, which despite of the better efficiency of the vectorized solver (as we saw in the CPU) led to slower results overall, demonstrating the importance of developing the batch vectorization for a high throughput hardware like the GPU. In conclusion, we see that utilizing the GPU can provide up to 500 times faster simulations and speed up most current approaches employed by researchers for path synthesis to a great extent.

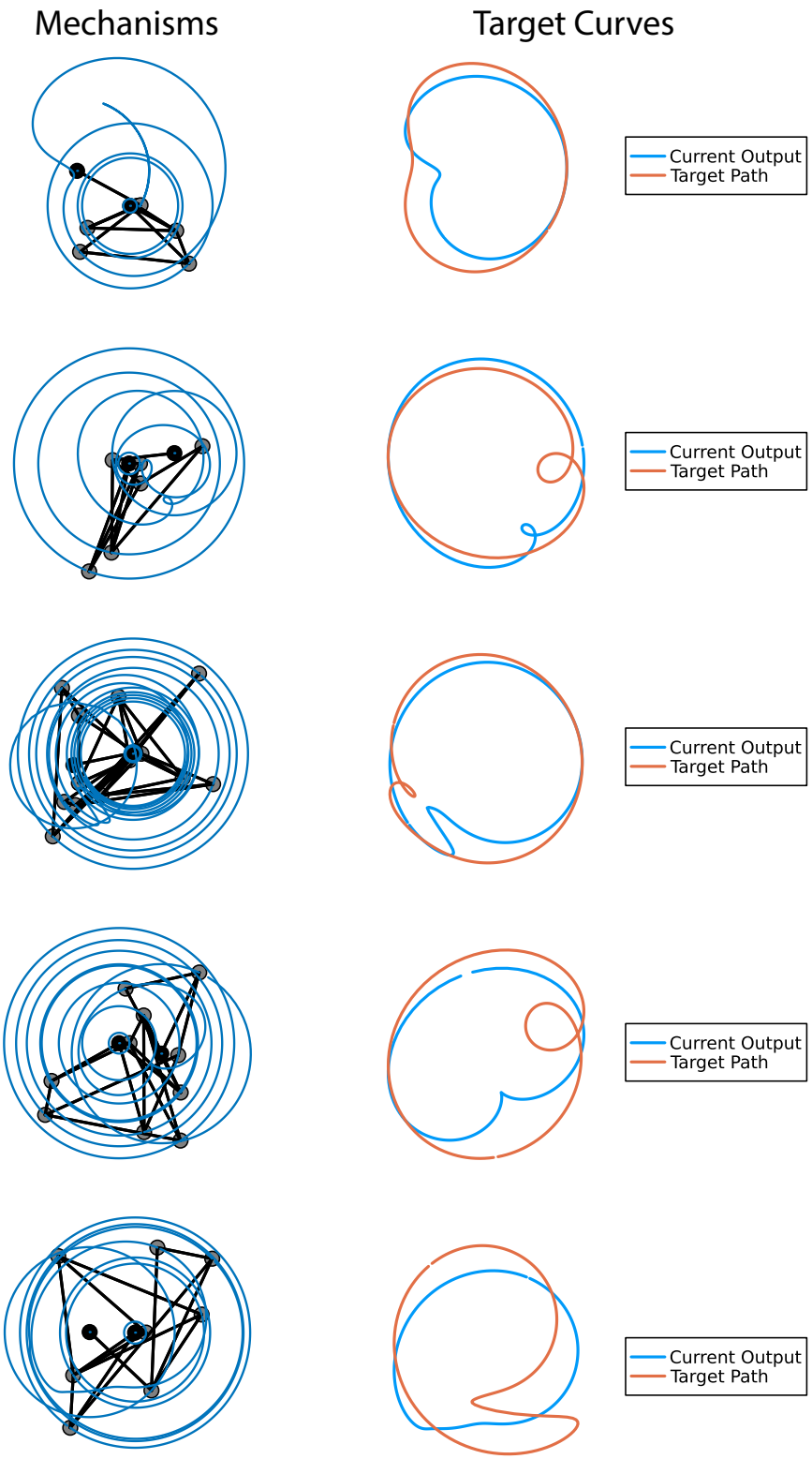

FIG. 6. *The 5 case studies for demonstrating the effectiveness of gradient-based optimization. Here we visualize the mechanism and their current output curves and the target curves we wish to achieve using these mechanisms through gradient-based optimization.*

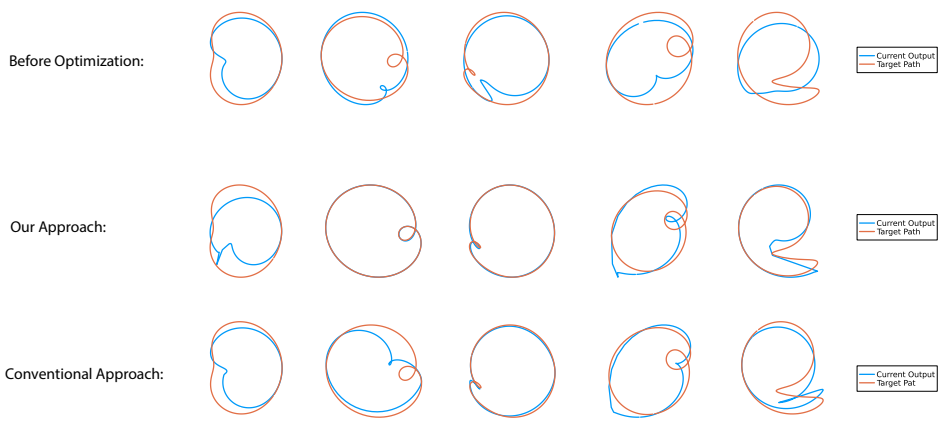

FIG. 7. *The results of gradient-based optimization on the 5 case studies demonstrating the effectiveness of gradient-based optimization. Here we see that our approach of orthogonalization instead of weighted penalty has led to much better results compared to conventional optimization.*

TABLE 1
Quantitave results (final chamfer distance) of the optimization comparing both methods is each case-study

Method	Study 1	Study 2	Study 3	Study 4	Study 5
Before Optimization	0.8155	0.5425	0.2955	0.5051	0.7207
Conventional	0.6307	0.3703	0.0215	0.3001	**0.1056**
Ours	**0.4948**	**0.0049**	**0.0065**	**0.2909**	0.1648

4.2. Experiments On Gradient-Based Optimization. To demonstrate the effects of the gradient-based optimization approach we propose we conduct a case study with 5 mechanisms and 5 target paths which for each mechanism. These mechanisms and the test targets for each case study can be seen in Figure 6. We then run gradient descent using our method and the naive method with a constant penalty with weight $lambda = 0.1$ (see (3.6)) for 10,000 steps in each case-study with a gradient decent step size of 0.0001 and compare the results visually in Figure 7 and report the minimum chamfer distance achieved by each method in Table 1.

The first thing we see visually is that in all cases the outputs of the mechanisms have improved significantly with the exception of the first case study where both conventional and our optimization have failed to improve the output of the mechanism. However, in all other cases, we can both visually (Figure 7) and quantitatively (Table 1) confirm the improvements. Specifically in the second, third, and last case studies, we see amazing improvements in the mechanisms matching their targets with great accuracy. However, as it is clear the orthogonalization has led to much better outcomes (the only exception is the last case study) with the mechanisms while the conventional approach has mostly failed to provide significant enough improvements in the mechanisms. This shows that unless the optimization is being done on a machine that traces a path that is already very close to the desired path the conventional approach is simply not good enough and does not provide a worthwhile refinement, which is likely why most researchers have not applied such methods. But

we showed that with our improved method, we are able to do much better and prove that refinement through optimization is a viable option for path synthesis.

5. Conclusion. In this project, we set out to address one of the major challenges that has slowed progress in inverse kinematics which is a lack of very fast solvers for linkage mechanisms. As we discussed the faster these solvers are the better existing methods for inverse kinematic synthesis can get. Given this, we developed very fast vectorized solvers for linkage mechanisms that can enable up to 500 times faster simulations when GPUs are utilized, which can improve the speed of existing optimization and data-driven approaches for inverse kinematic design. Furthermore, we took advantage of the ForwardDiff.jl package in Julia to develop an improved gradient-based optimization method that can outperform the existing methods for gradient-based optimization for path synthesis, and we demonstrated the efficacy of our method and its superiority to existing methods through a case study of 5 optimization problems. In conclusion, in this project, we were able to accelerate planar linkage mechanism simulations by up to 500 times while also improving gradient-based optimization methods for path synthesis using the automatic differentiation of our fast solvers. We hope that this contribution will help accelerate progress in the field of inverse kinematic design and path synthesis.

Finally, we provide the code used in this project publicly which can be found at https://github.com/ahnobari/18337-Linakge-Project.

Appendix A. Julia Code For Vectoized Solver. Below is the Julia code to traverse the path to the solution and compute all the timesteps at once.

```
function vectorized_solution(C, motor, x0, fixed_nodes, thetas = LinRange(0,2*pi,201)[1:200])
    p,o,f = find_path(C,motor,fixed_nodes)

    if f
        G = pairwise(Euclidean(),x0')


        x = repeat(reshape(x0,size(x0)[1],1,2),1, size(thetas)[1],1) * 0

        x[fixed_nodes,:,:] = repeat(reshape(x0[fixed_nodes,:], size(fixed_nodes)[1],1,2), 1, size(thetas)[1], 1)

        x[motor[2],:,:] = x[motor[1],:,:] + G[motor[1],motor[2]] * [cos.(thetas) sin.(thetas)]

        flag = true

        for step in p
            i = step[2]
            j = step[3]
            k = step[1]

            l_ij = norm.(eachcol((x[j,:,:]-x[i,:,:])'))

            cosphi = (l_ij .^ 2 .+ G[i,k]^2 .- G[j,k]^2)./(2 * l_ij * G[i,k])

            singularities = findall(x-> x>1.0 || x<-1.0, cosphi)

            cosphi[singularities] .= 0.0

            s = sign((x0[i,2]-x0[k,2])*(x0[i,1]-x0[j,1]) - (x0[i,2]-x0[j,2])*(x0[i,1]-x0[k,1]))

            phi = s * acos.(cosphi)

            R = reshape([[cos.(phi) sin.(phi)] [-sin.(phi) cos.(phi)]],size(thetas)[1],2,2)

            scaled_ij = (x[j,:,:]-x[i,:,:])./l_ij * G[i,k]

            x[k,:,:] = reshape(permutedims(permutedims(R,[2,3,1]) ⊠
                    permutedims(reshape(scaled_ij,size(thetas)[1],2,1),[2,3,1]),[3,1,2]),size(thetas)[1],2) + x[i,:,:]
            x[k,singularities,:] .= NaN
        end

        return permutedims(x,[2,1,3])
    else
        return zeros(size(x0)[1], size(thetas)[1],2).-1
    end

end
```

FIG. 8. *Vectorized solver to solve for all timesteps at once.*

Appendix B. Julia Code For Batch Vectoized Solver. Below is the code for preprocessing a list of mechanisms into a batch for the batch solver. This part includes the sorting and resizing of the batch.

```
function Fix_Order(C, x0, motor, fixed_nodes)
    path, order, f = find_path(C, motor, fixed_nodes)
    if f
        return C[order,:][:,order], x0[order,:], findall(x-> x in fixed_nodes, order)
    else
        return C, motor, fixed_nodes
    end
end

function Prepare_Batch(Cs,x0s,fixed_nodess, m_size = 20)
    Batch_Cs = zeros(size(Cs)[1],m_size,m_size)
    Batch_nt = zeros(size(Cs)[1],m_size)
    Batch_x0s = zeros(size(Cs)[1],m_size,2)
    Batch_Gs = zeros(size(Cs)[1],m_size,m_size)

    Threads.@threads for i in 1:size(Cs)[1]
        C,x0,fn = Fix_Order(Cs[i],x0s[i],[1,2],fixed_nodess[i])
        Batch_Cs[i,1:size(C)[1],1:size(C)[1]] = C
        Batch_x0s[i,1:size(C)[1],:] = x0
        Batch_nt[i,fn] .= 1
        Batch_nt[i,size(C)[1]+1:m_size] .= 1
        Batch_Gs[i,:,:] = pairwise(Euclidean(),Batch_x0s[i,:,:]')
    end

    return Batch_Cs,Batch_x0s,Batch_nt,Batch_Gs
end
```

FIG. 9. *Preprocessing code for the batch solver.*

Once the batches have been prepared the code below is used to solve not just all timesteps but all the mechanisms in the batch as well:

```
function batch_solve(Cs,x0s,Gs,node_types,thetas)

    x = zeros(size(x0s)[1],size(x0s)[2],size(thetas)[1],2)
    x = x .+ reshape(x0s .* node_types,size(x0s)[1],size(x0s)[2],1,2)

    x[:,2,:,:] = x[:,1,:,:] + repeat(reshape([cos.(thetas) sin.(thetas)],1,size(thetas)[1],2),size(x0s)[1],1,1) .* Gs[:,1,2]

    t = zeros(size(x0s)[1],size(x0s)[2],size(x0s)[2])
    t[:,:,[1,2]] .= 1

    t = t .* node_types

    Cs += t

    for k in 3:size(x0s)[2]
        inds = findall(x-> x>0 , Cs[:,k,1:k])
        x_vals = x[inds,:,:]

        xis = x_vals[1:size(x0s)[1],:,:]
        xjs = x_vals[size(x0s)[1]+1:2*size(x0s)[1],:,:]

        l_ijs = xis - xjs
        l_ijs = l_ijs .^ 2
        l_ijs = sum(l_ijs,dims=3)
        l_ijs = sqrt.(l_ijs)

        g_ij_k = Gs[inds,k]
        g_ik = g_ij_k[1:size(x0s)[1],:,:]
        g_jk = g_ij_k[size(x0s)[1]+1:2*size(x0s)[1],:,:]

        cosphis = (l_ijs.^2 .+ g_ik.^2 .- g_jk.^2)./(2 * l_ijs .* g_ik)

        singularities = findall(x->x>1.0 || x<-1.0, cosphis)

        cosphis[singularities] .= 0.0

        x0_val = x0s[inds,:]

        x0i1 = x0_val[1:size(x0s)[1],2]
        x0i0 = x0_val[1:size(x0s)[1],1]

        x0j1 = x0_val[size(x0s)[1]+1:2*size(x0s)[1],2]
        x0j0 = x0_val[size(x0s)[1]+1:2*size(x0s)[1],1]

        x0k1 = x0s[:,k,2]
        x0k0 = x0s[:,k,1]

        s = sign.((x0i1-x0k1).*(x0i0-x0j0) - (x0i1-x0j1).*(x0i0-x0k0))

        phi = s .* acos.(cosphis)

        R = reshape([[cos.(phi) sin.(phi)] [-sin.(phi) cos.(phi)]],size(x0s)[1],200,2,2)

        scaled_ij = (xjs-xis)./l_ijs .* g_ik

        x_k = xis + reshape(permutedims(permutedims(reshape(R,size(thetas)[1]*size(x0s)[1],2,2),[2,3,1]) ⊠
                permutedims(reshape(scaled_ij,size(thetas)[1]*size(x0s)[1],2,1),[2,3,1]),[3,1,2]),size(x0s)[1],size(thetas)[1],2)
        x_k = x_k - node_types[:,k] .* x_k

        x[:,k,:,:] += x_k

    end

    return permutedims(x,[1,3,2,4])
end
```

FIG. 10. *Vectorized solver to solve for all timesteps and mechanisms in the batch at once.*

REFERENCES

[1] *On the Extension of a Fourier Descriptor Based Method for Four-Bar Linkage Synthesis for Generation of Open and Closed Paths*, vol. Volume 2: 34th Annual Mechanisms and Robotics Conference, Parts A and B of International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, 08 2010, https://doi.org/10.1115/DETC2010-29028, https://doi.org/10.1115/DETC2010-29028, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/IDETC-CIE/proceedings-pdf/IDETC-CIE2010/44106/923/4549582/923_1.pdf.

[2] *Kinematic Synthesis Using Reinforcement Learning*, vol. Volume 2A: 44th Design Automation Conference of International Design Engineering Technical Confer-

ences and Computers and Information in Engineering Conference, 08 2018, https://doi.org/10.1115/DETC2018-85529, https://doi.org/10.1115/DETC2018-85529, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/IDETC-CIE/proceedings-pdf/IDETC-CIE2018/51753/V02AT03A009/2475681/v02at03a009-detc2018-85529.pdf. V02AT03A009.

[3] M. Bächer, S. Coros, and B. Thomaszewski, *Linkedit: Interactive linkage editing using symbolic kinematics*, ACM Trans. Graph., 34 (2015), https://doi.org/10.1145/2766985, https://doi-org.libproxy.mit.edu/10.1145/2766985.

[4] J. Cabrera, A. Simon, and M. Prado, *Optimal synthesis of mechanisms with genetic algorithms*, Mechanism and Machine Theory, 37 (2002), pp. 1165–1177, https://doi.org/https://doi.org/10.1016/S0094-114X(02)00051-4, https://www.sciencedirect.com/science/article/pii/S0094114X02000514.

[5] J. Chu and J. Sun, *A New Approach to Dimension Synthesis of Spatial Four-Bar Linkage Through Numerical Atlas Method*, Journal of Mechanisms and Robotics, 2 (2010), https://doi.org/10.1115/1.4001774, https://doi.org/10.1115/1.4001774, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/mechanismsrobotics/article-pdf/2/4/041004/5812581/041004_1.pdf. 041004.

[6] J. Chu and J. Sun, *Numerical atlas method for path generation of spherical four-bar mechanism*, Mechanism and Machine Theory, 45 (2010), pp. 867–879, https://doi.org/https://doi.org/10.1016/j.mechmachtheory.2009.12.005, https://www.sciencedirect.com/science/article/pii/S0094114X09002286.

[7] S. Deshpande and A. Purwar, *A Machine Learning Approach to Kinematic Synthesis of Defect-Free Planar Four-Bar Linkages*, Journal of Computing and Information Science in Engineering, 19 (2019), https://doi.org/10.1115/1.4042325, https://doi.org/10.1115/1.4042325, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/computingengineering/article-pdf/19/2/021004/5998446/jcise_019_02_021004.pdf. 021004.

[8] S. Deshpande and A. Purwar, *Computational Creativity Via Assisted Variational Synthesis of Mechanisms Using Deep Generative Models*, Journal of Mechanical Design, 141 (2019), https://doi.org/10.1115/1.4044396, https://doi.org/10.1115/1.4044396, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/mechanicaldesign/article-pdf/141/12/121402/5874716/md_141_12_121402.pdf. 121402.

[9] S. Deshpande and A. Purwar, *An Image-Based Approach to Variational Path Synthesis of Linkages*, Journal of Computing and Information Science in Engineering, 21 (2020), https://doi.org/10.1115/1.4048422, https://doi.org/10.1115/1.4048422, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/computingengineering/article-pdf/21/2/021005/6577132/jcise_21_2_021005.pdf. 021005.

[10] S. Deshpande and A. Purwar, *An image-based approach to variational path synthesis of linkages*, Journal of Computing and Information Science in Engineering, 21 (2021).

[11] S. Ebrahimi and P. Payvandy, *Efficient constrained synthesis of path generating four-bar mechanisms based on the heuristic optimization algorithms*, Mechanism and Machine Theory, 85 (2015), pp. 189–204, https://doi.org/https://doi.org/10.1016/j.mechmachtheory.2014.11.021, https://www.sciencedirect.com/science/article/pii/S0094114X14003036.

[12] M. B. Fogelson, C. Tucker, and J. Cagan, *GCP-HOLO: Generating High-Order Linkage Graphs for Path Synthesis*, Journal of Mechanical Design, 145 (2023), https://doi.org/10.1115/1.4062147, https://doi.org/10.1115/1.4062147, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/mechanicaldesign/article-pdf/145/7/073303/7001642/md_145_7_073303.pdf. 073303.

[13] N. Khan, I. Ullah, and M. Al-Grafi, *Dimensional synthesis of mechanical linkages using artificial neural networks and fourier descriptors*, Mechanical Sciences, 6 (2015), pp. 29–34, https://doi.org/10.5194/ms-6-29-2015, https://ms.copernicus.org/articles/6/29/2015/.

[14] N. Khan, I. Ullah, and M. Al-Grafi, *Dimensional synthesis of mechanical linkages using artificial neural networks and fourier descriptors*, Mechanical Sciences, 6 (2015), pp. 29–34.

[15] D. P. Kingma and M. Welling, *Auto-encoding variational bayes*, 2014, https://arxiv.org/abs/1312.6114.

[16] H. Lipson, *A Relaxation Method for Simulating the Kinematics of Compound Nonlinear Mechanisms*, Journal of Mechanical Design, 128 (2005), pp. 719–728, https://doi.org/10.1115/1.2198255, https://doi.org/10.1115/1.2198255, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/mechanicaldesign/article-pdf/128/4/719/5923963/719_1.pdf.

[17] H. Lipson, *Evolutionary synthesis of kinematic mechanisms*, Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 22 (2008), p. 195–205, https://doi.org/10.1017/S0890060408000139.

[18] J. M. McCarthy and G. S. Soh, *Geometric design of linkages*, vol. 11, Springer Science & Business Media, 2010.
[19] J. R. McGarva, *Rapid search and selection of path generating mechanisms from a library*, Mechanism and Machine Theory, 29 (1994), pp. 223–235, https://doi.org/https://doi.org/10.1016/0094-114X(94)90032-9, https://www.sciencedirect.com/science/article/pii/0094114X94900329.
[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013, https://arxiv.org/abs/1312.5602.
[21] P. Radhakrishnan and M. I. Campbell, *A graph grammar based scheme for generating and evaluating planar mechanisms*, in Design Computing and Cognition '10, J. S. Gero, ed., Dordrecht, 2011, Springer Netherlands, pp. 663–679.
[22] L. Regenwetter, A. H. Nobari, and F. Ahmed, *Deep generative models in engineering design: A review*, CoRR, abs/2110.10863 (2021), https://arxiv.org/abs/2110.10863, https://arxiv.org/abs/2110.10863.
[23] F. Reuleaux, *Lehrbuch der Kinematik*, vol. 1, Vieweg, 1875.
[24] S. Sharma and A. Purwar, *Using a Point-Line-Plane Representation for Unified Simulation of Planar and Spherical Mechanisms*, Journal of Computing and Information Science in Engineering, 20 (2020), https://doi.org/10.1115/1.4046817, https://doi.org/10.1115/1.4046817, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/computingengineering/article-pdf/20/6/061002/6537582/jcise_20_6_061002.pdf. 061002.
[25] K. Sohn, H. Lee, and X. Yan, *Learning structured output representation using deep conditional generative models*, in Advances in Neural Information Processing Systems, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds., vol. 28, Curran Associates, Inc., 2015, https://proceedings.neurips.cc/paper/2015/file/8d55a249e6baa5c06772297520da2051-Paper.pdf.
[26] J. Sun, H. Lu, and J. Chu, *Variable step-size numerical atlas method for path generation of spherical four-bar crank-slider mechanism*, Inverse Problems in Science and Engineering, 23 (2015), pp. 256–276, https://doi.org/10.1080/17415977.2014.890615, https://doi.org/10.1080/17415977.2014.890615, https://arxiv.org/abs/https://doi.org/10.1080/17415977.2014.890615.
[27] I. Ullah and S. Kota, *Optimal Synthesis of Mechanisms for Path Generation Using Fourier Descriptors and Global Search Methods*, Journal of Mechanical Design, 119 (1997), pp. 504–510, https://doi.org/10.1115/1.2826396, https://doi.org/10.1115/1.2826396, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/mechanicaldesign/article-pdf/119/4/504/5600257/504_1.pdf.
[28] S. Varedi-Koulaei and H. Rezagholizadeh, *Synthesis of the four-bar linkage as path generation by choosing the shape of the connecting rod*, Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science, 234 (2020), pp. 2643–2652, https://doi.org/10.1177/0954406220908616, https://doi.org/10.1177/0954406220908616, https://arxiv.org/abs/https://doi.org/10.1177/0954406220908616.
[29] A. Vasiliu and B. Yannou, *Dimensional synthesis of planar mechanisms using neural networks: Application to path generator linkages*, Mechanism and Machine Theory, 36 (2001), pp. 299–310, https://doi.org/10.1016/S0094-114X(00)00037-9.
[30] K. J. Waldron and S. V. Sreenivasan, *A Study of the Solvability of the Position Problem for Multi-Circuit Mechanisms by Way of Example of the Double Butterfly Linkage*, Journal of Mechanical Design, 118 (1996), pp. 390–395, https://doi.org/10.1115/1.2826898, https://doi.org/10.1115/1.2826898, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/mechanicaldesign/article-pdf/118/3/390/5747759/390_1.pdf.
[31] J. Wu, Q. J. Ge, F. Gao, and W. Z. Guo, *On the Extension of a Fourier Descriptor Based Method for Planar Four-Bar Linkage Synthesis for Generation of Open and Closed Paths*, Journal of Mechanisms and Robotics, 3 (2011), https://doi.org/10.1115/1.4004227, https://doi.org/10.1115/1.4004227, https://arxiv.org/abs/https://asmedigitalcollection.asme.org/mechanismsrobotics/article-pdf/3/3/031002/5590355/031002_1.pdf. 031002.