A framework for structural shape optimization based on automatic differentiation, the adjoint method and accelerated linear algebra

Gaoyuan Wu

Received: 29 November 2022 / Revised: 16 May 2023 / Accepted: 24 May 2023 / Published online: 20 June 2023
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2023

Abstract
Shape optimization is of great significance in structural engineering, as an efficient geometry leads to better performance of structures. However, the application of gradient-based shape optimization for form finding is limited, which is partly due to the difficulty and the complexity in gradient evaluation. In this work, an efficient framework based on automatic differentiation (AD), the adjoint method and accelerated linear algebra (XLA) is proposed to promote the implementation of gradient-based shape optimization for form finding. The framework is realized by the implementation of the high-performance computing (HPC) library JAX. We leverage AD and the adjoint method in the sensitivity analysis stage. The XLA feature is exploited by an efficient programming architecture that we proposed, which can boost gradient evaluation via just-in-time compilation of computer programs. The proposed framework also supports hardware acceleration such as GPUs. The framework is applied to the form finding of arches and different free-form gridshells: gridshell inspired by Mannheim Multihalle, four-point supported gridshell, and canopy-like structures. Two geometric descriptive methods are used: non-parametric and parametric description via Bézier surface. Unconstrained and constrained shape optimization problems are considered, where the former is solved by gradient descent and the latter is solved by sequential quadratic programming (SQP). Through these examples, the proposed framework is shown to be able to provide structural engineers with an efficient tool for form finding, enabling better design for the built environment.

Keywords Shape optimization · Form finding · Automatic differentiation · Adjoint method · JAX · Shell structure · Bézier surface

1 Introduction

Form finding has always been an important topic in architectural and structural design, especially in designing lightweight structures such as concrete shells and gridshells, as it finds an optimal structural shape that makes efficient use of material (Bletzinger et al. 2005; Adriaenssens et al. 2014). The optimal use of material is accomplished when the structural elements are subjected to membrane forces (tension and compression) other than bending (Bletzinger and Ramm 2001) and thus, the objective is to find a structural geometry that minimizes the bending behavior. There are various computational approaches for form finding, including Force Density Method (FDM) (Schek 1974; Cuvilliers 2020), Dynamic Relaxation (DR) (Barnes 1999) and Particle-Spring Systems Method (Kilian and Ochsendorf 2005), where a state of static equilibrium is found for form-active structures. Despite the popularity, these form finding approaches only output an equilibrium-state and a geometry, but the results are only preliminary and do not inform any actual structural response.

In contrast, the more generalized and rigorous methodology, based on optimization theory and finite element analysis (FEA), can be implemented to preserve the coherence between structural analysis and form finding. In this approach, a shape optimization problem is explicitly formulated and solved by gradient-based optimization algorithms (Bletzinger and Ramm 2001). The objective function to minimize is the total strain energy of the structural system, as minimizing the strain energy is equivalent to minimizing the bending (Bletzinger and Ramm 2001). Gradient-based
shape optimization is efficient, but calculating the gradients of the objective function and constraints with respect to the design variables, i.e., sensitivity analysis, is a challenging task. Some work offers insights on how the sensitivities are calculated. Uysal et al. (2007) implemented finite difference to calculate the sensitivity to obtain the optimum shape for shell structures. Rombouts et al. (2019) derived the analytical sensitivities of a shape optimization problem for gridshells and compared the analytical results to numerical results. Xia et al. (2021) calculated the sensitivity using central difference for a shape optimization subproblem and proposed a strut-and-tie model as an analogy for reinforced concrete. San et al. (2021) optimized free-form concrete shells considering material damage based on numerical calculations of sensitivities and they pointed out that the material nonlinearity makes analytical calculation hard to implement. Despite extensive work using analytical or numerical sensitivities, challenges still remain for their implementation. First, the manual analytical derivation is inefficient and error-prone. Moreover, the complexity of derivation process makes it hard to be implemented by engineers whose expertise is in structural design, not mathematics. Additionally, the derived expression is only applicable to a specific type of structural element so one has to derive different sensitivity expressions for different element types, which will be extremely time-consuming. As for numerical calculation, it suffers from truncation and round-off errors (Chandrasekhar et al. 2021).

Alternatively, automatic differentiation (AD) can be leveraged (Griewank et al. 1989; Griewank and Walther 2008). Functions expressed as computer programs, no matter how complicated, can be expressed as a sequence of basic operations and by applying the chain rule from calculus, AD can obtain the derivative value of the function “automatically” to the working precision of the computer. Compared to analytical derivatives that may require tremendous implementation efforts (Van Keulen et al. 2005) associated with the derivation process, the implementation effort of AD is much less; compared to numerical derivatives that suffer from truncation and round-off errors, AD is more accurate. AD has been widely used in different fields, such as computational fluid dynamics (Bezgin et al. 2023), molecular dynamics (Schoenholz and Cubuk 2020), and machine learning (Baydin et al. 2018). In structural optimization, the application of AD can be dated back to the work by Ozaki et al. (1995), in which an AD framework based on FORTRAN was implemented to optimize mechanical structures. Espath et al. (2011) implemented AD to optimize shell structures based on NURBS (non-uniform rational B-spline) description. Nørgaard et al. (2017) introduced the applications of AD in topology optimization. Chandrasekhar et al. (2021) proposed a framework of using AD for topology optimization. Pastrana et al. (2022) implemented AD for constrained form finding problems using the Combinatorial Equilibrium Modeling (CEM) method. Despite the efforts, the application of AD in shape optimization is still rarely seen for form finding purposes, which is partly due to the complexity and the high computational cost of the AD packages previously implemented, as pointed out by Van Keulen et al. (2005).

In other words, the power of modern AD software library and hardware acceleration has not been exploited in structural engineering to speed up structural shape optimization. Moreover, very few work proposed framework for shape optimization using AD, and the existing work does not offer insights on how the structure of computer programs should be configured and optimized. For instance, Espath et al. (2011) directly applied an AD program called TAPENADE AD (Hascoet and Pascual 2013) for shape optimization but the time scale of AD implementation and the structure of the computer program were not shown.

To fill the research gaps identified, a framework for structural shape optimization based on AD and accelerated linear algebra (XLA) is proposed herein. The framework is called JAX-SSO, as it is achieved by the implementation of a high-performance computing (HPC) library in Python called JAX (Bradbury et al. 2018). The complete codes of the framework are available at https://github.com/GaoyuanWu/JAX-SSO. The proposed framework is intended to facilitate the employment of gradient-based structural shape optimization for form finding. To obtain the sensitivity of the objective function, the adjoint method (Tortorelli and Michaleris 1994) and AD are implemented. The computation in JAX-SSO is accelerated by a compact programming structure that we proposed. Moreover, JAX-SSO also supports accelerators like GPUs to speed up the derivative calculation process, enabling researchers to deal with problems with high complexity. Recently, JAX has been implemented for different optimization problems: Paganini and Wechsung (2021) implemented it for shape optimization problems constrained to partial differential equations (PDEs); Chandrasekhar et al. (2021) developed a framework for topology optimization using JAX. However, to the best of the authors’ knowledge, JAX has not been exploited by structural engineers for gradient-based shape optimization for form finding.

The goal of this paper is to introduce an efficient framework built on AD, XLA and the adjoint method to facilitate the implementation of gradient-based shape optimization for form finding. The remainder of this paper is organized as follows. In Sect. 2, the structural shape optimization problem is formulated, followed by a brief introduction of the adjoint method and AD. The features of JAX, the proposed framework and its performance are introduced in Sect. 3. In Sect. 4, the framework is validated and applied to several examples, including the form finding of arches and free-form shells with different boundary conditions. Lastly, the
capability and the limitations of the framework are concluded in Sect. 5.

2 Problem formulation

In structural shape optimization, some widely used objective functions to minimize are: (i) the total strain energy of the system, i.e., the compliance; (ii) the maximum Von Mises stress (Ding 1986; iii) the volume displacement as defined in Robles and Ortega (2001); and (iv) the stress leveling as defined in Bletzinger and Ramm (1993). Here we only consider the minimization of the strain energy as the scope of this work is limited to form finding. In addition, structural system with two-node beam elements in 3D space will be considered. Each structural node has 3 translational and 3 rotational degrees of freedom. The structural shape optimization problem is then formulated as follows:

minimize $C(x) = \frac{1}{2} \int \sigma e dV = \frac{1}{2} f^T u(x)$ (1a)

subject to: $K(x)u(x) = f$ (1b)

$h_j(x) \leq 0, \ j \in \{1, ..., n_e\}$ (1c)

g_j(x) = 0, \ j \in \{1, ..., n_c\}$ (1d)

where $C$ is the compliance, i.e., the strain energy, which is equal to the work done by the external load; $x \in \mathbb{R}^{n_v}$ is a vector of $n_v$ design variables; $\sigma$, $e$ and $V$ are the stress, strain, and volume, respectively; $f \in \mathbb{R}^{6n}$ and $u(x) \in \mathbb{R}^{6n}$ are the generalized load vector and nodal displacement of $n$ structural nodes; $K \in \mathbb{R}^{6n \times 6n}$ is the stiffness matrix; $h_j(x)$ is the $j$-th inequality constraint out of $n_e$; $g_j(x)$ is the $j$-th equality constraint out of $n_c$. The design variables $x$ determine the geometric form of the structure, which can be (i) non-parametric: the design variables are the nodal coordinates of every design node in the system; (ii) parametric: the design variables are the parameters or nodal coordinates of control points in a geometry representation function such as Bézier surface or Non-uniform Rational B-spline (NURBS). In this study, both geometry descriptive methods are considered.

In terms of the constraints, the first constraint (Eq. 1b) is necessary since it is the governing equation for FEA that any structural system should obey, whereas the other constraints (Eq. 1c-1d) are optional. In the remainder of this work, we refer the optimization problem without the optional constraints as unconstrained shape optimization and the problem with optional constraints as constrained shape optimization. In this work, the unconstrained shape optimization problems will be solved by gradient descent; the constrained shape optimization problems will be solved by sequential least-square programming (SLSQP) (Kraft 1988), a method that belongs to sequential quadratic programming (SQP) family (Gill and Wong 2012).

2.1 The adjoint sensitivity analysis

For gradient-based optimization, calculating the gradient of the objective function $C(x)$ with respect to the design variables $x$, $\nabla C(x)$, is of great significance. This process is also known as sensitivity analysis and $\nabla C(x)$ is usually referred as the sensitivity of the optimization problem. Two approaches can be implemented for sensitivity evaluation: the direct method and the adjoint method. The adjoint method is preferable over the direct method when the number of objective functions is smaller than the number of design parameters (Tortorelli and Michaleris 1994) and thus we adopt the adjoint method herein.

Firstly, we differentiate the objective function with respect to the $i$-th design variable:

$$\frac{\partial C}{\partial x_i} = \frac{1}{2} \frac{\partial f^T}{\partial x_i} u + \frac{1}{2} f^T \frac{\partial u}{\partial x_i}$$ (2)

Then we differentiate the FEA equation (Eq. 1b):

$$\frac{\partial K}{\partial x_i} u + K \frac{\partial u}{\partial x_i} = \frac{\partial f}{\partial x_i}$$ (3a)

$$K \frac{\partial u}{\partial x_i} = \frac{\partial f}{\partial x_i} - \frac{\partial K}{\partial x_i} u$$ (3b)

$$\frac{\partial u}{\partial x_i} = K^{-1} \left( \frac{\partial f}{\partial x_i} - \frac{\partial K}{\partial x_i} u \right)$$ (3c)

We then plug Eq. 3c into Eq. 2:

$$\frac{\partial C}{\partial x_i} = \frac{1}{2} \frac{\partial f^T}{\partial x_i} u + \frac{1}{2} f^T K^{-1} \left[ \frac{\partial f}{\partial x_i} - \frac{\partial K}{\partial x_i} u \right]$$ (4)

Since $f^T = u^T K^T$ and the stiffness matrix is symmetric, we have:

$$\frac{\partial C}{\partial x_i} = \frac{1}{2} \frac{\partial f^T}{\partial x_i} u + \frac{1}{2} u^T K^{-1} \left[ \frac{\partial f}{\partial x_i} - \frac{\partial K}{\partial x_i} u \right]$$ (5a)

$$\frac{\partial C}{\partial x_i} = \frac{1}{2} \frac{\partial f^T}{\partial x_i} u + \frac{1}{2} u^T K^{-1} \left[ \frac{\partial f}{\partial x_i} - \frac{\partial K}{\partial x_i} u \right]$$ (5b)

$$\frac{\partial C}{\partial x_i} = \frac{1}{2} \frac{\partial f^T}{\partial x_i} u + \frac{1}{2} u^T \left[ \frac{\partial f}{\partial x_i} - \frac{\partial K}{\partial x_i} u \right]$$ (5c)
Here we consider constant loading and with that assumption, the sensitivity is further simplified to the following:

\[
\frac{\partial C}{\partial x_i} = -\frac{1}{2} u^T \frac{\partial K}{\partial x_i} u
\]

(6)

With the adjoint method, the sensitivity analysis can then be divided into two tasks: (i) calculate the derivatives of the global stiffness with respect to the design variables, \( \frac{\partial K}{\partial x} \); and (ii) conduct FEA to obtain the displacement vector \( u \).

### 2.2 The optimization procedure

The steps to obtain the form-found structures are shown in Fig. 1. Firstly, an initial geometry is input by choosing the initial design parameters \( x^0 \). In addition, constraints can be added to the optimization problem and the gradient-based optimizer is then selected together with the stopping criteria. The stopping criteria can be the maximum iteration number, the expected strain energy, the absolute or relative change in the strain energy between iterations, etc. The form finding process then starts with conducting the sensitivity analysis, i.e., obtaining \( \nabla C(x^k) \). The next step is to update the design variables at each iteration step \( k \):

\[
x^{k+1} = x^k + d^k
\]

(7)

where \( d^k \) is the search direction. If the shape optimization problem is not constrained, \( d^k \) is output by gradient descent and is simply the product between a step size (learning rate) \( \tau^k \) and the negative gradient: \( d^k = -\tau^k \cdot \nabla C(x^k) \). If constraints are imposed, SLSQP will output the search direction \( d^k \). SLSQP is an algorithm belonging to the SQP family. The details of SLSQP or SQP will not be elaborated in this paper and interested readers are referred to the work by Kraft (1988); Gill and Wong (2012). The name of SQP comes from the fact that at each iteration, an approximation of the original problem is formulated in the quadratic programming (QP) form as a sub-problem and it is solved to get the search direction \( d^k \). By iteratively solving the approximated original problem in QP forms, the final solution will converge to that of the original problem (Boggs and Tolle 1995). SLSQP takes the gradients of the objective function and the constraints as inputs: \( \nabla C(x^k), \nabla h_j(x^k), \) and \( \nabla g_j(x^k) \), and outputs the search direction \( d^k \). After the update of the design variables, the optimization goes to the next iteration until the stopping criteria is met.

The primary challenge in the optimization process is evaluating the necessary derivatives, especially the sensitivity. In our work, we will calculate all derivatives using automatic differentiation (AD). Here, we will only provide a brief overview of AD, and interested readers are referred to Baydin et al. (2018) for details. AD is an efficient tool for derivative computation of functions expressed as computer programs. AD exploits the fact that any derivative calculation can be decomposed as a set of basic derivative calculations so that one can trace these basic calculations sequentially to get the overall derivative by applying the chain rule (Baydin et al. 2018). AD gives

### Fig. 1 Flowchart of form finding with gradient-based shape optimization

![Flowchart of form finding with gradient-based shape optimization](image-url)
the exact derivative value to the working precision of the computer so it is not numerical differentiation. It is also different from manual derivation or symbolic differentiation because it does not output a mathematical expression of the derivative.

### 3 The proposed framework

#### 3.1 Introduction to JAX

JAX (Bradbury et al. 2018), developed by Google, is a Python library that aims at providing AD and high-performance computing to the research community. JAX can automatically differentiate Python and NumPy (Harris et al. 2020) functions, making it an ideal candidate for calculating derivatives in structural shape optimization. In addition, JAX supports just-in-time (JIT) compilation (Lam et al. 2015) via accelerated linear algebra (XLA), a compiler originally used in TensorFlow (Abadi et al. 2020), to improve the speed and memory usage of computer programs. The AD and XLA features make JAX a good candidate for gradient evaluation in gradient-based structural shape optimization.

We first introduce some key methods in JAX (Table 1). In array programming (Harris et al. 2020), numpy has been widely used. JAX has a similar array programming package called jax.numpy, which can be used to store values such as structural stiffness matrix using jax.numpy.array. The main difference between jax.numpy and numpy is that values and operations in jax.numpy can be traceable, assisting the implementation of AD using the chain rule.

To speed up sequences of operations such as the sensitivity analysis, jax.jit or @jit can be implemented. High level languages, for instance, Python codes, can be easily read by humans. However, the machine can not directly read the high level languages, so a compiler or an interpreter is needed to convert the high level languages to binary machine languages. The main difference between a compiler and an interpreter is that the compilation is done before the programs run while the interpreter converts the high level languages, line by line, during the programs run. As a result, the compiled codes are much faster than the interpreted codes. Standard Python codes use interpreter and thus, it is rather slow. In response, the implementation of jax.jit is able to use the XLA compiler to convert a sequence of high-level Python operations into optimized machine binary codes that operate faster once they have been compiled (Chandrasekhar et al. 2021). For operations that will be repeated during the shape optimization process like derivative evaluation, the use of jax.jit will greatly reduce the computational cost.

In addition to JAX’s JIT compiler via XLA, vectorization can also significantly reduce the computational cost of computer programs. Vectorization involves parallel processing of code, as opposed to for-loops. There are two primary reasons to use vectorization. First, vectorized programs are faster than those written with for-loops. For example, assembling the global stiffness matrix $K$ involves adding the values of the local stiffness matrix of each structural element to the global stiffness matrix. With vectorization, all the structural elements can be evaluated simultaneously, resulting in faster computation (Harris et al. 2020). With for-loops, on the other hand, only one structural element can be evaluated at a time, making the assembly process much slower. Second, vectorized programs are preferable to the XLA compiler as they reduce compilation time. When compiling Python code, the compiler processes operations sequentially before compiling them into machine code that can be executed simultaneously. Writing code with for-loops can significantly increase the compilation time, negating the benefits of the compiled code. JAX provides a method called jax.vmap that can vectorize functions written for one element into functions that can operate on multiple elements simultaneously.

To calculate the gradient of functions with AD, jax.jacfwd can be used. In order to implement JAX methods like jax.vmap for Python objects of any customized Python class, @register_pytree_node_class decorator can be used to register the customized Python class as a valid JAX type. In addition, tree_flatten and tree_unflatten methods are needed to tell JAX how to

| Table 1 Key methods in JAX |
|---------------------------|
| JAX methods | Description |
| jax.numpy | Array programming package in JAX, its syntax is similar to numpy |
| jax.jit/@jit | JIT feature that compiles operations written in Python functions to optimized machine codes with XLA, to speed up the operations |
| jax.vmap | Method to vectorize the operations for faster implementation |
| jax.jacfwd | Method to calculate the Jacobian |
| @register_pytree_node_class | Register a new class as a valid type in JAX |
flattened and unflattened the attributes of customized objects. A toy-code example is attached in Appendix 1 to illustrate how the aforementioned JAX methods work by presenting a simple example: calculating how the length of a 2D line changes with the nodal coordinates.

### 3.2 The JAX-SSO Framework

As mentioned in Sect. 2, the main challenge is the efficient evaluation of the sensitivity. Here we propose JAX-SSO, an efficient code structure that tackles the aforementioned challenge. In this subsection, the underlying mathematical background and the code structure of JAX-SSO will be illustrated. Here we use 3D beam-columns as a proof-of-concept but the proposed framework can be directly transferable to structures with other structural elements, such as springs, plates, shells and solids, thanks to the implementation of AD. Without loss of generality, the design variables we consider in this subsection will be the ground and the code structure of JAX-SSO will be illustrated.

The goal of JAX-SSO is to output the adjoint sensitivity as shown in Eq. 6. Firstly, FEA is conducted to obtain the sensitivity, 

$$s_{e} = \frac{\partial C}{\partial x}$$

in the system. The sensitivity, 

$$s_{e} = \frac{\partial C}{\partial x}$$

of a structural element can then be obtained.

The JAX-SSO framework is based on the mathematical foundations described above. However, unlike Fig. 2 where the evaluation of each structural element is conducted

![Fig. 2](image_url)
sequentially, JAX-SSO evaluates all structural elements simultaneously through the implementation of vectorization using \texttt{vmap}. The code structure of JAX-SSO is illustrated in Fig. 3, which is similar to FEA solvers: it has different modules for the nodes, the elements (beam-columns) and the model.

The \texttt{Node.py} module in Fig. 3 contains a class called \texttt{Node()}, which is used to define structural nodes. Each node is uniquely identified by a tag and defined by its nodal coordinates.

The \texttt{BeamCol.py} module in Fig. 3 includes a class called \texttt{BeamCol()} for beam-column elements, as well as functions for calculating $\frac{dk}{dx}$. To create a \texttt{BeamCol()} object, attributes such as the tag of this element, the cross-sectional properties, and the connectivity are needed. The object’s methods include calculating the element’s length, the coordinate transformation matrix, the stiffness matrix in the local coordinate system, and the stiffness matrix in the global coordinate system. In \texttt{BeamCol.py}, a function called \texttt{Ele_Sens_K_Coord} is defined for \texttt{BeamCol()} objects (see Appendix B, Code 4), which returns the derivatives of the element’s local stiffness matrix with respect to its coordinates, $\frac{dk}{dx}$, using AD. Additionally, the \texttt{Ele_Sens_K_Coord} function is JIT-compiled into optimized machine code (see Appendix B, Code 4).

The \texttt{Model_Sens.py} module is the primary interface of the JAX-SSO package for user interaction. A few lines of code on the user’s end are sufficient to return the sensitivity required for the optimization process, and the steps can be viewed in Fig. 4. A JAX-SSO model is firstly built by initializing an object from class \texttt{Model_Sens()} (Code 1). Nodes, structural elements, loads, and boundary conditions can then be added to the model (Code 1). After having configured the model, FEA can be performed to obtain the displacement $u$ (Code 2). Later, one can simply use a one-liner to obtain the sensitivity term $\nabla C(x)$ by calling the \texttt{Sens_C_Coord()} function (Code 2). Under the hood, JAX-SSO leverages the benefits of vectorization through the use of \texttt{vmap} and XLA-enabled \texttt{jit} compilation, which results in significant speed-up of the computations, including both the FEA and the computation of the sensitivity.

```python
import JaxSSO as sso # JAX-SSO module

model = sso.Model_Sens.Model_Sens() # JAX-SSO model
model.add_node(nodeTag, x, y, z) # add an node
model.beamcol(i, j_node, k_node, E, G ,Iy, Iz, J, A) # add a beam-column element
model.add_nodal_load(f) # add load
model.add_support(i ,[1,1,1,0,1]) # add boundary condition

Code 1 Creating a sensitivity model, adding nodes and elements in JAX-SSO
```
The structure of the FEA functionality in JAX-SSO can be seen in Fig. 4. The construction of the global stiffness matrix $K$ is optimized in a vectorized approach: the calculation of the local stiffness matrix $k_e$ and the assembly of $K$ from $k_e$ are done simultaneously for all structural elements through the implementation of vmap. Then, this function is compiled by @jit into optimized machine codes. In addition, as the global stiffness matrix $K$ is sparse, it is stored in the compressed sparse row (CSR) format herein to save the memory and computational cost. Finally, the FEA is conducted by solving the sparse linear system via scipy.sparse.linalg.spsolve to get $u$.

The structure under the Sens_C_Coord() function for the sensitivity calculation is also illustrated in Fig. 4. We first create one BeamCol() object that stores the attributes of all the beam-columns in the system. The Ele_Sens_K_Coord function in BeamCol.py is able to calculate $\frac{dk_e}{dx}$ for each element and it is further “jitted” into optimized machine codes for better performance. Later, the “jitted” Ele_Sens_K_Coord function is vectorized by vmap so it can be applied to the BeamCol() we created that integrates the attributes of all the structural elements. This then enables the calculation of $\frac{dk}{dx}$ for all structural elements simultaneously. With the displacement vector $u$ obtained by FEA and the $\frac{dk}{dx}$ for all elements, the sensitivity of the strain energy contributed by each element can be calculated by Eq. 8. To this end, a function called dcdx_bamcol_single() is firstly created for the calculation of $\frac{dc}{dx}$ for one structural element and its addition to the global sensitivity vector according to Eqs. 8 and 9. Later, the function dcdx_bamcol_single() is further “vmapped” and “jitted” into dcdx_bamcol so that the calculation and assembly of the sensitivity $\frac{dc}{dx}$ is done simultaneously for all elements via optimized machine codes. This efficient and compact structure of Sens_C_Coord() is of great significance in the proposed framework, as it realizes accurate and fast calculation of $\frac{dc}{dx}$ using AD.

The JAX-SSO framework is efficient and novel in a sense that it uses a combination of mathematical derivations via the adjoint method and JAX-boosted AD. Instead of calculating $\frac{dc}{dx}$ directly using AD, we decompose it into few tasks:
FEA, the calculation of $\frac{dC}{dx}$, the calculation of $\frac{dC_e}{dx_e}$ and the assembly of $K$. The main advantages are as follows. First, the calculation of $\frac{dC}{dx}$, $\frac{dC_e}{dx_e}$ and the assembly of $C$ can all be decomposed into independent element-wise operations, which can then be boosted by vectorization. In addition, the jit compilation via XLA can be applied to these element-wise operations to further improve the computational speed.

The proposed framework is also flexible because new element types can be added to JAX-SSO to support the optimization of new structural systems without carrying out mathematical derivations as AD calculates the derivatives “automatically”. Lastly, all the operations can be further boosted by GPU acceleration. The GPU computing is automatically enabled if the GPU device is detected.

### 3.3 Performance of JAX-SSO

We now illustrate the performance of JAX-SSO by presenting its speed of calculating $\frac{dC}{dx}$ for structural systems with different numbers of structural elements (Fig. 5). The number of structural elements vary from $2^3$ (8) to $2^{13}$ (8192).

Three different code structures are compared: (i) using for-loops for the calculation of $\frac{dC}{dx}$ for each structural element, the assembly of global stiffness matrix $K$ in FEA, and the assembly of the sensitivity $\frac{dC}{dx}$ from $\frac{dC_e}{dx_e}$; (ii) using vmap to vectorize the operations so that the calculation and assembly are done simultaneously; and (iii) using jit to further compile the vectorized codes into optimized machine binary codes using XLA, which is the structure adopted by JAX-SSO. Furthermore, we use different hardware to showcase the speed difference brought by GPU acceleration: (i) CPU: 2.30GHz Intel i7–10,875 H CPU operated on a personal laptop, (ii) CPU+GPU: the CPU plus NVIDIA GeForce RTX 3080 Laptop GPU with the video memory of 8GB on the same personal laptop.

As can be seen in Fig. 5a, the performance of the code structure used by JAX-SSO (with vmap and jit) outperforms other codes significantly. With the for-loop code, the computational cost increases linearly with the number of elements: on CPU, the time grows from 4.89 s for a system with 8 elements to 3182.42 s for a system with 8192 elements; on GPU, the time increases from 4.84 to 5072.25 s. This is extremely slow and not ideal. By replacing the explicit loops with vectorized codes using vmap, the computational cost is greatly reduced. For instance, on GPU, the computational time is reduced from 5072.25 s (for-loops) to 1.34 s by the implementation of vmap for the sensitivity analysis of a structural system with 8192 elements, which is around 3800 times faster; on CPU, the time is reduced from 3182.42 to 2.54 s, which is about 1250 times faster. In addition, with vectorization, the computational time remains almost a constant for a range of element numbers: on CPU, the computational time of a system with 256 elements is around the same as that of a system with 8 elements; on GPU, the sensitivity analysis time is about the same for an 8-element system and a 8192-element system. This shows that vectorized codes are not sensitive to the dimension of the problems until the computer capacity is fully exploited. In addition to vmap, the code structure can be further optimized with the implementation of jit via XLA, which is the code structure (JAX-SSO) proposed herein. On CPU, it takes 0.01 s for the sensitivity analysis of an 8-element system and the computational cost increases to 0.23 s for a 2048-element system; the computational time further increases to 1.17 s for a system with 8192 elements. On GPU, the time increases from 0.01 s (8-element system) to 0.04 s (2048-element system), and further to 0.17 s for the system with 8192 elements. For the JAX-SSO code, the low-dimension problem does not benefit much from GPU but for high-dimensional problems, GPU is preferable as it significantly reduces the computational cost: with the 8192 finite elements, GPU reduces 85.8% of the computational time as opposed to the device with only CPU.

![Fig. 5 Performance of JAX-SSO: a total time for sensitivity analysis; b FEA time; c total time minus FEA time](image-url)
The sensitivity analysis via the adjoint method can be decomposed into two sub-tasks: (i) FEA (Sub-task 1): consisting of the assembly of the global stiffness matrix \( K \) and the solution of the linear system \( Ku = f \); (ii) sensitivity analysis given FEA results (Sub-task 2): the calculation of \( \frac{\partial K}{\partial x} \) \( \frac{\partial C}{\partial x} \) (Eq. 8), and the assembly of \( \frac{\partial C}{\partial x} \) from \( \frac{\partial C}{\partial x} \) (Eq. 9). The computational cost of each sub-task is illustrated in Fig. 5b and c, respectively. Similar to Fig. 5a, the JAX-SSO code structure outperforms the other two code structures in both Fig. 5b and c, indicating the optimized code structure of JAX-SSO not only benefits the FEA stage, but also the Sub-task 2. With only CPU, the FEA takes 0.004 s for the 8-element system and it takes 0.21 s for the 8192-element system. In terms of the Sub-task 2, the computational time increases from 0.003 to 0.96 s. With the implementation of GPU for the 8192-element system, the FEA time decreases by 45.0% to 0.12 s and the time for Sub-task 2 is reduced to 0.05 s, manifesting a reduction of 94.8%.

The proposed code structure greatly benefits the calculation and assembly of the global stiffness \( K \), as well as the calculation of \( \frac{\partial C}{\partial x} \) given the FEA results. The maximum number of elements tested herein is sufficient for form finding purposes where hundreds of structural elements are usually used. It has to be pointed out that the speed-up of solving the sparse linear system \( Ku = f \) itself is not addressed by JAX-SSO as it falls outside the scope of this research. Nevertheless, the performance evaluation demonstrates that the proposed framework efficiently helps the structural designers to calculate the sensitivity which is challenging using conventional methods.

### 4 Validation and examples

In this section, we use examples to validate and test the capability of the proposed framework for structural shape optimization. The FEA functionality is validated against the commercial FEA software SAP2000 and the functionality for sensitivity analysis is validated against numerical differentiation via the finite difference method. The properties of the structural members used can be seen in Table 2 unless otherwise specified. For constrained shape optimization, the SLSQP algorithm from the NLopt library (Johnson 2014) is used.

#### 4.1 Validation

We first validate the developed JAX-SSO package through the example illustrated in Fig. 6. A 2D parabolic arch with the maximum height of 5 m spans 10 m and it is discretized into 31 nodes. The left-most and right-most are pin-supported. 5000 kN downward point load is applied to the unsupported 29 structural nodes. For the FEA capability, we validate JAX-SSO by comparing its results to SAP2000.

![Fig. 6](image_url) Validation of JAX-SSO: a structures and loads; b validation of the FEA module: comparison between JAX-SSO and SAP2000; c validation of the sensitivity analysis module: comparison between JAX-SSO and finite difference method

| Property            | Value  | Unit |
|---------------------|--------|------|
| E, Young’s modulus  | 37,900 | Mpa  |
| G, Shear modulus    | 14,577 | Mpa  |
| \( I_y \), Moment of inertia | 0.0072 | m^4 |
| \( I_z \), Moment of inertia | 0.0032 | m^4 |
| A, Sectional area   | 0.24   | m^2  |
The JAX-SSO result shows that the maximum displacement along -Z axis takes place at the center node (X = 5 m) with a magnitude of 5.806 cm and the SAP2000 result shows the maximum displacement is 5.807 cm which happens at the same node (Fig. 6b). The difference in the maximum displacement is 0.017%. In terms of the strain energy, JAX-SSO returns a value of 3011.104 kN·m compared to the SAP2000 result of 3011.115 kN·m and the difference is $3.6 \times 10^{-6}$ (Fig. 6b). The validity of JAX-SSO for FEA is thus proved.

For the sensitivity analysis module in JAX-SSO, we validate it by comparing the partial derivatives of $C$ with respect to the Z coordinates of nodes obtained by JAX-SSO and by the finite (forward) difference method. While all the nodes are checked, we showcase the sensitivity of $C$ with respect to Z-coordinate of the center node (X = 5 m) herein. Figure 6c illustrates the comparison: with the decrease of the increment in finite difference, the partial derivative value decreases to $-12.792$ kN when the step size equals $1 \times 10^{-8}$ m and it is close to the JAX-SSO result $-12.788$ kN. The difference of 0.03% shows the validity of JAX-SSO for sensitivity evaluation.

### 4.2 2D arch

The proposed framework is first implemented for the form finding of a 2D arch. The initial structure (Fig. 7a) is a simply supported bridge spanning 10 m with 21 nodes. The leftmost and rightmost nodes are pinned and their positions remain unchanged during the optimization. The other nodes who are not restrained are the design nodes with randomly generated Z coordinates. A 10kN nodal load along -Z axis is applied to each design node. The design variables for this problem are the Z coordinates of the design nodes while the X coordinates remain unchanged. No constraints are imposed and gradient descent is used as the optimizer.
After 1000 iterations, an arch bridge is found (Fig. 7b). The initial compliance of the bridge is 201 N·m whereas the compliance of the optimized arch is only 6.32 N·m, which is a reduction by 96.9%. As has been known in engineering practice and confirmed by equilibrium-state based form finding methods, the optimal geometry for a two-end-pinned structure under gravity is an arch/catenary. The validity of the framework is then confirmed by this benchmark example. Unlike most equilibrium-state based form finding methods, the gradient-based shape optimization does not limit the structural elements to resisting only membrane forces, it is the “strain energy minimization” process that naturally outputs the arch form that is membrane-force dominant.

4.3 3D barrel arch

The framework is further applied to the form finding of a 3D barrel arch (Fig. 8). The initial structure (Fig. 8a) spans 5 m in both the X and the Y direction and there are in total 225 structural nodes in the system. The structural nodes with X = 0 m or with X = 5 m are pinned. Each node is applied with a 10KN downward (-Z) nodal load. The design nodes are the 195 nodes who are not restrained and they have the same randomly generated Z coordinates prior to the optimization process. The design variables are the Z coordinates of the design nodes. With no constraints, gradient descent is used as the optimizer. The compliance of the system is reduced from 13,495 N·m by 83.2% to 2259 N·m after 500 iterations (Fig. 8c). The optimized structure is in the form of a barrel arch as expected, which can be seen in Fig. 8b.

4.4 Free-form gridshell inspired by Mannheim Multihalle

Mannheim Multihalle (Fig. 9) is a famous gridshell structure designed by Frei Otto via hanging model experiments.

It originally served as a temporary structure for a horticultural exhibition in Mannheim, Germany. Figure 9b and c shows a 1:150 model created by a group of undergraduate students at Princeton University, illustrating how the form of this gridshell structure is found by hanging model experiments. Inspired by the fascinating Mannheim Multihalle, here we apply the proposed framework to the form finding of an exhibition hall that is similar to Mannheim Multihalle.

The initial structure is shown in Fig. 10a and the plan view is similar to that of Mannheim Multihalle. The exhibition hall has a total length of 80 m and a total width of 60 m. There are two entrances for the exhibition hall, located at Y = 0 m and Y = 80 m. After entering the exhibition hall, there is an exhibition space of around 2400 m². The initial structure is discretized into triangular meshes and has 440 structural nodes, which is further connected by 1215 structural members. The nodes on the edges of the structure are non-design nodes and they are pinned. The other 348 nodes are the design nodes with an initial height of 0.5 m. The loads applied to the exhibition hall is downward (-Z) nodal load at each node with a magnitude of 100 kN. In the optimization process, the design variables are the Z coordinates of the design nodes. Gradient descent is selected as the optimizer for this problem since no constraints are imposed on the design variables.

Figure 10c illustrates the history of the compliance during optimization. The initial structure has a total strain energy of 3018.8 kN·m and after 150 iterations, the total compliance is reduced to 30.3 kN·m, which is about 1% as much as the initial compliance. The resulting geometry is shown Fig. 10b: between the entrances and the exhibition space, a barrel-arch
shaped rooftop is found; on the top of the exhibition area, a dome-like geometry is found. The resulting geometry has a natural geometry that resembles the original Mannheim Multihalle as shown in Fig. 9c.

### 4.5 Shape optimization based on Bézier Surface

In the previous examples, the design variables $x$ are the nodal $Z$ coordinates of all the design nodes, which is the non-parametric approach to describe the structural geometry. Sometimes it may cause some issues: (i) there will be too many design variables if there is an extremely fine mesh to the structure; and (ii) the optimized structure may have a jagged surface, which is not ideal in architectural design for aesthetic reasons. In the examples of this subsection, we parameterize the geometry using Bézier Surface to reduce the number of design variables and ensure the smoothness of the structure. For a detailed introduction of Bézier Surface, please refer to the work by Farin (2014). Here we only introduce the basics of Bézier Surface. An $(n, m)$ degree Bézier Surface has a set of $(n+1, m+1)$ control points. The global coordinate of the $(i, j)$-th control point is denoted by $p_{i,j}$ where $i \in \{0, ..., n\}$ and $j \in \{0, ..., m\}$. The global coordinate of any point $P$ is then a function of a set of parametric coordinate $(u, v)$:

$$P(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i}^{n}(u)B_{j}^{m}(v)p_{i,j}$$  \hspace{1cm} (10)

where $B_{i}^{n}(u)$ is a basis Bernstein polynomial:

$$B_{i}^{n}(u) = \frac{n!}{i!(n-i)!}u^i(1-u)^{n-i}$$ \hspace{1cm} (11)

In structural design, the parametric coordinate $(u, v)$ is determined by the projected mesh of the structure. An example of the geometry parameterized by Bézier Surface can be seen in Fig. 11. In the following examples, the design variable is the $Z$-coordinate $p_z$ of the control point $p$. The derivatives of the objective function $C$ or the constraints with respect to the design variable can be easily obtained by
applying the chain rule together with AD. For instance, 
\[
\frac{\partial C}{\partial p_z} = \frac{\partial C}{\partial p} \frac{\partial p}{\partial p_z},
\]
which can all be easily obtained by AD.

### 4.5.1 Four-point supported free-form gridshell

The framework is applied to form-find a gridshell that has four corner-point supports (Fig. 12), which is a classic form finding problem.

The structure (Fig. 12a) spans 5 m along the X direction and 5 m along the Y direction where only the four corner-nodes are pinned. The initial geometry is a dome-like geometry, which is parameterized by 36 control points of Bézier Surface. The Z-coordinate of the control points that are along the edges is 0 m and otherwise the control points have a height of 1 m. The initial geometry is ideal when all four edges are supported but intuitively it is not efficient when only the corner-nodes are supported. The four control points at the corners remain unchanged while the Z coordinates of other 32 control points are the design variables in the optimization process. The structure is discretized by quadrilateral mesh and there are 256 nodes in the structural system. Each structural node is subjected to a 500 KN load along -Z axis.

To find the optimum geometry for this four-point supported gridshell, gradient descent is selected as the optimizer with a maximum iteration number of 150. No constraints are imposed on the optimization problem. For the initial structure, the total strain energy is 1194.2 KN⋅m and after 150 iterations, the compliance converges to about 210 KN⋅m (Fig. 12c). The optimized structure’s total strain energy is only 17.6% of the initial structure. The overall geometry of the optimized structure has positive Gaussian curvature that acts like a dome, which is similar to the initial one.
However, the edge of the optimized structure changes from a straight line to an arch, which is an ideal load path to transport the nodal loads to the support by membrane forces. In addition, the resulting shape corresponds to the shape from equilibrium-state based form finding and as one can see in Bletzinger et al. (2005), which further validates the proposed framework.

4.5.2 Two-edge supported free-form gridshell

In this subsection, two optimization tasks are conducted for the form finding of a gridshell that is supported along two adjacent edges (Fig. 13): (i) unconstrained shape optimization; and (ii) constrained shape optimization: the maximum structural height is constrained.

Unconstrained shape optimization The initial geometry, the load and the mesh are the same as the example in Sect. 4.5.1 but the boundary conditions are different. The boundary condition in this example gives a design problem of finding an optimized geometry for a canopy: the 31 structural nodes whose $X=0$ m or $Y=0$ m are pinned while the other 225 structural nodes are free. The structural geometry is parameterized by 36 control points of Bézier Surface. During the optimization process, 11 control points whose $X=0$ m or $Y=0$ m remain unchanged while the Z coordinates of the other 25 control points are the design variables for this task. Gradient descent is implemented as the optimizer and the maximum iteration number is 150. The history of the compliance is illustrated by the black line in Fig. 14a. With the canopy-like boundary condition, the dome-like initial geometry has a total strain energy of 1487 KN·m. After 150 iterations, gradient descent reduces the total strain energy by 76% to 357 KN·m. The optimized structure is a naturally formed canopy (Fig. 13b) with positive Gaussian curvature.

Constrained shape optimization In structural and architectural design, constraints usually need to be applied to the form finding process, such as the constraint on the height of the structure. In this paragraph, a constrained shape optimization problem is formulated and solved. The initial geometry, the load, the boundary conditions as well as the design variables are the same as the unconstrained problem. Different to the unconstrained problem, constraints are imposed to the structural height: $Z_{\text{max}}(x) \leq 3$ m, where $Z_{\text{max}}(x)$ is the maximum structural height. SLSQP is set as the optimizer and the maximum number of iterations is set to be 150. Figure 14a and b illustrate the optimization history of the compliance and the maximum structural height during the optimization process, respectively. After 150 iterations, the compliance is reduced from 1487 KN·m to 368.8 KN·m which is a reduction of 75%. The maximum structural height is successfully limited to 3 m as opposed to the unconstrained case where the maximum structural height of the optimized structure goes up 3.93 m. The optimized structural shape is illustrated in Fig. 13c. To limit the maximum structural height to 3 m, the height of some control points is much lower to parameterize a geometry that fulfills the height constraints. For instance, the rightmost control point in Fig. 13c has a significantly low height of around 1.5 m as if it is “pulling” the structure downwards to fulfill the height requirement.

4.5.3 Free-form canopy inspired by Carioca Wave

“Carioca Wave” (Helbig et al. 2014) is a free-form shell canopy located in Rio de Janeiro (Fig. 15a), which covers an atrium of a shopping center with its elegant geometry. In this example, the proposed framework is applied to the shape optimization of a free-form canopy and the problem is modified from the the design of “Carioca Wave”. Three different tasks are considered herein: (i) unconstrained shape optimization; (ii) constrained shape optimization: constraint is imposed on the maximum structural height; and (iii) unconstrained shape optimization: as opposed to task (i), the bending stiffness of the structural elements are reduced.
Unconstrained shape optimization

The initial structure is shown in Fig. 15b, where the total length of the structure is 50 m and the total width is 25 m. The plan view of the initial structure resembles that of “Carioca Wave” and triangular mesh is used to discretize the structure. The initial geometry is parameterized by a Bézier Surface with a grid of 10 × 10 control points. The height of the control points is Z=0 m when Y=0 m and it is Z=-6 m when Y=25 m. For all the other control points, the initial height is set to be Z=10 m. The structural nodes are supported by pin supports when Y=0 m and Y=25 m. During the optimization process, the design variables are the the Z coordinates of the control points whose Y ≠ 0 m and Y ≠ 25 m. Gradient descent is selected as the optimizer and the maximum number of iteration is set to be 150. After 150 iterations, the optimized shape can be seen in Fig. 15c. Different from the initial structure that all the design control points have a height of 10 m, the optimized geometry forms a partial barrel arch shape that carries the force to the supports by axial force. The black line in Fig. 16a shows the history of compliance during the optimization process. The initial structure has a compliance of 842 kN·m and after 150 iterations, the compliance decreases by 96% to of 34.8 kN·m.  

Constrained shape optimization

In this task, the optimization problem is formulated in the same way as the
unconstrained case except that the maximum height of the structure is limited to 10 m. SLSQP is implemented as the optimizer and the maximum number of iteration is set to be 150. The optimized structure can be seen in Fig. 15d and similarly, a smooth barrel arch geometry is found despite the chaotic distribution of the control points. This smooth geometry results in a 94.5% reduction of the compliance from 842 kN⋅m to 46.5 kN⋅m, as illustrated by the red line in Fig. 16a. The history of the maximum structural height is shown in Fig. 16. Compared to the unconstrained case where the maximum height goes up to more than 15 m after 150 iterations, the maximum structural height of the form-found geometry is successfully limited to 10 m.

Unconstrained shape optimization with reduced bending stiffness

Compared to the Carioca Wave, the form-found geometries in the above examples are not doubly curved. In this example we will illustrate how the bending stiffness of structural elements influences the optimized geometry by conducting an additional unconstrained optimization problem in which the moment of inertia $I_y$ and $I_z$ are reduced to 10% of the original values shown in Table 2. All the other conditions remain the same as the first unconstrained example. Gradient descent is selected as the optimizer and after 150 iteration, the optimized structure (Fig. 17) is found, which is similar to Carioca Wave. With the decrease of the bending stiffness, the resulting structure is doubly curved and manifests negative Gaussian curvature, which is different from the original case (Fig. 15c). The final geometry shows that by having smaller bending stiffness in structural members, the load-resisting mechanism is transformed to a combination of compression and tension, indicated by the negative Gaussian curvature. Through this example, it is shown that the properties of the structural elements play a significant role in the form-found geometry, indicating the necessity of applying gradient-based shape optimization for form finding. In gradient-base shape optimization, the structural elements used in form finding are identical to the ones used in structural analysis and construction. On the contrary, in most equilibrium-based form finding methods, the structural elements used may not be the same as the elements used in FEA and actual construction, which may lead to meaningless form-found geometry.

5 Conclusions

To facilitate gradient-based structural shape optimization for efficient form finding, a framework based on automatic differentiation (AD), accelerated linear algebra (XLA) and the adjoint method is proposed herein. The features of this framework are highlighted as follows.

- JAX-enabled AD and the adjoint method for gradient calculation: in gradient-based structural shape optimization, the sensitivity analysis is of great significance and it is often the bottleneck. AD is implemented herein for efficient and accurate sensitivity analysis in combination with the adjoint method, which alleviates the implementation efforts and avoids the possible errors of the derivation of analytical sensitivity.
- JAX-enabled XLA and vectorization for fast sensitivity analysis: an efficient code structure is proposed herein with the implementation of vectorization and just-in-time compilation to speed up the sensitivity analysis. The code structure also enables GPU-acceleration for faster calculation for high-dimensional problems.
- Coherence between structural analysis and form finding: in this framework, the FEA is conducted alongside with the optimization process and the structural response is obtained once the optimization is done. By applying this framework, there will be less design iterations as the optimization process is coherent with the analysis using FEM as opposed to most equilibrium-based form finding methods.

The proposed framework is applied to several form finding test cases, including arches and free-form shells. Two geometry descriptive methods are used, including non-parametric and parametric via Bézier Surface. In addition, unconstrained and constrained shape optimization problems are considered. The optimization results demonstrate the validity and capability of the proposed framework. In addition, the examples also show that the properties of the structural elements will influence the optimized geometry, which illustrates the necessity of implementing the gradient-based shape optimization approach as it is informed by the properties of structural elements used in structural analysis. However, it has to be pointed out that the optimization algorithms implemented may stuck at
local minimum so future work should be conducted to see how the initial geometry influences the optimized geometry. Some other directions of future research include: the extension of JAX-SSO to more structural elements such as shell elements; the comparison between gradient-based shape optimization and other form finding methods; the development of new technique that integrates the benefits of different form finding methods; the development of codes that exploits the power of HPC clusters and multi-CPUs/GPUs to benefit problems with larger scales.

Nevertheless, this paper demonstrates the benefits of utilizing AD, XLA, and the adjoint method in the context of efficient form finding through gradient-based shape optimization, providing valuable assistance to designers in the creation of better structures for the built environment. The differentiable nature of this framework also serves as a demonstration of its potential for facilitating the development of a differentiable FEA solver that can assist addressing inverse design problems and machine learning research within the field of structural engineering.

Appendix A: code example—derivative calculation using JAX for customized python objects

The code snippet (Code 3) illustrates how the JAX methods work by presenting a simple problem: calculating how the length of a 2D line changes with the nodal coordinates, i.e., the gradient of the length with respect to the nodal coordinates of the line. Firstly, a new class that represents a line in 2D world is defined and its attributes are the nodal coordinates. The line() class is registered to JAX by @register_pytree_node_class, tree_flatten and tree_unflatten. A function called L() is firstly defined, outputting the distance between two 2D points. Note that jax.numpy is used in the function L() for calculating distance instead of numpy, which is of great significance because we need to trace the operations so that the derivatives can be obtained through automatic differentiation. An external function called line_sens() is then defined for line() objects to calculate the derivatives of its length with respect to nodal coordinates. This original line_sens() works for a single line() object that represents one line. 100 lines with random coordinates are then created. We then implement jit to compile the line_sens() function into optimized machine codes. Lastly, we use vmap to vectorize the line_sens() function so that it can be applied to a single line() object that represents multiple lines.
Appendix B example codes in JAX-SSO

```python
#Stiffness matrix of the element

def BeamCol_K(eleTag, i_nodeTag, j_nodeTag, x1, y1, z1, x2, y2, z2, E, G, Iy, Iz, J, A):
    #Create a beam-column
    this_beamcol = BeamCol(eleTag, i_nodeTag, j_nodeTag, x1, y1, z1, x2, y2, z2, E, G, Iy, Iz, J, A)
    #Get the local stiffness matrix of this element
    return this_beamcol.K()

@jit
def Ele_Sens_K_Coord(BeamCol):
    
    Return the sensitivity of element's local stiffness matrix (in global coordinates) w.r.t. the element's coordinates.

    Inputs:
    BeamCol: A 'BeamCol' object

    #Properties of this beam column
    eleTag = BeamCol.eleTag
    i_nodeTag, j_nodeTag = [BeamCol.i_nodeTag, BeamCol.j_nodeTag]
    x1, y1, z1, x2, y2, z2 = [BeamCol.x1, BeamCol.y1, BeamCol.z1, BeamCol.x2, BeamCol.y2, BeamCol.z2]
    E = BeamCol.E
    G = BeamCol.G
    Iy = BeamCol.Iy
    Iz = BeamCol.Iz
    J = BeamCol.J
    A = BeamCol.A

    #Calculate the sensitivity
    #arguments indicates the variables to which the Jacobian is calculated
    #and in our case they are (x1,..., z2)
    return jacfwd(BeamCol_K, arguments =((3,4,5,6,7,8))(eleTag, i_nodeTag, j_nodeTag, x1, y1, z1, x2, y2, z2, E, G, Iy, Iz, J, A))
```

Code 4 Functions that calculate the sensitivity of element’s stiffness matrix with respect to its nodal coordinates

Acknowledgements The author would like to thank his advisor Professor Maria Garlock and friends at Princeton University for the insightful discussions.

Author Contributions GW: Conceptualization, Methodology, Software Validation, Formal analysis, Writing—original draft, Writing - review & editing.

Declarations

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

Replication of results The code for this paper is available online: https://github.com/GaoyuanWu/JaxSSO

References

Abadi M, Agarwal A, Barham P et al. (2015) TensorFlow. Large-scale machine learning on heterogeneous systems. https://doi.org/10.5281/zenodo.4724125

Adriaenssens S, Block P, Veenendaal D et al. (2014) Shell structures for architecture: form finding and optimization. Routledge. https://doi.org/10.4324/9781315849270

Barnes MR (1999) Form finding and analysis of tension structures by dynamic relaxation. Int J Space Struct 14(2):89–104. https://doi.org/10.1260/02663519919494722

Baydin AG, Pearlmutter BA, RADUL AA et al. (2018) Automatic differentiation in machine learning: a survey. J Mach Learn Res 18:1–43

Bezgin DA, Buhendwa AB, Adams NA (2023) JAX-fluids: a fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. Comput Phys Commun 282(108):527. https://doi.org/10.1016/j.cpc.2022.108527

Bletzinger KU, Ramm E (1993) Form finding of shells by structural optimization. Eng Comput 9(1):27–35. https://doi.org/10.1007/BF01198251

Bletzinger KU, Ramm E (2001) Structural optimization and form finding of lightweight structures. Comput Struct 79(22–25):2053–2062. https://doi.org/10.1016/S0045-7949(01)00052-9

Bletzinger KU, Wüchner R, Daoud F et al. (2005) Computational methods for form finding and optimization of shells and membranes. Comput Methods Appl Mech Eng 194(30–33):3438–3452. https://doi.org/10.1016/j.cma.2004.12.026

Boggs PT, Tolle JW (1995) Sequential quadratic programming. Acta Numerica 4:1–51

Bradbury J, Frostig R, Hawkins P, et al. (2018) JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax

Chandrasekhar A, Sridhara S, Suresh K (2021) Auto: a framework for automatic differentiation in topology optimization. Struct Multidiscip Optim 64(6):4355–4365. https://doi.org/10.1007/s00158-021-03025-8

Cuvilliers PPE (2020) The constrained geometry of structures: optimization methods for inverse form-finding design. PhD thesis, Massachusetts Institute of Technology

Ding Y (1986) Shape optimization of structures: a literature survey. Comput Struct 24(6):985–1004

Espath L, Linn RV, Awruch A (2011) Shape optimization of shell structures based on nurb's description using automatic differentiation. Int J Numer Meth Eng 88(7):613–636. https://doi.org/10.1002/nme.3183

Farin G (2014) Curves and surfaces for computer-aided geometric design: a practical guide. Elsevier

Gill PE, Wong E (2012) Sequential quadratic programming methods. Mixed integer nonlinear programming. Springer, pp 147–224
Griewank A, Walther A (2008) Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM
Griewank A et al. (1989) On automatic differentiation. Math Program 6(6):83–107
Harris CR, Millman KJ, Van Der Walt SJ et al. (2020) Array programming with numpy. Nature 585(7825):357–362. https://doi.org/10.1038/s41586-020-2649-2
Hascoet L, Pascual V (2013) The tapenade automatic differentiation tool: principles, model, and specification. ACM Trans Math Softw 39(3):1–43. https://doi.org/10.1145/0000000.000000
Helbig T, Giampellegrini L, Oppe M (2014) “Carioca wave”—a free-form steel-and-glass canopy in rio de janeiro, brazil. Steel Construct 7(4):252–257. https://doi.org/10.1002/stco.201420033
Johnson SG (2014) The nlopt nonlinear-optimization package. https://github.com/steve ngj/nlopt
Kilian A, Ochsendorf J (2005) Particle-spring systems for structural form finding. J Int Assoc Shell Spatial Struct 46(2):77–84
Kraft D (1988) A software package for sequential quadratic programming. Forschungsbericht- Deutsche Forschungs- und Versuchsanstalt fur Luft- und Raumfahrt
Lam SK, Pitrou A, Seibert S (2015) Numba: A llvm-based python JIT compiler. In: Proceedings of the second workshop on the LLVM compiler infrastructure in HPC, pp 1–6
Nørgaard SA, Sagebaum M, Gauger NR et al. (2017) Applications of automatic differentiation in topology optimization. Struct Multidiscip Optim 56(5):1135–1146. https://doi.org/10.1007/s00158-017-1708-2
Ozaki I, Kimura F, Berz M (1995) Higher-order sensitivity analysis of finite element method by automatic differentiation. Comput Mech 16(4):223–234. https://doi.org/10.1007/BF00369867
Paganini A, Wechsung F (2021) Fireshape: a shape optimization toolbox for firedrake. Struct Multidiscip Optim 63(5):2553–2569. https://doi.org/10.1007/s00158-020-02813-y
Pastrana R, Ohlbrock PO, Oberbichler T et al. (2022) Constrained form-finding of tension-compression structures using automatic differentiation. Comput Aid Design. https://doi.org/10.1016/j.cad.2022.103435
Robles SI, Ortega NF (2001) Study of volumetric displacements of shells. J Int Assoc Shell Spatial Struct 42(3):139–147
Rombouts J, Lombaert G, De Laet L et al. (2019) A novel shape optimization approach for strained gridshells: design and construction of a simply supported gridshell. Eng Struct 192:166–180. https://doi.org/10.1016/j.engstruct.2019.04.101
San B, Feng D, Qiu Y (2021) Shape optimization of concrete free-form shells considering material damage. Eng Optim. https://doi.org/10.1080/0305215X.2021.1968854
Schek HJ (1974) The force density method for form finding and computation of general networks. Comput Methods Appl Mech Eng 3(1):115–124. https://doi.org/10.1016/0045-7825(74)90045-0
Schoenholz S, Cubuk ED (2020) JAX MD: a framework for differentiable physics. Adv Neural Inf Process Syst 33:11428–11441
Tortorelli DA, Michaleris P (1994) Design sensitivity analysis: overview and review. Inverse Prob Eng 1(1):71–105. https://doi.org/10.1080/174159794000027573
Uysal H, Gul R, Uzman U (2007) Optimum shape design of shell structures. Eng Struct 29(1):80–87. https://doi.org/10.1016/j.englstruct.2006.04.007
Van Keulen F, Haftka R, Kim NH (2005) Review of options for structural design sensitivity analysis: part 1—linear systems. Comput Methods Appl Mech Eng 194(30–33):3213–3243. https://doi.org/10.1016/j.cma.2005.02.002
Xia Y, Langelaar M, Hendriks MA (2021) Optimization-based three-dimensional strut-and-tie model generation for reinforced concrete. Comput Aid Civ Infrastruct Eng 36(5):526–543. https://doi.org/10.1111/mice.12614
Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.