Computer-Aided Design 146 (2022) 103199

Contents lists available at ScienceDirect

Computer-Aided Design

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

NURBS-Diff: A Differentiable Programming Module for NURBS

Anjana Deva Prasad [a], Aditya Balu [a], Harshil Shah [a], Soumik Sarkar [a], Chinmay Hegde [b], Adarsh Krishnamurthy [a,*]

[a] Iowa State University, Ames, IA, USA
[b] New York University, New York, NY, USA

ARTICLE INFO

Article history:
Received 3 September 2021
Received in revised form 13 December 2021
Accepted 5 January 2022

Keywords:
Differentiable NURBS module
NURBS
Geometric deep learning
Surface modeling

ABSTRACT

Boundary representations (B-reps) using Non-Uniform Rational B-splines (NURBS) are the de facto standard used in CAD, but their utility in deep learning-based approaches is not well researched. We propose a differentiable NURBS module to integrate NURBS representations of CAD models with deep learning methods. We mathematically define the derivatives of the NURBS curves or surfaces with respect to the input parameters (control points, weights, and the knot vector). These derivatives are used to define an approximate Jacobian used for performing the "backward" evaluation to train the deep learning models. We have implemented our NURBS module using GPU-accelerated algorithms and integrated it with PyTorch, a popular deep learning framework. We demonstrate the efficacy of our NURBS module in performing CAD operations such as curve or surface fitting and surface offsetting. Further, we show its utility in deep learning for unsupervised point cloud reconstruction and enforce analysis constraints. These examples show that our module performs better for certain deep learning frameworks and can be directly integrated with any deep-learning framework requiring NURBS.

1. Introduction

In modern CAD systems, a solid model is represented using boundary representation (B-Rep), where the solid boundaries are defined using spline surfaces. Non-Uniform Rational B-splines (NURBS) are the standard representation used for defining the spline surfaces [1]. NURBS surfaces offer a high level of control and versatility; they can also compactly represent the surface geometry. NURBS surfaces can represent more complex shapes than Bèzier or B-splines due to the non-uniformity of the knot vectors and the non-linear transformation due to the weights assigned to the control points. In addition, the NURBS definition allows for local control via the knots and the control points and global control via the weights. On the other hand, deep learning for 3D Euclidean geometry is emerging as a critical and well-explored research area in engineering. This area includes fundamental computer vision works such as 3D shape reconstructions from point clouds or multi-view stereo and 3D semantic segmentation for shape understanding [2–10]. While NURBS are the standard CAD representation in engineering, their utility in deep learning-based approaches is not well researched. Current 3D deep learning (DL) research focuses on converting standard CAD representations to geometric representations that are more amenable to machine learning (such as voxels, triangular meshes, etc.) [11]. This conversion from the standard CAD geometries to other representations is often irreversible and is not trivial to incorporate with DL algorithms.

One of the main challenges in extending NURBS-based representation to deep learning is the differentiable programming of the NURBS evaluations. The main idea of differentiable programming is to define each operation in a neural network in a differentiable manner. Differentiable programming uses gradient-based approaches to optimize the neural network parameters, thereby obtaining the desired output through neural network operations. This differentiable programming paradigm allows an end-to-end programmable system that can be used to train deep neural networks. This paradigm has found use in a large variety of applications such as scientific computing [12–14], image processing [15], physics engines [16], computational simulations [17], and graphics [18,19].

For differentiable programming of NURBS, we need to compute gradients of the NURBS surface points with respect to the parameters of the NURBS representation (see Fig. 1). However, since the NURBS surface points are a function of knots, control points, and weights, the partial derivative with respect to each input parameter needs to be computed, making the backward evaluation challenging. Further, due to the use of basis functions

* Corresponding author.
E-mail addresses: anjana@iastate.edu (A. Deva Prasad), baditya@iastate.edu (A. Balu), harshil@iastate.edu (H. Shah), soumiks@iastate.edu (S. Sarkar), chinmay.h@nyu.edu (C. Hegde), adarsh@iastate.edu (A. Krishnamurthy).

https://doi.org/10.1016/j.cad.2022.103199
0010-4485/

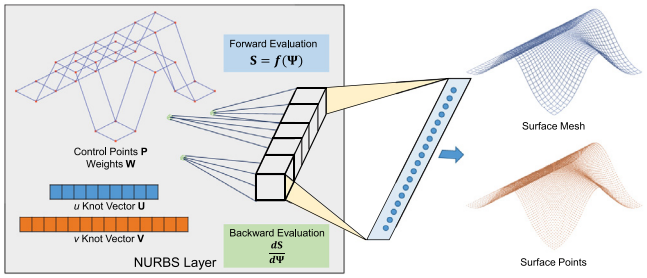


Fig. 1. We propose a differentiable NURBS module that can be used for CAD geometric modeling using standard deep learning systems. The NURBS parameters are input to the module during the forward evaluation, and we evaluate the surface mesh. Once a loss is computed, and gradients for the surface mesh are obtained, we perform a backward evaluation to enable backpropagation of the losses to modify the input parameters.

(which are piecewise-continuous polynomials), gradients might be discontinuous (or even zero) at the knots. The recursive definition of the basis functions adds an additional challenge to define derivatives with respect to the knot vectors. To alleviate these issues, we exploit the recent theoretical advances in the paradigm of differentiable programming. It has been theoretically proven that if a weak form of the Jacobian for the "forward" evaluation operator can be represented using a *block-sparse* matrix, then this approximate Jacobian can be used to perform the "backward" evaluation of that operation [20,21]. This approach has been used to define approximate derivatives for operations such as sorting [20,22], loops and algorithmic conditions, and even derivatives for piecewise polynomial functions [23]. We use a similar approach by defining a *block-sparse* Jacobian for NURBS surface evaluation. Existing differentiable programming approaches for splines [24] mainly focus on optimizing the control point locations. In this paper, we provide a complete module for integrating NURBS with differentiable frameworks that optimize not just the control points but also the knots for reparameterization.

Formally, using the differentiable programming approach explained above, we formulate a differentiable NURBS ("*NURBS-Diff*") module, which enables deep learning frameworks to integrate NURBS-based representation of B-Rep surfaces and perform CAD operations using them. The forward pass of our NURBS module uses the standard NURBS evaluation as explained in Section 3.1. The backward pass uses the derivatives of the NURBS curve or surface with respect to the input parameters of the NURBS (Section 3.2). The derivatives are used to define the Jacobian that is then used for backpropagation of the losses during training. After defining *NURBS-Diff*, we validate our approach by performing traditional CAD operations such as curve fitting, surface fitting, and surface offsetting. Finally, we show the applicability of *NURBS-Diff* in deep learning by using it as an additional decoder for point cloud reconstruction.

In this paper, we have developed a differentiable NURBS module that can be used in machine learning and CAD applications. The key contributions of this work are:

1. A differentiable programming framework using NURBS representation where the losses can be back-propagated using the NURBS definitions. Specifically, we define the Jacobian based on the derivatives of the NURBS with respect to its input parameters.
2. A GPU-accelerated implementation of our NURBS module in PyTorch for better integration with existing deep learning programming frameworks.
3. A gradient descent-based optimization framework using *NURBS-Diff* for performing CAD operations such as curve or surface fitting and surface offsetting.
4. The applicability of our proposed differentiable programming framework to extend the training process of unsupervised point cloud reconstruction of NURBS surfaces.

The rest of the paper is arranged as follows. We outline some close related work in point cloud reconstruction, machine learning approaches in CAD, and differentiable programming in Section 2. We provide the mathematical details of our differentiable NURBS module in Section 3. We show the application of *NURBS-Diff* to CAD operations in Section 4 and to unsupervised point cloud reconstruction in Section 5.

2. Related work

The problem of extracting concise geometry representations from a spectrum of input data formats such as images, depth maps, and point clouds has been extensively studied over the last few decades. While methodologies that derive such representations are pervasive in 3D reconstruction literature today, our NURBS module focuses on filling the gap between the NURBS-based CAD representation and the other input formats used in machine learning. In this context, we broadly categorize the prior related work under differentiable programming and splines in deep learning.

2.1. Differentiable programming

NURBS surfaces are obtained as a tensor product of two-piecewise polynomial B-spline curves. To conceptualize end-to-end trainable deep learning systems that can fit NURBS surfaces to various input geometries, we require a framework that can backpropagate over such piecewise polynomial functions. Several recent works have been proposed that take advantage of the differentiable programming paradigm to approximate gradients for such functions. Cuturi et al. [20] and Blondel et al. [25] propose differentiable operators for sorting based tasks. Similarly, Vlastelica et al. [26] compute gradients for several optimization problems by constructing linear approximations to discrete-values functions. We model our module on prior work that incorporates structured priors as modules in the deep learning framework, similar to Sheriffdeen et al. [27], Joshi et al. [28], and Djolonga and Krause [29]. Beyond deep learning-based approaches, automatic differentiation for NURBS parametric coordinates for obtaining the surface derivatives for Adjoint-based sensitivity analysis has been performed by Zhang [30]. Ugolotti et al. [31] performed a gradient-based aerodynamic shape optimization using a robust Machine Learning model, which is created to integrate the geometry generation and the mesh generation process using one single polynomial module for the volumetric mesh. Mykhaskiv et al. [32] and Müller et al. [33] define a differentiated CAD kernel in OpenCASCADE for applying algorithmic differentiation to a complete CAD system for shape optimization and imposing constraints. These works encourage us to pursue a similar research direction in developing the *NURBS-Diff* module. However, their approach for obtaining the gradients involves tedious operations such as performing singular value decomposition (SVD) on the control points to obtain the gradients. We also note that the mathematical definition of the derivatives for NURBS and its application to fitting has been explored previously [34,35]. However, due to discontinuities, a more stable and faster approach for computing the derivatives is needed.

2.2. Splines in deep learning

Several deep learning frameworks use splines. Minto et al. [36] use NURBS surfaces fitted over the 3D geometry as an input representation for the object classification task of ModelNet10 and ModelNet40 datasets. Erwinski et al. [37] presented a neural-network-based contour error prediction method for NURBS paths. Fey et al. [38] present a new convolution operator based on B-splines for irregular structured and geometric input, e.g., graphs or meshes. Balestriero et al. [39] build a theoretical link between deep networks and spline functions and build end-to-end deep learning systems using spline-based activation functions. Balu et al. [40] propose a NURBS-aware convolutional neural network that maintains the topological structure similar to a parametric NURBS surface evaluation grid. Very recently, Sharma et al. [41] performed point cloud reconstruction to predict a B-spline surface, which is later processed to obtain a complete CAD model with other primitives "stitched" together. In our work, we perform comparison between our approach and Sharma et al. [41] in Section 5.

3. NURBS-Diff module

Modern CAD systems make use of boundary-representation (B-Rep) for representing a solid geometry, Ω, which is embedded in a 2D or 3D Euclidean space ($\Re^2$ or $\Re^3$). A B-Rep consists of a set of surfaces $d\Omega$ representing the boundary of the solid. Each surface $\mathcal{S} \subset d\Omega$ in a standard CAD system is represented using a Non-Uniform Rational B-spline (NURBS) surface. The NURBS representation is a compact representation that uses a set of control points, knot vectors, degrees, and weights to map a parametric space to span the entire surface $\mathcal{S}$ in the Euclidean space. In this work, we propose a differentiable NURBS module which could evaluate the surface $\mathcal{S}$ given the control points, knot vectors, degrees, and weights, usually obtained as an output from a deep learning system, $NN(\theta)$. This deep learning system is trained using a loss function $\mathcal{L}(\cdot,\cdot)$, computed between a target point cloud $\mathbf{P} \in \{\Re^{Nx2}$ or $\Re^{Nx3}\}$ (where N is the number of points) and a set of points $\mathbf{S}$ sampled (or evaluated) from the surface $\mathcal{S}$. During the training process, the gradient of the loss function with respect to the parameters of the deep learning model, $\partial\mathcal{L}/\partial\theta$ is required for back propagation. It is usually straightforward to compute the derivative of the loss function $\partial\mathcal{L}/\partial\mathcal{S}$ for some of the standard loss functions used, such as Chamfer distance, L_2 distance, etc., since they are differentiable. However, $\partial\mathcal{L}/\partial\theta$ requires a mathematically consistent definition of $\partial\mathcal{S}/\partial\Psi$, where Ψ refers to the complete set of NURBS parameters (i.e. the set of control points $\mathbf{P}$, its corresponding weights $\mathbf{W}$, and the knot vectors $\mathbf{U}$ and $\mathbf{V}$) that define the surface. The gradient $\partial\mathcal{S}/\partial\Psi$ is necessary because the deep learning system $NN(\theta)$ predicts this set of NURBS parameters Ψ and computing $\partial\mathcal{L}/\partial\theta$ requires computing $\partial\mathcal{S}/\partial\Psi$, $\partial\mathcal{L}/\partial\mathcal{S}$ and finally, $\partial\Psi/\partial\theta$. Formally, this can be explained using the chain rule as:

$$\frac{\partial\mathcal{L}}{\partial\theta} = \frac{\partial\mathcal{L}}{\partial\mathcal{S}}\frac{\partial\mathcal{S}}{\partial\Psi}\frac{\partial\Psi}{\partial\theta} \tag{1}$$

The main challenge in this approach is computing $\partial\mathcal{S}/\partial\Psi$. To this end, we propose a differentiable NURBS module implemented as a forward and backward machine learning module. While our module can handle both curve and surface point computations, we limit the discussions of our forward and backward algorithms to surfaces. As shown in the results section, the approach can be directly used for curves embedded in both 2D and 3D space by suitably adjusting the dimensions of the NURBS parameters.

3.1. Forward evaluation for NURBS surface

The NURBS surface $\mathcal{S}$ is sampled over a finite parametric space (u, v) where $(u, v) \in ([0, 1] \times [0, 1])$, and this set of finite points $\mathbf{S}$ representing the surface is used for performing the loss computation and the backward gradient computation. This set $\mathbf{S}$ is computed as a function of NURBS parameters $\Psi = \{\mathbf{P}, \mathbf{U}, \mathbf{V}, \mathbf{W}\}$. Given the NURBS surface points are a function of the NURBS parameters in Eq. (2), we compute the forward evaluation using the NURBS formulation:

$$\mathbf{S} = \mathbf{f}(\mathbf{P}, \mathbf{U}, \mathbf{V}, \mathbf{W}) \tag{2}$$

3.1.1. NURBS formulation

Formally, a point in the NURBS surface parametrized using (u, v) is defined as follows:

$$\mathbf{S}(u, v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij} \mathbf{P}_{ij}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij}}, \tag{3}$$

Here, the basis functions of NURBS, (N_i, N_j) are polynomials that are recursively computed using Cox–de Boor recursion formula in Eq. (4), where u is the parameter value, N_i^p is the ith basis function of degree p.

$$N_i^p(u) = \frac{u - u_i}{u_{i+p} - u_i} N_i^{p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1}^{p-1}(u) \tag{4}$$

$$N_i^0(u) = \begin{cases} 1 & \text{if } u_i \le u \le u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Here, u_i (also known as knots) refers to the elements of the knot vector $\mathbf{U}$ (similarly, $v_i \in \mathbf{V}$). The knot vector is a non-decreasing sequence of parametric coordinates, which divides the B-spline into non-uniform piecewise functions. The basis function N_i^p spans over the parametric domain based on the knot vector and degree as shown in Eqs. (4) and (5). Note that the formulation explained in Eq. (3) uses the vector notation, where $\mathbf{P}_{ij}$ is embedded in $\Re^3$.

3.1.2. Surface point evaluation

The complete algorithm for forward evaluation of $\mathbf{S}(u, v)$ as described in Piegl and Tiller [1] can be divided into three steps:

1. Finding the knot span of $u \in [u_i, u_{i+1})$ and the knot span of $v \in [v_j, v_{j+1})$, where $u_i, u_{i+1} \in \mathbf{U}$ and $v_j, v_{j+1} \in \mathbf{V}$. This is required for the efficient computation of only the non-zero basis functions.
2. Now, we compute the non-zero basis functions $N_i^p(u)$ and $N_j^q(v)$ using the knot span. The basis functions have specific mathematical properties that help us evaluate them efficiently. The partition of unity and the recursion formula ensure that the basis functions are non-zero only over a finite span of $p + 1$ control points. Therefore, we only compute those $p + 1$ non-zero basis functions instead of the entire n basis function. Similarly in the v direction we only compute $q + 1$ basis functions instead of m.
3. We first compute the weighted control points $\mathbf{P}_{ij}^w$ for a given control point $\mathbf{P}_{ij} = \{\mathbf{P}_x, \mathbf{P}_y, \mathbf{P}_z\}$ and weight w_{ij} as $\{\mathbf{P}_x w, \mathbf{P}_y w, \mathbf{P}_z w\}$ representing the surface after homogeneous transformation for ease of computation. Once the basis functions are computed we multiply the non-zero basis functions with the corresponding weighted control points, $\mathbf{P}_{ij}^w$. This result, $\mathbf{S}'$ is then used to compute $\mathbf{S}(u, v)$ as $\{S'_x/S'_w, S'_y/S'_w, S'_z/S'_w\}$.

3.1.3. Implementation

In a deep learning system, each module is considered an independent unit that performs the computation. During the forward pass, the module takes a batch of input and transforms them using the parameters of the module parameters. Further, to reduce the computations needed during the backward pass, we store extra information for computing the gradients during the forward computation. The *NURBS-Diff* module takes as input the control points, weights, and knot vectors for a batch of NURBS surfaces. We define a parameter to control the number of points evaluated from the NURBS surface. We define a mesh grid of a uniformly spaced set of parametric coordinates $u_{grid} \times v_{grid}$. We perform a parallel evaluation of each surface point $S(u, v)$ in the $u_{grid} \times v_{grid}$ for all surfaces in the batch and store all the required information for the backward computation. The complete algorithm is shown in Algorithm 1.

Algorithm 1: Forward algorithm for multiple surfaces

Input : **U**, **V**, **P**, **W**, output resolution n_{grid}, m_{grid}
Output: **S**
Initialize a meshgrid of parametric coordinates uniformly from [0, 1] using $n_{grid} \times m_{grid} : u_{grid} \times v_{grid}$
Initialize: $\mathbf{S} \rightarrow \mathbf{0}$
for $k = 1 :$ *surfaces in* **parallel do**
  for $j = 1 : m_{grid}$ *points in* **parallel do**
    for $i = 1 : n_{grid}$ *points in* **parallel do**
      Compute u_{span} and v_{span} for the corresponding u_i and v_i using knot vectors $\mathbf{U_k}$ and $\mathbf{V_k}$
      Compute basis functions N_i and N_j basis functions using u_{span} and v_{span} and knot vectors $\mathbf{U_k}$ and $\mathbf{V_k}$
      Compute surface point $\mathbf{S}(u_i, v_j)$ (in x, y, and z directions).
      Store u_{span}, v_{span}, N_i^p, N_j^q, and $\mathbf{S}(u_i, v_j)$ for backward computation

Our implementation is robust and modular for different applications. For example, if an end-user desires to use this for a B-spline evaluation, they need to set the knot vectors to be uniform and weights **W** to be 1.0. In this case, the forward evaluation can be simplified to $\mathbf{S}(u, v) = \mathbf{f}(\mathbf{P})$. Further, we can also pre-compute the knot spans and basis functions during the initialization of the NURBS-Diff module. During computation, we could use tensor comprehension that significantly increases the computational speed. We can also handle NUBS (Non-Uniform B-splines), where the knot vectors are still non-uniform, but the weights W are set to 1.0. Note in the case of B-splines $\Psi = \{\mathbf{P}\}$ (the output from the deep learning framework) and in the case of NUBS $\Psi = \{\mathbf{P}, \mathbf{U}, \mathbf{V}\}$.

3.2. Backward evaluation for NURBS surface

In a modular machine learning system, each computational module requires the gradient of a loss function with respect to the output tensor for the backward computation or the backpropagation. For our NURBS-Diff module this corresponds to $\partial\mathcal{L}/\partial\mathcal{S}$. As an output to the backward pass, we need to provide $\partial\mathcal{L}/\partial\Psi$. While we represent $\mathcal{S}$ for the boundary surface, computationally, we only compute **S** (the set of surface points evaluated from $\mathcal{S}$). Therefore, we would be using the notation of $\partial\mathbf{S}$ instead of $\partial\mathcal{S}$ to represent the gradients with respect to the boundary surface. Here, we assume that with increasing the number of evaluated points, $\partial\mathbf{S}$ will asymptotically converge to $\partial\mathcal{S}$. Now, we explain the computation of $\partial\mathbf{S}/\partial\Psi$ in order to compute $\partial\mathcal{L}/\partial\Psi$ using the chain rule. To explain the implementation of the backward algorithm, we first explain the NURBS derivatives for a given surface point with respect to the different NURBS parameters.

3.2.1. NURBS derivatives

We rewrite the NURBS formulation as follows:

$$\mathbf{S}(u, v) = \frac{\mathbf{NR}(u, v)}{w(u, v)} \tag{6}$$

where,

$$\mathbf{NR}(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij} \mathbf{P}_{ij}$$

$$w(u, v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i^p(u) N_j^q(v) w_{ij}$$

For the forward evaluation of $\mathbf{S}(u, v) = \mathbf{f}(\mathbf{P}, \mathbf{U}, \mathbf{V}, \mathbf{W})$, we can define four derivatives for a given surface evaluation point: $\mathbf{S}_{,u} := \partial\mathbf{S}(u,v)/\partial u$, $\mathbf{S}_{,v} := \partial\mathbf{S}(u,v)/\partial v$, $\mathbf{S}_{,\mathbf{P}} := \partial\mathbf{S}(u,v)/\partial\mathbf{P}$, and $\mathbf{S}_{,\mathbf{W}} := \partial\mathbf{S}(u,v)/\partial\mathbf{W}$. Note that $\mathbf{S}_{,\mathbf{P}}$ and $\mathbf{S}_{,\mathbf{W}}$ are represented as a vector of gradients $\{\mathbf{S}_{,P_{ij}} \forall P_{ij} \in \mathbf{P}\}$ and $\{\mathbf{S}_{,w_{ij}} \forall w_{ij} \in \mathbf{W}\}$.

Now, we show the mathematical form of each of these four derivatives. The first two derivatives are traditionally known as the parametric surface derivatives, $\mathbf{S}_{,u}$ and $\mathbf{S}_{,v}$. Here, $N_{i,u}^p(u)$ refers to the derivative of basis functions with respect to u and v, respectively. These are the standard parametric derivatives, and we do not repeat them here; they are provided in the Appendix for completeness. These derivatives are useful in the sense of differential geometry of NURBS for several CAD applications [42]. However, we do not use it in our module since many deep learning applications such as surface fitting are not dependent on the (u, v) parametric coordinates. Also, note that $\mathbf{S}_{,u}$ and $\mathbf{S}_{,v}$ are not the same as $\mathbf{S}_{,\mathbf{U}}$ and $\mathbf{S}_{,\mathbf{V}}$. The formulation for $\mathbf{S}_{,\mathbf{U}}$ and $\mathbf{S}_{,\mathbf{V}}$ is provided later in this section.

Now, let us define $\mathbf{S}_{,p_{ij}}(u, v)$:

$$\mathbf{S}_{,\mathbf{P}_{ij}}(u, v) = \frac{N_i^p(u) N_j^q(v) w_{ij}}{\sum_{k=0}^{n}\sum_{l=0}^{m} N_k^p(u) N_l^q(v) w_{kl}} \tag{7}$$

where $\mathbf{S}_{,\mathbf{P}_{ij}}(u, v)$ is the rational basis functions themselves. Computing $\mathbf{S}_{,w_{ij}}(u, v)$ is more involved with w_{ij} terms in both the numerator and the denominator of the evaluation.

$$\mathbf{S}_{,w_{ij}}(u, v) = \frac{\mathbf{NR}_{,w_{ij}}(u, v) w(u, v) - \mathbf{NR}(u, v) w_{,w_{ij}}(u, v)}{w(u, v)^2} \tag{8}$$

where,

$$\mathbf{NR}_{,w_{ij}}(u, v) = N_i^p(u) N_j^q(v) \mathbf{P}_{ij}$$

$$w_{,w_{ij}}(u, v) = N_i^p(u) N_j^q(v)$$

For the forward evaluation of $\mathbf{S}(u, v) = \mathbf{f}(\mathbf{P}, \mathbf{U}, \mathbf{V}, \mathbf{W})$, we have defined $\mathbf{S}_{,\mathbf{P}}(u, v)$ and $\mathbf{S}_{,\mathbf{W}}(u, v)$ along with the derivatives $\mathbf{S}_{,u}(u, v)$ and $\mathbf{S}_{,v}(u, v)$. However, computing the $\mathbf{S}_{,\mathbf{U}}(u, v)$ and $\mathbf{S}_{,\mathbf{V}}(u, v)$ is not trivial. $\mathbf{S}_{,\mathbf{U}}(u, v)$ and $\mathbf{S}_{,\mathbf{V}}(u, v)$ refer to the $\partial\mathbf{S}(u,v)/\partial u_i$, $s.t. u_i \in \mathbf{U}$ and $\partial\mathbf{S}(u,v)/\partial v_i$, $s.t. v_i \in \mathbf{V}$. **U** and **V** influence the computation of the basis functions, and these derivatives are helpful for reparameterization of the surfaces by changing the knot vectors. However, due to the recursive computation of the basis functions, the derivatives for **U** and **V** are not defined. Therefore, we need a more rigorous approach for defining the differentiable programming of knot vectors.

First, let us decompose the derivative $\partial\mathbf{S}(u,v)/\partial u_i$, $s.t. u_i \in \mathbf{U}$[1] into the derivative of $\partial\mathbf{S}(u,v)/\partial N_i^p(u)$ and the partial derivative of $\partial N_i^p(u)/\partial u_i$. The derivative $\partial\mathbf{S}(u,v)/\partial N_i^p(u)$ can be easily computed from chain rule as shown here:

[1] We explain the formulation for **U**; a similar formulation exists for **V**.

$$\frac{\partial \mathbf{S}(u,v)}{\partial N_i^p(u)} = \frac{\sum_{j=0}^{m} N_j^q(v) w_{ij} \mathbf{P}_{ij}}{\sum_{r=0}^{n}\sum_{j=0}^{m} N_r^p(u) N_j^q(v) w_{rj}} - \frac{\sum_{r=0}^{n}\sum_{j=0}^{m} N_r^p(u) N_j^q(v) w_{rj} \mathbf{P}_{rj}}{(\sum_{r=0}^{n}\sum_{j=0}^{m} N_r^p(u) N_j^q(v) w_{rj})^2} \left(\sum_{j=0}^{m} N_j^q(v) w_{ij}\right) \tag{9}$$

Now, we evaluate the derivative of $N_i^p(u)$ with respect to the knot points $\{u_i\}$. We observe that due to the recursive nature of the definition, we can accordingly compute the derivatives of $N_i^p(u)$ in a recursive fashion using chain rule, *provided* we can evaluate:

$$\frac{\partial N_i^0(u)}{\partial u_i} = \frac{\partial \mathbf{1}([u_i, u_{i+1}])}{\partial u_i} \tag{10}$$

(and likewise for u_{i+1}) where $\mathbf{1}$ denotes the indicator function over an interval. However, this derivative is not well-defined since the gradient is zero everywhere and undefined at the interval edges.

We propose to approximate this derivative using *Gaussian smoothing* by rewriting the interval as the difference between step functions convolved with deltas shifted by u_i and u_{i+1} respectively:

$$\mathbf{1}([u_i, u_{i+1}))(u) = \text{sign}(u) \star \delta(u - u_i) - \text{sign}(u) \star \delta(u - u_{i+1}) \tag{11}$$

and approximate the delta function with a Gaussian of sufficiently small (but constant) bandwidth:

$$\mathbf{1}([u_i, u_{i+1}])(u) = \text{sign}(u) \star G_\sigma(u - u_i) - \text{sign}(u) \star G_\sigma(u - u_{i+1}) \tag{12}$$

where

$$G_\sigma(u - \mu) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(u-\mu)^2}{2\sigma^2}) \tag{13}$$

The derivative with respect to μ is therefore given by:

$$G'_\sigma(u = \mu) = \frac{(u-\mu)}{2\sigma^2} G_\sigma(u - \mu) \tag{14}$$

which means that the approximate gradient introduces a multiplicative $(u - \mu)$ factor with the original basis function.

Therefore, now we can compute $\partial N_i^p(u)/\partial u_i$ by recursively defining the derivatives $\partial N_i^0(u)/\partial u_i$, $\partial N_i^1(u)/\partial u_i$, until $\partial N_i^{p-1}(u)/\partial u_i$. The derivative of $\partial N_i^p(u)/\partial N_i^{p-1}(u)$ can easily be obtained from chain rule of Eq. (4). Now, we perform the same operations of $\partial N_i^p(u)/\partial u_i \forall u_i \in \mathbf{U}$ to obtain $\partial N_i^p(u)/\partial \mathbf{u}$ and finally obtain $\partial \mathbf{S}(u,v)/\partial \mathbf{u}$. Same operations can be performed to obtain $\partial \mathbf{S}(u,v)/\partial \mathbf{v}$. With all these operations for each point parameterized in the surface by (u, v), we extend this to all the surfaces as explained in the next section.

3.2.2. Jacobian for surface evaluation

We define the Jacobian for the NURBS evaluation, which is then directly used for the backward evaluation. The Jacobian for each surface evaluation point $S_{i,j}$ is represented as the vector:

$$\mathbf{B}_{i,j} = \begin{pmatrix} \{\mathbf{S}_{,p_{ij}}(u,v)\} \forall i \in [1,m], \forall j \in [1,n] \\ \{\mathbf{S}_{,w_{ij}}(u,v)\} \forall i \in [1,m], \forall j \in [1,n] \\ \{\mathbf{S}_{,u_i}(u,v)\} \forall u_i, \in \mathbf{U} \\ \{\mathbf{S}_{,v_j}(u,v)\} \forall v_j, \in \mathbf{V} \end{pmatrix} \tag{15}$$

Each Jacobian vector represented here is the contribution of the gradient from one evaluation point at the grid locations (i, j) in the parametric coordinate space. These Jacobian vectors are each of length 4 nm. However, as noted in the previous section on forward evaluation, the basis functions satisfy the partition of unity and span only $p + 1$ control points starting from u_{span} (correspondingly, $q + 1$ control points starting from v_{span} in the other parametric direction). Therefore, the total number of non-zero elements in a 4 nm size Jacobian vector is $4(p + 1)(q + 1)$, making it sparse. However, note that this Jacobian is for only one surface point. The complete Jacobian for the backward pass is given as:

$$J = \begin{pmatrix} \mathbf{B}_{1,1} \\ \mathbf{B}_{1,2} \\ \vdots \\ \mathbf{B}_{m_{grid}, n_{grid}} \end{pmatrix}. \tag{16}$$

The size of this Jacobian is $n_{grid} m_{grid} \times 4$ nm. Here, $\mathbf{B}_{i,j}$ is the Jacobian for one surface point evaluation. As the parametric coordinates keep changing, the position of u_{span} and v_{span} keep changing, and the location of the non-zero elements keeps shifting to form a block diagonal matrix. This Jacobian is $\partial \mathbf{S}/\partial \Psi$. For completing the backward pass, we multiply $\partial \mathcal{L}/\partial \mathbf{s}$ to $\partial \mathbf{S}/\partial \Psi$, giving us $\partial \mathcal{L}/\partial \Psi$. Since, each module in the deep learning framework is independent and modular, we just return this output for the NURBS backward evaluation.

3.2.3. Implementation

For the implementation of the backward pass, since the basis functions are block sparse, we make use of the stored information of u_{span} and v_{span} for identifying the index of the control points derivative and we use the stored basis functions information for computing the Jacobian explained above. This computation is performed for all the surfaces in the batch. This complete algorithm is explained in detail in Algorithm 2.

Algorithm 2: Backward Algorithm

Input : $\mathbf{S}'$
Output: $\mathbf{P}'$, $\mathbf{W}'$
Initialize: $\mathbf{P}' \rightarrow 0$
Initialize: $\mathbf{W}' \rightarrow 0$
for $k = 1 : surfaces$ **do**
  for $j = 1 : m_{grid}$ **do**
    for $i = 1 : n_{grid}$ **do**
      Retrieve u_{span}, v_{span}, N_i^p, N_j^q, $\mathbf{S}(u, v)$
      for $r = 0 : p + 1$ **do**
        for $h = 0 : q + 1$ **do**
          $\mathbf{P}'_{u_{span}+r, v_{span}+h} = \mathbf{S}_{,p_{ij}}(u_i, v_j)$
          $\mathbf{W}'_{u_{span}+r, v_{span}+h} = \mathbf{S}_{,w_{ij}}(u_i, v_j)$
          $\mathbf{U}'_{u_{span}+r} = \mathbf{S}_{,u_{span}+r}(u_i, v_j)$
          $\mathbf{V}'_{v_{span}+h} = \mathbf{S}_{,v_{span}+h}(u_i, v_j)$

3.3. GPU implementation

We implemented the code in *Python 3.6* [43]. The backend for the GPU-accelerated code is written in *C++* using the Pybind11 API [44] and CUDA toolkit [45] for GPU acceleration and is integrated with PyTorch [46] using a custom layer definition. The forward evaluation can be performed for each surface in the batch for each tuple (u, v) in the mesh grid of $u_{grid} \times v_{grid}$ in parallel. Further, the three coordinates x, y, z are evaluated simultaneously. This enables an embarrassingly parallel implementation on the GPU for the forward evaluation of the *NURBS-Diff* module. Each x, y, and z component is mapped to a separate thread on the GPU using the 3D block and grid structure in CUDA. The same process is employed in the backward algorithm with one additional operation. Each surface point gradient needs to be added to several control points that lie in the evaluated point's span during the backward pass. Hence we perform this operation of the gradient update using a `scatter` operation by using the indices stored from u_{span} and v_{span}.

4. CAD applications using the *NURBS-Diff* module

The differentiable programming approach explained above is designed mainly for deep learning applications. However, we can also use the framework for standard CAD operations, such as curve fitting, surface fitting, and surface offsetting. Note that some of these operations could be performed much faster using traditional approaches explicitly optimized for each application. However, using *NURBS-Diff* along with gradient-descent-based optimization approaches for these CAD applications is not well explored. Moreover, this shows the versatility of the *NURBS-Diff* module in handling traditional CAD operations as constraints in a deep learning system.

To use our differentiable programming approach for CAD applications, we have to define two key elements: a loss function $\mathcal{L}$ for computing the gradients and an optimization algorithm. We consider four loss functions: $\mathcal{L}_1$ loss, $\mathcal{L}_2$ loss (also called as mean squared error), the Chamfer distance $\mathcal{L}_{CD}$, and the Hausdorff distance $\mathcal{L}_{HD}$.

The $\mathcal{L}_1$ loss can be mathematically defined as:

$$\mathcal{L}_1(\mathbf{P}, \mathbf{Q}) = \frac{1}{n_{points}} \Big(\sum_{(\mathbf{P_i}, \mathbf{Q_i}) \in (\mathbf{P}, \mathbf{Q})} \|\mathbf{P_i} - \mathbf{Q_i}\|_1 \Big) \tag{17}$$

Here, $\|\mathbf{P_i} - \mathbf{Q_i}\|_1$ refers to the L_1 norm of the difference between the two points $\mathbf{P_i}$ and $\mathbf{Q_i}$. Similarly, we can define $\mathcal{L}_2$ loss based on the L_2 norm.

$$\mathcal{L}_2(\mathbf{P}, \mathbf{Q}) = \frac{1}{n_{points}} \Big(\sum_{(\mathbf{P_i}, \mathbf{Q_i}) \in (\mathbf{P}, \mathbf{Q})} \|\mathbf{P_i} - \mathbf{Q_i}\|_2 \Big) \tag{18}$$

While both the $\mathcal{L}_1$ and $\mathcal{L}_2$ loss functions are pairwise distance metrics, the Chamfer distance ($\mathcal{L}_{CD}$) and the Hausdorff distance ($\mathcal{L}_{HD}$) are global distance metrics between two sets of points as shown below:

$$\mathcal{L}_{CD} = \sum_{\mathbf{P_i} \in \mathbf{P}} \min_{\mathbf{Q_j} \in \mathbf{Q}} \|\mathbf{P_i} - \mathbf{Q_j}\|_2 + \sum_{\mathbf{Q_j} \in \mathbf{Q}} \min_{\mathbf{P_i} \in \mathbf{P}} \|\mathbf{P_i} - \mathbf{Q_j}\|_2 \tag{19}$$

$$\mathcal{L}_{HD} = \max_{\mathbf{P_i} \in \mathbf{P}} \left(\min_{\mathbf{Q_j} \in \mathbf{Q}} \|\mathbf{P_i} - \mathbf{Q_j}\|_2 \right) + \max_{\mathbf{Q_j} \in \mathbf{Q}} \left(\min_{\mathbf{P_i} \in \mathbf{P}} \|\mathbf{P_i} - \mathbf{Q_j}\|_2 \right) \tag{20}$$

For each CAD application, a target point cloud $\mathbf{Q}$ is obtained directly from measurements (for the case of fitting from point clouds) or from analytical computations (for CAD operations such as surface offsetting). While our formulation works well when we initialize the control points and weights to random values (Gaussian distributed), we can also initialize it using a prior for faster convergence. After we initialize the control points $\mathbf{P}$ and weights $\mathbf{W}$, we initialize the knot vectors $\mathbf{U}$ and $\mathbf{V}$. While the knot vectors are fixed in most applications we demonstrate, we show one example where we could reparameterize the surface to obtain a better fit. We evaluate the surface using $\mathbf{P}$, $\mathbf{W}$, $\mathbf{U}$, and $\mathbf{V}$ and compute the loss between the evaluated surface $\mathbf{S}$ and $\mathbf{Q}$ using the appropriate loss function $\mathcal{L}$. We perform an update using gradient descent algorithms and their variants. We can write a simple update for the NURBS parameters as:

$$\Psi = \Psi - \alpha \frac{\partial \mathcal{L}}{\partial \Psi} \tag{21}$$

While our formulation can use a simple gradient descent algorithm, in our work, we use more sophisticated algorithms such as stochastic gradient descent (SGD), SGD with momentum, Adam [47], and Adagrad [48]. Our experiments illustrated in the Appendix show that SGD with momentum and Adam perform well in all scenarios and have faster convergence. Now, we discuss the specific CAD applications using our *NURBS-Diff* module.

Table 1
Performance ($\mathcal{L}_{CD}$) of fitting different curves using the *NURBS-Diff* module with and without curve length regularization.

Curve	No regularization	Regularization
Analytical	0.0025	**0.0016**
Helix	0.0370	**0.0320**
Apple	8.5267	**1.6863**
Flower	23.2484	**0.8383**
Bunny	50.0020	**1.4099**

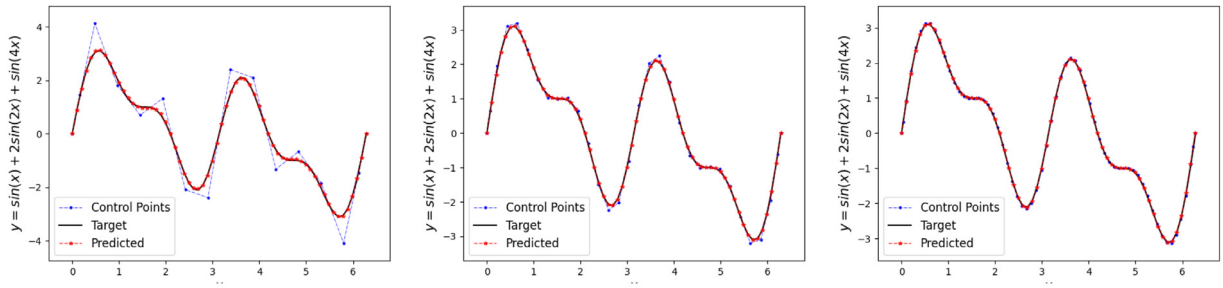
4.1. Curve fitting

We first demonstrate curve fitting using our *NURBS-Diff* module. While the problem of fitting splines to point clouds using data-driven techniques has been extensively studied, we show curve fitting to validate our approach and provide insights (such as the convergence of the optimization, behavior of the fit with the variation in control points, evaluation points, etc.). In this validation case, we initialize a uniform knot vector $\mathbf{U}$ (which is kept fixed) and compute the non-zero basis functions $N(u)$ and u_{span} as a pre-computation step before evaluating the points on the curve. This pre-computation step reduces the computational time significantly (due to the removal of repeated basis function evaluations). This approach can be used when we are trying to fit a B-spline curve/surface where the basis functions do not change or if the user does not want to change the parameterization of the curve or surface.

For a simple validation, we sample points from an analytically defined curve $y = sin(x)+2sin(2x)+sin(4x)$ and obtain the best-fit curves for different control points and evaluation points as shown in Fig. 2. Similarly, for points sampled from a 3D helical curve, we fit a B-Spline curve as shown in Fig. 3. The behavior of our fitting method with different loss functions and the number of control points is shown in Fig. 4. While all the loss functions converge well with the number of iterations, $\mathcal{L}_2$ loss and $\mathcal{L}_{CD}$ losses have better convergence characteristics. $\mathcal{L}_{CD}$ loss plateaus after some iterations since that is the best error that could be achieved for a given number of control points and evaluation points. Please note that the $\mathcal{L}_{CD}$ also reduces the oscillations that might occur in fitting a higher degree curve.

We extend the curve fitting framework to a more general fitting of random unordered point cloud data. For the 2D point cloud data, we use the images from the Pixel dataset of the Skelneton challenge [49] and sample points from the object boundaries as shown in Fig. 5. Since the problem is ill-posed (due to the unordered aspect of the point cloud), we initialize the control points using randomly sampled points from the point cloud. To avoid unnecessary loops and self-intersections in our output curve, we use a curve-length regularization term in addition to the $\mathcal{L}_{CD}$. As shown in Fig. 5 our framework fits the target well for point clouds that are not excessively complex. Some complex point cloud data that had self-intersections are illustrated in Appendix.

We also analyze the performance of our fitting method for different curves in Table 1. For a consistent comparison, we use the $\mathcal{L}_{CD}$ between a dense set of points evaluated on the fitted curve (using 16 control points) and the input point cloud. We set the number of evaluation points to be twice the number of points in the input. Our *NURBS-Diff* module achieves better fitting results with the curve length regularization added. While this difference is less evident for simple analytical curves, the advantage of using a curve length regularization is more pronounced over curves fitted using the Pixel dataset.



(a) *16 control points, 64 evaluation points* **(b)** *32 control points, 64 evaluation points* **(c)** *64 control points, 64 evaluation points*

Fig. 2. Curve fitting to points from an analytically generated curve $y = sin(x) + 2sin(2x) + sin(4x)$ with different number of control points.

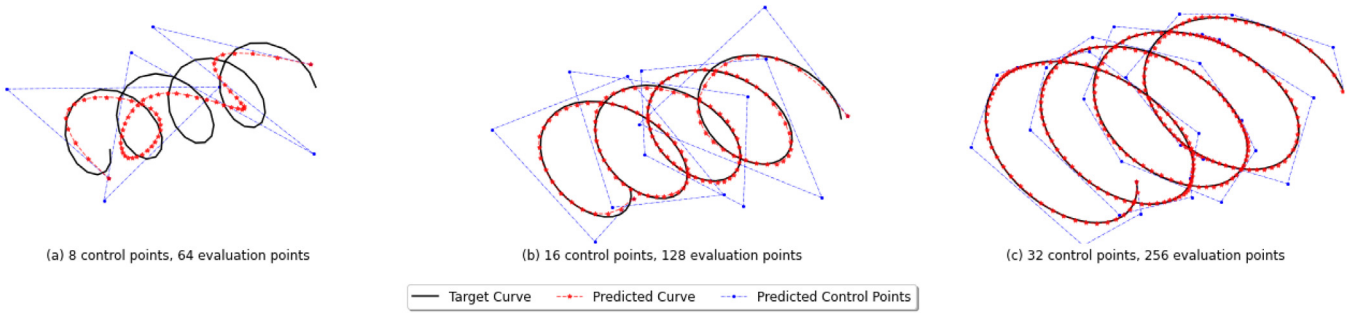


Fig. 3. Curve fitting for points sampled from a 3D helical curve in $\mathcal{R}^3$ for different numbers of control points and evaluated points.

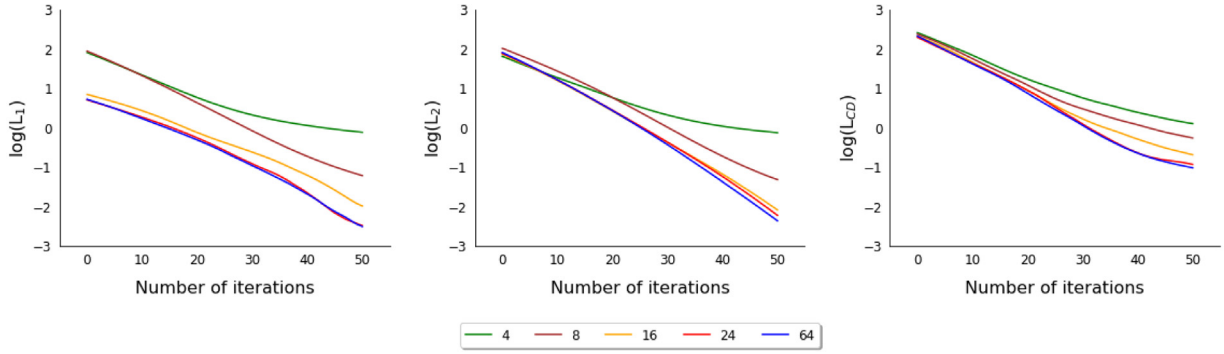


Fig. 4. The loss performance with the number of iterations for fitting the 3D helical curve shown in Fig. 3. The behavior of the loss is shown for different loss functions and different numbers of control points used for the fitting.

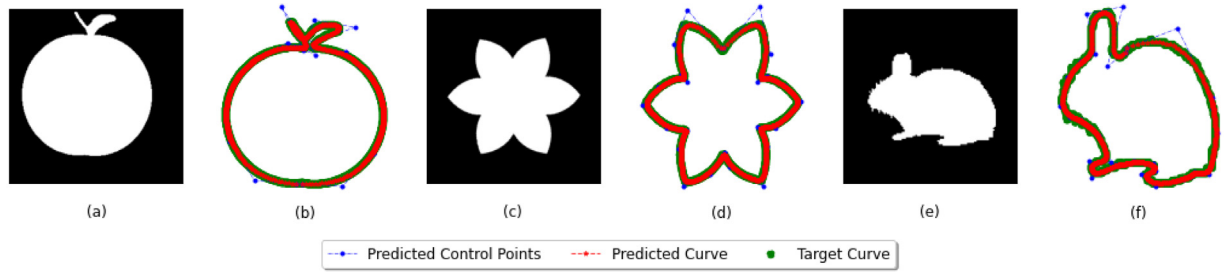


Fig. 5. Curve fitting on point cloud data obtained from binary images. We convert the image data into a point cloud using the pixel size and the locations of the pixels. Once we obtain the point cloud, we use a curve-fitting module with Chamfer distance loss function and regularization to obtain these results.

4.2. Surface fitting

We extend the validation of our *NURBS-Diff* module to the fitting of NURBS surfaces. In this case, we consider two scenarios for testing the fitting process. In the first scenario, we keep the knot vectors **U** and **V** fixed. In this scenario, we can precompute the basis functions $N_i^p(u)$ and $N_j^q(v)$ along with their respective span indices, u_{span} and v_{span} . In the second scenario, we allow the knot vectors to be changed, leading to the reparameterization of the surface to obtain a better fit. We cannot precompute the basis functions in the second scenario since the knot vector is updated each iteration. We define a set of points sampled from the analytical surface $z = xysin(x)cos(y)$ to compare both scenarios. The control points and weights are initialized randomly for the

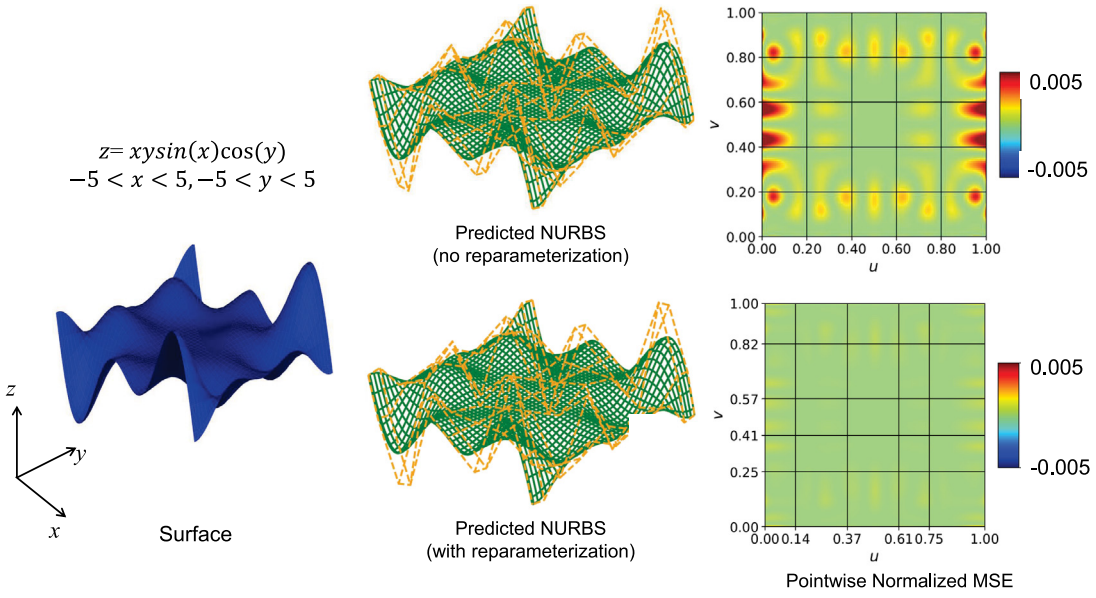


Fig. 6. Surface fitting using NURBS for points sampled from an analytical surface $z = sin(x) * cos(y)$.

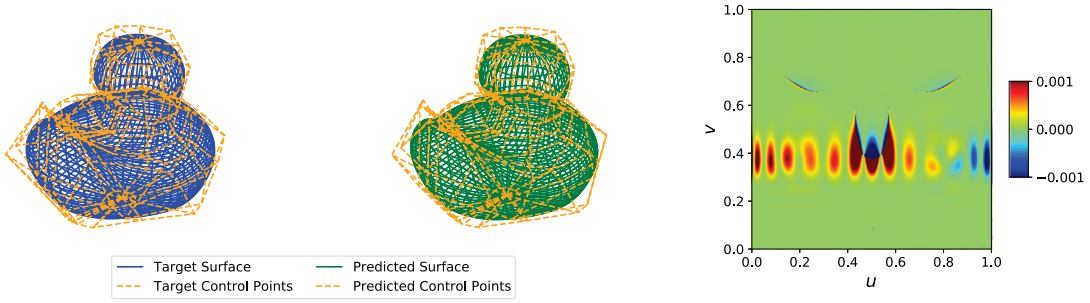


Fig. 7. NURBS surface fitting for point cloud representation of Ducky's body.

Table 2
Performance $\mathcal{L}_2$ of the different surface CAD applications using the *NURBS-Diff* module. B-Spline refers to fixed uniform knot vector, and NURBS refers to non-uniform knot vector obtained through optimization.

Test case	B-spline	NURBS
Analytical	0.039528	**0.005262**
Ducky	0.000195	**0.000180**

fitting, while the knot vectors are initialized as uniformly spaced. Using the $\mathcal{L}_2$ loss for the optimization, we perform the fitting for both the scenarios as shown in Fig. 6. While the first scenario provides a good fit, we can reduce the minor oscillation errors in the surface fit with the knot optimization to obtain an order of magnitude lower $\mathcal{L}_2$ loss for the second scenario, as shown in Table 2. Note that both the surface fits look similar and fit the target surface well, but the surface reparameterization reduces the overall error. The changes in the knot vector are depicted using the gridlines on the heatmap showing the error.

We extend this surface fitting to a more complicated surface (the Ducky shown in Fig. 7). We have the target control points, weights, and knot vectors for this geometry. Using this information, we sample points from the geometry uniformly and then use that as a target for performing the surface fitting. We only show the results for the best fit surface with knot vector optimization for brevity. We observe improvement in the fit by performing knot optimization as shown in Table 2. However, in this case, the improvement is not as pronounced as seen in the analytical geometry. Finally, we study the performance of our *NURBS-Diff* module for fitting the analytical surface shown in Fig. 6 using a variety of control point sizes, evaluation point sizes, and degrees. To benchmark the performance of our module, we study the first 500 iterations of the optimization (both forward and backward pass) and report the $\mathcal{L}_2$ loss. The results of these experiments are highlighted under Tables 3–5. We observe that 12 control points are required to fit the surface properly. As expected, increasing the number of control points reduces the $\mathcal{L}_2$ error. For the number of evaluation points, a similar trend is observed, where increasing the number of evaluation points reduces the $\mathcal{L}_2$ error. Similarly, the $\mathcal{L}_2$ error decreases as the degree of the surface is increased until it stabilizes, after which it shows a slight increase in the error probably due to overfitting oscillations. Particularly, in Table 5, we notice that the error is higher with a lower degree due to being an underdetermined system. With increasing the degree, the fit improves, and then finally, at degree 4, the system becomes overdetermined and hence overfits.

4.3. Surface offsetting

Generating an offset surface is one of the fundamental CAD operations. Traditionally an offset surface for NURBS is generated by first performing a Bèzier decomposition of the NURBS surface and performing the offset for each patch. However, this changes the parameterization of the resulting offset surfaces. However, specific applications might require an offset surface with the same parameterization. We can easily use our *NURBS-Diff* module

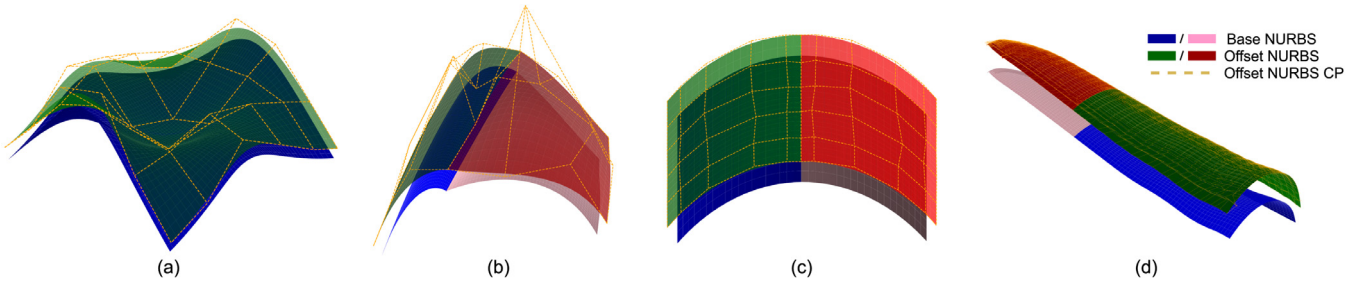


Fig. 8. Test cases for NURBS surface offset (a) Double curved surface, (b) C^0 continuous multi patch, (c) C^1 continuous multi patch, and (d) C^1 continuous patches of the aerofoil profile of a wind turbine blade.

Table 3
$\mathcal{L}_2$ error for fitting different control point mesh sizes for the analytical surface with 128 × 128 evaluation points and surface degree 3.

Control points	$\mathcal{L}_2$ error
6 × 6	$2.2719 \times 10^{+1}$
9 × 9	3.9528×10^{-2}
12 × 12	6.9547×10^{-4}
24 × 24	3.7953×10^{-7}
48 × 48	1.0997×10^{-7}

Table 4
$\mathcal{L}_2$ error for fitting a 9 × 9 control mesh to the analytical surface for degree 3 with different numbers of evaluated points.

Evaluation points	$\mathcal{L}_2$ error
64 × 64	0.0402
128 × 128	0.0395
256 × 256	0.0391
512 × 512	0.0389

to perform such an offset operation. Note that this approach only works for small offset distances such that the topology of the offset surface does not change.

To perform the offset operation, we first compute a dense set of points and normals at specific (u, v) and calculate the points on the corresponding offset surface by moving the points along the normal by the offset distance. We then fit a NURBS surface for the offset point cloud data using the same parameterization of the base surface using the surface fitting method (see Section 4.2).

Using our offsetting method, we can also generate offsets of objects consisting of multiple NURBS surfaces. Generating offset surfaces from multiple base surfaces requires applying constraints on the control points on the common edges of the base surfaces. While computing the surface normals, we identify the surface points along the shared edge. We then compute the average normals of these common points and then normalize them. Calculating the average normal allows us to ensure the continuity of the offset point cloud data. We then apply the constraints on the fitted control points of the common edges to be the same on both the surfaces that share the common edge. To ensure this continuity for the NURBS surfaces, we create a list of the shared control points. After the surface fitting iteration, the control points are updated; based on the shared list, i.e., the control points of the common edge are assigned the same coordinates. We perform this by computing the average of all the common points and setting them the same on both surfaces.

Fig. 8 shows examples of the surface offsets generated using our *NURBS-Diff* module. Fig. 8(a) is a single surface patch with a double-curved surface. Fig. 8(b) is a set of C^0 continuous surfaces with a single shared edge. Fig. 8(c) is a set of C^1 continuous conic section surfaces. Fig. 8(d) is a set of surfaces from a wind turbine blade model. To compare the accuracy of our *NURBS-Diff* module approach for offset surfaces, we computed a simpler offset by offsetting the control points along the average normal direction of the control mesh. This approach works well for surfaces with low curvature. We then evaluated the computed offset surface at 25× denser points than the number of points used for fitting and computed the $\mathcal{L}_{CD}$ with the input offset points. We find that our fitting approach achieves a lower $\mathcal{L}_{CD}$ than the control point offset approach for all cases, as seen in Table 6.

Table 5
$\mathcal{L}_2$ error for different degrees of a 9 × 9 control mesh to the analytical surface using 128 × 128 evaluation points.

Degree	$\mathcal{L}_2$ error
1	0.3615
2	0.0702
3	0.0395
4	0.0585

4.4. Timings for NURBS-Diff

In the previous sections, we demonstrate how the *NURBS-Diff* module can perform CAD operations such as curve fitting, surface fitting, and surface offsetting. In this section, we assess the computational performance of our module. For brevity, we restrict our analysis to surface fitting operation and analyze the timings with variations in the number of control points, evaluation points, and surface degree. We only study the first 500 iterations (which include both the forward and backward pass). We perform all our experiments on a desktop with a 32 core 2.4 GHz Intel Xeon processor, 64 GB RAM, and an NVIDIA Titan Black GPU with 6 GB RAM.

We analyze the timings against the different number of control points as shown in Table 7. We begin with the minimum number of control points we would get meaningful surfaces for the given experiment (i.e., 6 × 6 control points). We observe that the timings increase with the number of control points. Variations in iteration times for different evaluation point sizes are shown in Table 8. Similar to Table 7, there is a steady increase in iteration times with an increase in the number of evaluation points. Note that, while the iteration time is increasing drastically (especially when going from 256 × 256 to 512 × 512), the performance of the fit does not improve much, as seen in Table 4. Therefore, the end-user must judiciously choose the number of evaluation points sampled for the *NURBS-Diff* module to get an accurate fit of the surface while not increasing the computational time. Finally, increasing the degree also increases the time required to fit a surface, as shown in Table 9. However, for most cases, it can be seen the *NURBS-Diff* module can perform 5–10 iterations per second, which makes it tractable for fitting a large number of surfaces.

Table 6
Normalized Chamfer distance between the offset surface and the offset point cloud using our *NURBS-Diff* module and SGD compared to a direct offset of the control points (CP offset). The Chamfer distance is normalized using the minimum size of the bounding box of the base surface along the offset direction.

Test case	Min BB size	Offset distance	$\mathcal{L}_{CD}$ *NURBS-Diff* offset	$\mathcal{L}_{CD}$ CP offset
Double curve	11.54	1.50	**0.0235**	0.0239
Multi patch - C^0	0.75	0.10	**0.0006**	0.0008
Multi patch - C^1	6.32	2.00	**0.0389**	0.0390
Aerofoil surface	0.66	0.25	**0.0529**	0.0545

Table 7
Time to fit a surface for different numbers of control points.

Control points	Iteration time (s)
6 × 6	0.098
12 × 12	0.106
24 × 24	0.110
48 × 48	0.110

Table 8
Time to fit a surface for different numbers of evaluation points.

Evaluation points	Iteration time (s)
64 × 64	0.074
128 × 128	0.120
256 × 256	0.170
512 × 512	0.266

Table 9
Computation time to fit a surface of different degrees.

Degree	Iteration time (s)
1	0.074
2	0.120
3	0.170
4	0.266

5. Point cloud reconstruction

In this section, we show the utility of the *NURBS-Diff* module for unsupervised point cloud reconstruction. We use the experiments performed by Sharma et al. [41] as our baseline, which is a supervised learning framework for surface reconstruction framework. Sharma et al. [41] introduced an end-to-end trainable network called ParSeNet that fits an assembly of geometric primitives, including B-spline patches, to a segmented point cloud. In the ParSeNet framework, the authors develop a spline fitting module (called SplineNet) which takes an input point cloud and reconstructs a spline surface. This surface, however, is obtained using a supervised learning approach, as we highlight later. Therefore, to improve the framework and obtain a better fit without the supervised labels, we integrate our *NURBS-Diff* module with SplineNet to develop an unsupervised training approach for spline fitting.

The ParSeNet framework is divided into three stages. The first stage incorporates prior work done in point cloud segmentation [50] to decompose the input point cloud into segments classified under a parametric patch type. The second stage is the spline fitting SplineNet that generates B-spline patches to the segmented point cloud data. The final stage performs geometric optimizations to seamlessly stitch the collection of predicted primitives together into a single object. We are interested in replacing the surface evaluation performed in the SplineNet stage of their network with our *NURBS-Diff* module for the experiments in this section. For training and testing our experiments, we use the SplineDataset provided by Sharma et al. [41]. The SplineDataset is a diverse collection of open and closed splines that have been extracted from one million CAD geometries included in the ABC dataset. A random set of points is sampled from the surfaces as the input for the point cloud reconstruction task. We run our experiments on open splines split into 24 K, 4 K, and 4 K for training, testing, and validation.

In our work, we focus only on the SplineNet module of the ParSeNet framework to illustrate the applicability of our proposed *NURBS-Diff* module. SplineNet is a module for performing supervised learning of B-spline surfaces using the SplineDataset. The training procedure includes a loss definition as follows:

$$\mathcal{L}_{SplineNet} = \mathcal{L}_{CD} + \lambda_1 \mathcal{L}_{LapMat} + \lambda_2 \mathcal{L}_{CP}. \tag{22}$$

Here, $\mathcal{L}_{CD}$ refers to the Chamfer distance between the input point cloud and the surface points evaluated on the target surface. $\mathcal{L}_{LapMat}$ is the Laplacian matching loss where the difference in the Laplacian (the second derivative of the control points mesh obtained using a Sobel filter) of the fitted control points and the target control points is minimized. $\mathcal{L}_{CP}$ is the control point regression loss which is $\mathcal{L}_2$ error between the predicted and the target control points. λ_1 and λ_2 are used for weighting different loss functions. For brevity, we use the same values for these constants as reported in Sharma et al. [41].

In the SplineNet approach, the $\mathcal{L}_{LapMat}$ and the $\mathcal{L}_{CP}$ make the framework supervised since they require a target fitted control point mesh to compute these losses. In real life, obtaining the target control point mesh for point clouds is challenging. Therefore, we need an unsupervised learning approach for point cloud reconstruction of spline surfaces. To make the framework unsupervised and not require the target control points of the spline surface, we modify the framework to the following:

$$\mathcal{L}_{NURBS\text{-}Diff} = \mathcal{L}_{CD} + \lambda_1 \mathcal{L}_{Lap} + \lambda_2 \mathcal{L}_{HD}. \tag{23}$$

Here, $\mathcal{L}_{CD}$ and $\mathcal{L}_{HD}$ refer to the Chamfer distance and the Hausdorff distance between the input point cloud and the surface points evaluated using the *NURBS-Diff* module. $\mathcal{L}_{Lap}$ is a modified loss of $\mathcal{L}_{LapMat}$ where instead of matching the Laplacian of the predicted control points and actual control points, we minimize the Laplacian itself. Further, we scale down the contribution of $\mathcal{L}_{Lap}$ by an order of magnitude to reduce any adverse effects from minimizing the Laplacian. In addition, to fit the edges of the surfaces well, we add a Hausdorff distance loss to the objective function. We use different scale factors λ_1, λ_2 for $\mathcal{L}_{Lap}$ and $\mathcal{L}_{HD}$ to tune the objective function to obtain the best possible results.

Apart from the loss function defined above, the baseline supervised approach is restricted to non-rational B-spline surfaces because of no target weights. Here, in our case, we can perform the fitting for rational B-Spline surfaces or even B-Spline surfaces with reparameterized knots. Since the dataset available is very primitive and does not contain many complex structures, we restrict our analysis to just studying rational B-Splines reconstruction from the point clouds in an unsupervised manner.

Table 10 illustrates all the experiments we have performed for choosing an appropriate loss function. Performing surface reconstruction under no supervision is very challenging, and therefore several metrics have to be compared together to understand the performance. In Fig. 9, we illustrate how $\mathcal{L}_{CD}$ alone provides a bad fit by not covering the edges of the surfaces. Similarly, $\mathcal{L}_{HD}$ alone does not perform well. Therefore, we need a loss function that performs well in combination. We compare the results

Table 10
Comparison between SplineNet [41] and our *NURBS-Diff* implementation (with different number of control points). We compare the two-sided Chamfer distance (scaled by 100, $100\times\mathcal{L}_{CD}$), the two-sided Hausdorff distance (scaled by 100, $100\times\mathcal{L}_{HD}$) between the input point cloud and a dense set of points sampled on the fitted surface, and also the Laplacian of the predicted control points $\mathcal{L}_{Lap}$ (scaled by 100, $100\times\mathcal{L}_{Lap}$).

Loss function	Baseline (20 × 20)			*NURBS-Diff* (20 × 20)			*NURBS-Diff* (5 × 5)			*NURBS-Diff* (4 × 4)		
	$\mathcal{L}_{CD}$	$\mathcal{L}_{HD}$	$\mathcal{L}_{Lap}$	$\mathcal{L}_{CD}$	$\mathcal{L}_{HD}$	$\mathcal{L}_{Lap}$	$\mathcal{L}_{CD}$	$\mathcal{L}_{HD}$	$\mathcal{L}_{Lap}$	$\mathcal{L}_{CD}$	$\mathcal{L}_{HD}$	$\mathcal{L}_{Lap}$
$\mathcal{L}_{CD}+\mathcal{L}_{LapMat}+\mathcal{L}_{CP}$	1.184	0.994	1.340									
$\mathcal{L}_{CD}+0.1\mathcal{L}_{Lap}+10\mathcal{L}_{HD}$				**0.018**	**0.234**	**0.202**	**0.020**	**0.256**	**2.462**	**0.028**	**0.274**	**3.356**
$\mathcal{L}_{CD}+0.1\mathcal{L}_{Lap}+\mathcal{L}_{HD}$				0.027	0.725	0.106	0.032	0.805	1.972	0.047	1.173	3.606
$\mathcal{L}_{CD}+0.1\mathcal{L}_{Lap}$				0.098	2.914	0.189	0.105	3.511	1.296	0.113	3.602	2.637
$\mathcal{L}_{CD}$				0.012	0.329	42.288	0.013	0.446	21.236	0.015	0.499	28.212
$\mathcal{L}_{HD}$				0.018	0.192	30.877	0.018	0.210	23.076	0.026	0.282	32.443

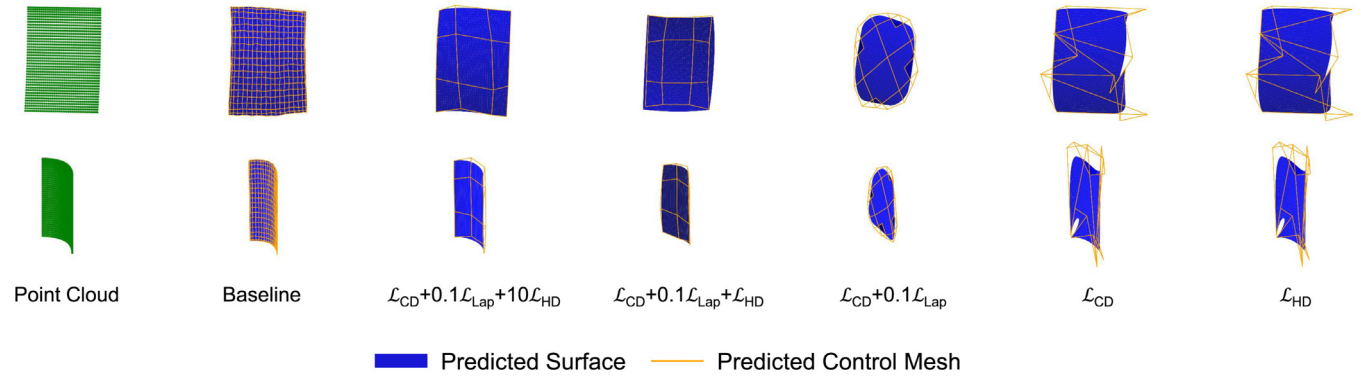


Fig. 9. Comparison of the reconstructed B-spline surfaces obtained using *NURBS-Diff* module for different loss functions using a 5 × 5 control mesh.

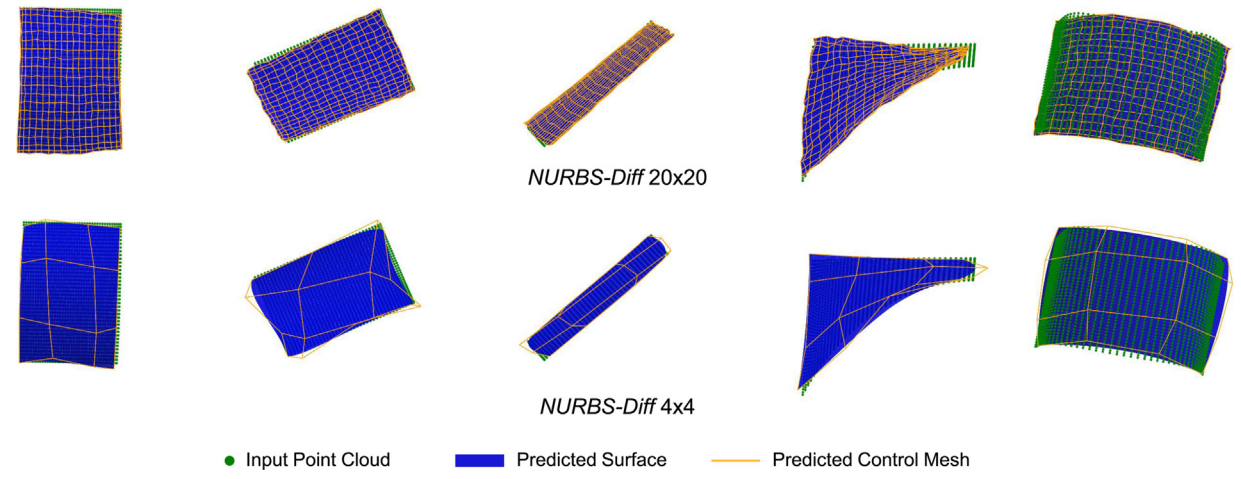


Fig. 10. Point clouds and unsupervised reconstruction of B-spline surfaces (20 × 20 and 5 × 5) predicted by SplineNet using the *NURBS-Diff* module.

obtained from our approach with the baseline results obtained from Sharma et al. [41], as illustrated in Table 10. We observe that our approach performs an order of magnitude better than the baseline architecture in terms of $\mathcal{L}_{CD}$. Further, unlike Table 3 where we have a complex analytical surface, the SplineDataset includes a collection of simpler open and closed surfaces. Therefore, we observe that reducing the number of control points required for representing the surface does not affect the fitting error as demonstrated by the (5 × 5) and (4 × 4) columns in Table 10. Also, we note that using just $\mathcal{L}_{CD}$ and $\mathcal{L}_{HD}$ is not recommended due to high laplacian loss. Further, the combination of $\mathcal{L}_{CD}+0.1\mathcal{L}_{Lap}+10\mathcal{L}_{HD}$ gives the best result in terms of combined loss. We can also verify the same visually from Fig. 9. We also visualize a few anecdotal predicted surfaces along with the input point cloud in Fig. 10. We see that the surfaces in the SplineNet dataset are not complex enough to require a large 20 × 20 control point mesh and can be easily fitted with a small 4 × 4 control point mesh.

6. Geometric constraints using *NURBS-Diff*

One of the advantages of our *NURBS-Diff* module is the ease of enforcing surface constraints in deep-learning applications. We showcase the utility of our module in enforcing constraints using the example of valve deformation analysis. Analyzing bioprosthetic heart valves is essential for obtaining diagnostic information such as estimating the remaining life, fatigue, and patient-specific design. Traditionally, analysis of deformation behavior is performed using finite element or isogeometric analysis. However, such analyses are often computationally intensive. Recently, Balu et al. [40] proposed a deep learning framework for performing finite element analysis (called DLFEA) using a NURBS-aware convolutional neural network. Each heart valve is represented using three NURBS surface patches, and isogeometric analysis is performed to obtain the deformations for each control point under constant pressure applied during the valve closure.

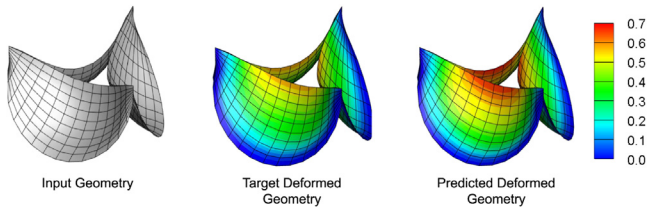


Fig. 11. Visualization of an anecdotal bioprosthetic heart valve for DLFEA-BC from the test dataset. We show the input geometry, the target deformed geometry, and the predicted deformed geometry.

Table 11
Performance comparison of DLFEA, DLFEA-SR, and DLFEA-BC. We compare the error on the test dataset for control points reconstruction, NURBS surface reconstruction, and boundary condition enforcement.

Test case	CP loss	SR loss	BC loss
DLFEA	7.79×10^{-3}	4.47×10^{-3}	37.69
DLFEA-SR	12.84×10^{-3}	4.76×10^{-3}	46.14
DLFEA-BC	12.83×10^{-3}	5.69×10^{-3}	10.90

Fixed boundary conditions are applied on the valve edges that will be sutured to the aorta. Their previous work generated a large dataset of input geometry, pressure, thickness, and corresponding target deformations of the control points. However, their work does not perform a NURBS surface reconstruction loss for obtaining the deformations, which are more physically meaningful. Further, they did not explore the application of the fixed boundary conditions; it was implicitly enforced by modifying the loss function.

In this work, we use the dataset and deep learning framework available from their work to demonstrate the utility of our *NURBS-Diff* module in performing an evaluation of the deformations for the leaflets of the bioprosthetic heart valve. We perform a comparison between DLFEA, DLFEA with additional surface reconstruction loss (DLFEA-SR), and DLFEA with additional surface reconstruction loss and boundary condition enforcement loss (DLFEA-BC). We show the visualization of DLFEA-BC along with the input geometry, the target deformed geometry, and the predicted deformed geometry in Fig. 11. In DLFEA-BC, apart from the surface reconstruction loss, we add a constraint (BC loss) that the Dirichlet boundary conditions on the edge which is sutured to the aorta must be satisfied (i.e., the edges closest to the blue region in Fig. 11 must have zero deformation).

In Table 11, we show the results obtained on a test dataset (not used during the training). The CP loss is the $\mathcal{L}_2$ loss between the input control points and the target control points, whereas the surface reconstruction (SR) loss represents the $\mathcal{L}_2$ loss between the NURBS surface reconstruction of the target deformed shape and the actual deformed shape. We observe that DLFEA performs very well for CP loss (naturally because it was originally trained using that loss) but does not do well on the BC loss. The DLFEA-SR performs comparably for SR loss but has worse performance for BC loss. At the same time, DLFEA-BC performs the best for BC loss while performing comparably (although worse) on CP loss and SR loss. While the DLFEA-BC performs worse, the enforcement of boundary conditions makes it more physically meaningful. Further, in this ablation study, we show the result for each loss function applied independently. In general, we use a combined loss with several loss functions in tandem as done in Section 5.

7. Conclusions

We have developed a differentiable NURBS module that can be directly integrated with existing machine learning frameworks. We have developed a mathematical framework that enables both forward evaluation and backpropagation of the losses while training. Our module is GPU-accelerated to allow fast evaluation of NURBS surface points and fast backpropagation of the derivatives. We have demonstrated the utility of our NURBS module for several CAD applications and deep learning applications. *NURBS-Diff* performs at the same level as existing standalone spline solutions used previously in the literature. Future work on the NURBS module includes developing support for trimmed NURBS surfaces and integrating complex curve constraints along trim edges for the watertight representation of CAD models. We have released the code for our NURBS module along with this paper. We believe this NURBS module will be the first step to better integrate deep learning with CAD and would lead to more diverse machine learning CAD applications.

Declaration of competing interest

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

Acknowledgments

This work was partly supported by the NSF, United States under grant CMMI-1644441 and the ARPA-E, United States under DIFFERENTIATE:DE-AR0001215. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by NSF, United States grant ACI-1548562 and the Bridges system supported by NSF, United States grant ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

Appendix A. Parametric derivatives

Here we provide the parametric surface derivatives with respect to the parameters u and v for completeness. Please refer to Piegl and Tiller [1] for details.

$$\mathbf{S}_{,u}(u,v) = \frac{\mathbf{NR}_{,u}(u,v)w(u,v) - \mathbf{NR}(u,v)w_{,u}(u,v)}{w(u,v)^2} \tag{A.1}$$

where,

$$\mathbf{NR}_{,u}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^{p}(u)N_{j}^{q}(v)w_{ij}\mathbf{P}_{ij}$$

$$w_{,u}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,u}^{p}(u)N_{j}^{q}(v)w_{ij}$$

Appendix B. Test surface details

Table B.12 shows geometrical details for each surface used for the fitting and surface offset tests. The surface fitting examples (Analytical and Ducky) are evaluated at 128^2 and 512^2 points, respectively. The surface offset test examples are evaluated at 20^2 points each.

Table B.12
Test NURBS surface parameters.

Surface model	p	q	n	m
Analytical	3	3	12	12
Ducky	3	3	14	13
Double curve	3	3	6	6
Multi patch - C^0	3	3	4	4
Multi patch - C^1	3	3	6	6
Aerofoil	3	3	50	24

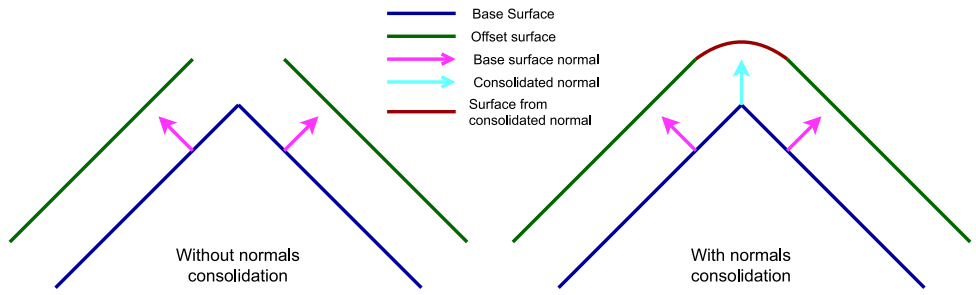


Fig. C.12. Without consolidated normals may lead to discontinuous surfaces, especially for C^0 connectivity (shown on the left). Normals consolidation ensures connectivity for the offset surfaces (shown on the right).

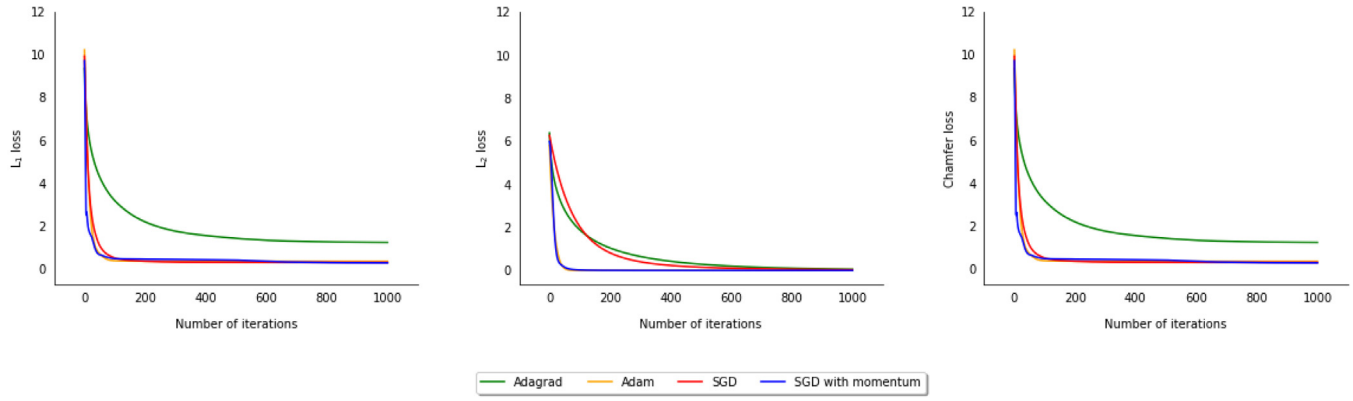


Fig. D.13. Results of the curve-fitting on a 3D helix point data with different optimizers.

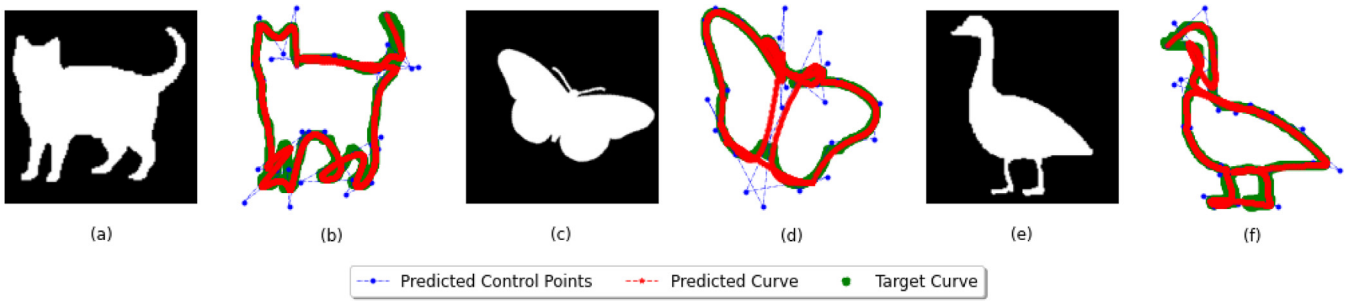


Fig. D.14. Curve fitting test performed on various skeleton geometry which results in an invalid curve generation due to self-intersecting geometry.

Appendix C. Computing normals on the common edge

For C^0 continuous surfaces, the surface normals at the common edge of two surfaces point in different directions. Hence, if the individual surface normals are used for the offset operation, it might lead to a gap or self-intersection between the offset surfaces. To deal with this case, we identify the common control points at the surface edges, and we recompute the resulting normal as the average of both the individual surface normals as shown in Fig. C.12. We then use this average normal to move the points on the common edges to generate the offset points.

Appendix D. Additional curve fitting results

We tested the curve fitting example of the 3D helix shown in Fig. 3 with different optimizers. Fig. D.13 shows the results for four different optimizers used in this test. We can see that all the optimizers converge to the same value for the $\mathcal{L}_2$ loss, but the Adagrad optimizer converges to a different value for the $\mathcal{L}_1$ and $\mathcal{L}_{CD}$ loss. In addition, SGD with momentum has the fastest convergence rates for all losses.

For some complex shapes, our method cannot generate a curve without self-intersections and loops for the Pixel dataset. This is because the weightage of the curve length regularization parameter needs to be tuned for each object based on its complexity. Fig. D.14 shows the results from 3 curve fitting tests where the curves generated are self-intersecting. Adding additional constraints to prevent this is a possible future research direction.

Appendix E. *NURBS-Diff* implementation details

Our *NURBS-Diff* module was implemented using Pytorch library, which allows us to implement custom deep learning layers that we can use alongside traditional deep learning layers such as convolution, max-pooling, dense layers, etc. `torch.nn.module` gives us an interface to create our custom layers; however, the functions have to be defined in an automatically differentiable manner. In our application, we use `torch.autograd.Function` to define our custom forward and backward pass computations for the NURBS basis functions evaluation and curve/surface evaluation. This function is now used in the `torch.nn.module` to

create the layer. The actual code for the forward and backward pass (for evaluation) is written in C++ Language with CUDA integration for GPU acceleration. These modules (written in C++) are compiled along with PyBind11 package to use in Python (supported by Pytorch). The compilation is performed using a simple setup from `torch.utils.cpp_extension`. The main source code of *NURBS-Diff* module will be made public.

We create two different layers: (i) B-Spline evaluation (ii) NURBS evaluation. Although the functions used in both are the same, in B-Spline evaluation, we avoid computing the basis functions every forward pass. Instead, we precompute the basis functions initially and only perform the forward evaluation of the curve/surface. This saves us computational time during the forward and backward pass.

References

[1] Piegl L, Tiller W. The NURBS book. Berlin, Heidelberg: Springer-Verlag; 1997.
[2] Mescheder L, Oechsle M, Niemeyer M, Nowozin S, Geiger A. Occupancy networks: Learning 3D reconstruction in function space. In: Conference on computer vision and pattern recognition. 2019, p. 4460–70.
[3] Park JJ, Florence P, Straub J, Newcombe R, Lovegrove S. DeepSDF: Learning continuous signed distance functions for shape representation. In: Computer vision and pattern recognition. 2019, p. 165–74.
[4] Niemeyer M, Mescheder L, Oechsle M, Geiger A. Differentiable volumetric rendering: Learning implicit 3D representations without 3D supervision. In: Conference on computer vision and pattern recognition. 2020, p. 3504–15.
[5] Deng B, Genova K, Yazdani S, Bouaziz S, Hinton G, Tagliasacchi A. CvxNet: Learnable convex decomposition. In: Conference on computer vision and pattern recognition. 2020, p. 31–44.
[6] Chen Z, Tagliasacchi A, Zhang H. BSP-Net: Generating compact meshes via binary space partitioning. In: Conference on computer vision and pattern recognition. 2020, p. 45–54.
[7] Peng S, Niemeyer M, Mescheder L, Pollefeys M, Geiger A. Convolutional occupancy networks. 2020, ArXiv 2.
[8] Davies T, Nowrouzezahrai D, Jacobson A. Overfit neural networks as a compact shape representation. 2020, arXiv.
[9] Atzmon M, Lipman Y. SAL: Sign agnostic learning of shapes from raw data. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2020; p. 2565–74.
[10] Groueix T, Fisher M, Kim VG, Russell BC, Aubry M. AtlasNet: A Papier-Mâché approach to learning 3D surface generation. 2018, arXiv.
[11] Barill G, Dickson NG, Schmidt R, Levin DIW, Jacobson A. Fast winding numbers for soups and clouds. ACM Trans Graph 2018;37(4):1–12.
[12] Innes MJ. Algorithmic differentiation. In: Machine learning and systems. 2020, p. 1–12.
[13] Innes M, Edelman A, Fischer K, Rackauckas C, Saba E, Shah VB, Tebbutt W. A differentiable programming system to bridge machine learning and scientific computing. 2019, arXiv abs/1907.07587.
[14] Schafer F, Kloc M, Bruder C, Lorch N. A differentiable programming method for quantum control. 2020, ArXiv.
[15] Li T-M, Gharbi M, Adams A, Durand F, Ragan-Kelley J. Differentiable programming for image processing and deep learning in Halide. Trans Graph 2018;37(4):1–13.
[16] Degrave J, Hermans M, Dambre J, Wyffels F. A differentiable physics engine for deep learning in robotics. Front Neurorobot 2017;13.
[17] Alnæs M, Blechta J, Hake J, Johansson A, Kehlet B, Logg A, Richardson C, Ring J, Rognes ME, Wells GN. The FEniCS project version 1.5. Arch Numer Softw 2015;3(100).
[18] Li T-M, Aittala M, Durand F, Lehtinen J. Differentiable Monte-Carlo ray tracing through edge sampling. Trans Graph 2018;37(6):1–11.
[19] Chen W, Ling H, Gao J, Smith E, Lehtinen J, Jacobson A, Fidler S. Learning to predict 3D objects with an interpolation-based differentiable renderer. In: Advances in neural information processing systems, Vol. 32. 2019, p. 1–11.
[20] Cuturi M, Teboul O, Vert J-P. Differentiable ranks and sorting using optimal transport. 2019, p. 1–10, arXiv.
[21] Cho M, Joshi A, Lee XY, Balu A, Krishnamurthy A, Ganapathysubramanian B, Sarkar S, Hegde C. Differentiable programming for piecewise polynomial functions. In: Association for the advancement of artificial intelligence conference. 2021, p. 1–10.
[22] Blondel M, Teboul O, Berthet Q, Djolonga J. Fast differentiable sorting and ranking. 2020, ArXiv abs/2002.08871.
[23] Baydin AG, Pearlmutter BA, Radul AA, Siskind JM. Automatic differentiation in machine learning: A survey. 2018, arXiv.
[24] Müller J-D, Zhang X, Akbarzadeh S, Wang Y. Geometric continuity constraints of automatically derived parametrisations in CAD-based shape optimisation. Int J Comput Fluid Dyn 2019;33(6–7):272–88.
[25] Blondel M, Teboul O, Berthet Q, Djolonga J. Fast differentiable sorting and ranking. 2020, arXiv.
[26] Vlastelica M, Paulus A, Musil V, Martius G, Rolínek M. Differentiation of blackbox combinatorial solvers. 2020, arXiv.
[27] Sheriffdeen S, Ragusa JC, Morel JE, Adams ML, Bui-Thanh T. Accelerating PDE-constrained inverse solutions with deep learning and reduced order models. 2019, arXiv.
[28] Joshi A, Cho M, Shah V, Pokuri B, Sarkar S, Ganapathysubramanian B, Hegde C. InvNet: Encoding geometric and statistical invariances in deep generative models. In: Association for the advancement of artificial intelligence conference. 2020, p. 1–8.
[29] Djolonga J, Krause A. Differentiable learning of submodular models. In: Neural information processing systems. 2017, p. 1014–24.
[30] Zhang X. CAD-based geometry parametrization for shape optimization using Non-uniform Rational B-Splines (Ph.D. thesis), Queen Mary University of London; 2018.
[31] Ugolotti M, Vaughan B, Orkwis PD. Differentiated ML-based modeling of structured grids for gradient-based optimization. In: AIAA scitech 2021 forum. 2021, p. 0895.
[32] Mykhaskiv O, Banović M, Auriemma S, Mohanamuraly P, Walther A, Legrand H, Müller J-D. NURBS-based and parametric-based shape optimization with differentiated CAD kernel. Comput-Aided Des Appl 2018;15(6):916–26.
[33] Müller J-D, Zhang X, Akbarzadeh S, Wang Y. Geometric continuity constraints of automatically derived parametrisations in CAD-based shape optimisation. Int J Comput Fluid Dyn 2019;33(6–7):272–88.
[34] Hoschek J. Intrinsic parametrization for approximation. Comput Aided Geom Design 1988;5(1):27–31.
[35] Piegl LA, Tiller W. Computing the derivative of NURBS with respect to a knot. Comput Aided Geom Design 1998;15(9):925–34.
[36] Minto L, Zanuttigh P, Pagnutti G. Deep learning for 3D shape classification based on volumetric density and surface approximation clues. In: International joint conference on computer vision, imaging and computer graphics theory and application. 2018, p. 317–24.
[37] Erwinski K, Paprocki M, Wawrzak A, Grzesiak LM. Neural network contour error predictor in CNC control systems. In: 2016 21st international conference on methods and models in automation and robotics (MMAR). IEEE; 2016, p. 537–42.
[38] Fey M, Lenssen JE, Weichert F, Müller H. SplineCNN: Fast geometric deep learning with continuous B-spline kernels. In: Conference on computer vision and pattern recognition. 2018, p. 869–77.
[39] Balestriero R, et al. A spline theory of deep learning. In: International conference on machine learning. PMLR; 2018, p. 374–83.
[40] Balu A, Nallagonda S, Xu F, Krishnamurthy A, Hsu M-C, Sarkar S. A deep learning framework for design and analysis of surgical bioprosthetic heart valves. Sci Rep 2019;9(1):1–12.
[41] Sharma G, Liu D, Maji S, Kalogerakis E, Chaudhuri S, Měch R. ParSeNet: A parametric surface fitting network for 3D point clouds. 2020, arXiv.
[42] Krishnamurthy A, Khardekar R, McMains S, Haller K, Elber G. Performing efficient NURBS modeling operations on the GPU. IEEE Trans Vis Comput Graphics 2009;15(4):530–43.
[43] Rossum G. Python reference manual. Tech. rep., Amsterdam: CWI (Centre for Mathematics and Computer Science); 1995.
[44] Jakob W, Rhinelander J, Moldovan D. Pybind11 - Seamless operability between C++ 11 and Python. 2017.
[45] NVIDIA. NVIDIA CUDA C programming guide, Vol. 120. NVIDIA Corporation; 2011, p. 8, (18).
[46] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al. PyTorch: An imperative style, high-performance deep learning library. 2019, arXiv.
[47] Kingma DP, Ba J. Adam: A method for stochastic optimization. 2014, arXiv.
[48] Lydia A, Francis S. Adagrad-An optimizer for stochastic gradient descent. Int J Comput Inf Sci Eng 2019;6(5).
[49] Demir I, Hahn C, Leonard K, Morin G, Rahbani D, Panotopoulou A, Fondevilla A, Balashova E, Durix B, Kortylewski A. SkelNetOn 2019: Dataset and challenge on deep learning for geometric shape understanding. 2019, arXiv.
[50] Qi CR, Su H, Mo K, Guibas LJ. Pointnet: Deep learning on point sets for 3D classification and segmentation. 2017, arXiv.