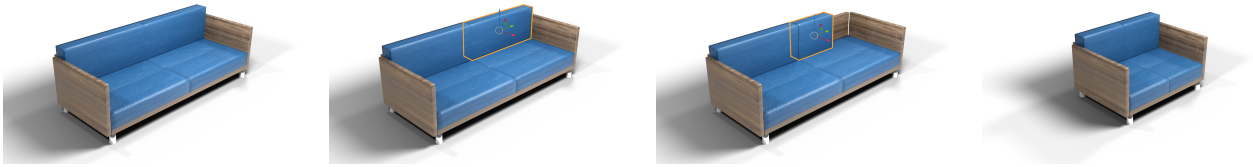
DOI: 10.1111/cgf.14475
EUROGRAPHICS 2022 / R. Chaine and M. H. Kim
(Guest Editors)
Volume 41 (2022), Number 2

Automatic Differentiable Procedural Modeling

Mathieu Gaillard[1], Vojtěch Krs[2], Giorgio Gori[2], Radomír Měch[2], Bedrich Benes[1]

[1] Computer Science, Purdue University, West Lafayette, IN, USA
[2] Adobe Research, San Jose, CA, USA



a) b) c) d)

Figure 1: ***Pipeline of ADPM:*** *The initial procedural model designed by a technical artist (a). The end user edits the model by shrinking the top right cushion (b-c). ADPM solves the inverse problem and propagates the modifications of the cushion back to the input parameters of the procedural model (d).*

Abstract

Procedural modeling allows for an automatic generation of large amounts of similar assets, but there is limited control over the generated output. We address this problem by introducing Automatic Differentiable Procedural Modeling (ADPM). The forward procedural model generates a final editable model. The user modifies the output interactively, and the modifications are transferred back to the procedural model as its parameters by solving an inverse procedural modeling problem. We present an auto-differentiable representation of the procedural model that significantly accelerates optimization. In ADPM the procedural model is always available, all changes are non-destructive, and the user can interactively model the 3D object while keeping the procedural representation. ADPM provides the user with precise control over the resulting model comparable to non-procedural interactive modeling. ADPM is node-based, and it generates hierarchical 3D scene geometry converted to a differentiable computational graph. Our formulation focuses on the differentiability of high-level primitives and bounding volumes of components of the procedural model rather than the detailed mesh geometry. Although this high-level formulation limits the expressiveness of user edits, it allows for efficient derivative computation and enables interactivity. We designed a new optimizer to solve for inverse procedural modeling. It can detect that an edit is under-determined and has degrees of freedom. Leveraging cheap derivative evaluation, it can explore the region of optimality of edits and suggest various configurations, all of which achieve the requested edit differently. We show our system's efficiency on several examples, and we validate it by a user study.

CCS Concepts

• ***Computing methodologies*** → ***Shape modeling;*** *Interactive simulation;*

1. Introduction

Procedural modeling automatically generate virtual assets, such as 3D models, textures, or effects, with an algorithm [EMP*03]. It allows for an automatic generation of large amounts of similar assets by varying the procedural model's parameters, rules, and initialization. Procedural models are often complex non-linear systems with an intricate cascade of feedback loops that can quickly amplify small changes in parameters over a few iterations.

One major problem with procedural modeling is controllability. In fact, procedural generation is usually a one-way process and the final model, e.g., a 3D scene composed of a hierarchy of meshes, loses all parametric information. Consequently, any edit on a procedural model must occur in the parametric space as opposed to the 3D space. Yet, users are most comfortable expressing their intent in 3D space. This discrepancy makes the controlled generation of variations difficult.

We address the controllability problem of Procedural Modeling by introducing Automatic Differentiable Procedural Modeling (ADPM). ADPM is a node-based procedural modeling engine that generates hierarchical 3D scene geometry and has an intuitive

GUI. The key feature of ADPM is that it can translate the generated model's modifications in 3D space back to the parametric space, and thus couples interactive editing with procedural modeling. There is no need for the technical artist to spend extra time to rig the procedural model like it is traditionally done in animation. The typical workflow that we target is when a technical artist first designs the procedural model. Later the end-user can customize the resulting procedural model without knowing its underlying representation. With ADPM, the procedural model is always available, all modifications are non-destructive, and the user can use a procedural model comparable to non-procedural interactive modeling.

For the user, editing the model directly in 3D space is the most convenient, but it comes with a disadvantage; edits can be ambiguous as multiple changes in parameter space may match the edit in 3D space. Because it is unreasonable to ask the user to specify all possible degrees of freedom when editing the model. Our optimizer detects when an edit is under-determined, explores the region of optimality, and gathers several solutions. These optimal solutions are ranked, shown to the user, and several solutions are highlighted, e.g., the closest solution to the initial configuration, or the solution that best keeps the model's proportions. A user study justifies this strategy.

Our work is inspired by recent work on the differentiability of rendering [KBM*20], physics systems [dABPSA*18, HKUT20], and inverse control of parametric shapes [MB21, CSQ*22]. We build a proxy differentiable representation of the procedural model that allows for efficient computation of the derivatives of the output with respect to the model's parameters. This concept can be implemented on top of any node-based procedural engine, as long as nodes' operations are differentiable and all input parameters are continuous. To make the problem tractable and implementation practical, we focus on the differentiability of higher-level primitives and their bounding volumes instead of the detailed mesh geometry. This allows us to efficiently compute derivatives of affine transformations of all of the primitives that compose the generated model. We show our system on various examples, such as models of furniture, insects, or robots. Figure 1 shows an example editing workflow enabled by our system. Note that these edits are interactive and affect the underlying procedural representation of the model.

We claim the following contributions: (1) WYSIWYG (What You See Is What You Get) interactive editing of models aimed at non-expert users in the context of a node-based procedural engine. (2) A new optimizer for inverse procedural modeling that detects the under-determined nature of a problem and finds various optimal configurations within a budget of time.

2. Related Work

Procedural Modeling and Representations: Early PM methods include L-systems [Lin68] that are a parallel string rewriting system used to simulate vegetation [AK84]. Fractals [Man82] were used to generate terrains [FFC82], and we refer the reader to a recent review of the state of the art by Galin et al. [GGP*19]. Shape grammars [SG71] were used for generating man-made structures, such as facades [WWSR03] and buildings [MWH*06]. Various works attempt to improve the control over the procedural generation such as [PJM94, BŠMM11, LBZ*10]. Most of these algorithms attempt to add some guidance based on the environment that guides the forward modeling. Finally, some of the grammars were extended and specialized in handling more complex models, such as the *CGA++* for architecture [SM15], the method of Krecklau and Kobbelt [KK11] for interconnected structures or the method from Santoni and Pellacini [SP16] for 2D patterns. We refer the reader to a survey by Smelik et al. [STBB14] for a further overview of procedural modeling. The direct procedural models do not provide editing capabilities beyond running the procedural generation from scratch.

The grammar representation, be it for bottom-up growth like L-systems, or top-down like split grammars, is powerful but difficult to understand for novice users. While there is research on grammar manipulation tools, such as a visual grammar editor [LBZ*10], a common representation and user interface of procedural systems is the node graph. Node graphs model the data flow [JHM04], usually of 3D geometry, and operations, such as repetition or transformation, are performed on the data. Early use in procedural modeling includes methods for modeling architecture and urban environments [Pat10, SMBC13, SEBC15]. Currently, node graphs are used in many popular 3D modeling and texturing packages, such as Houdini [Sid21], Substance [Ado21], and Grasshopper [MA21]. While the visual programming aspect of node graphs reduces entry barriers, it does not fully address or solve procedural modeling complexity and control issues. Our system extends the node graph systems, but it does not require the end-user to know about it, as all the model editing happens in the viewport.

Inverse Procedural Modeling (IPM) aims to translate existing geometry to a procedural representation. This is useful for generating variations of a given non-procedural model but also as a way to control the procedural generation.

Šťava et al. [ŠBM*10] inferred L-system grammar parameters from vector data and Bokeloh et al. [BWS10] automatically finds the grammar building blocks by analyzing partial symmetries. Talton et al. [TLL*11, TYK*12] used Monte Carlo Markov Chain (MCMC) method to find grammar expansion to minimize multiple objectives and later used it for finding a more general grammar. Šťava et al. [SPK*14] used MCMC to estimate and optimize parameters of a tree procedural model. Ritchie et al. [RMGH15] improved on Talton's approach by using Sequential Monte Carlo (SMC). Statistical analysis of example scenes to synthesize virtual worlds via a brush interface interactively was used by Emilien et al. [EVC*15], and IPM was applied to urban modeling by Vanegas et al. [VAB10, VGDA*12]. Genetic algorithms were applied to control procedural generation by Haubenwallner et al. [HSS17] and Krs et al. [KMG*20] used a genetic algorithm to evolve a procedural node graph.

Recently, neural-based approaches have been used in IPM. Ritchie et al. [RTHG16] trained a neural network to act as an importance sampler for an SMC inference algorithm, amortizing the optimization cost. Kalojanov et al. [KLMK19] represent shapes as strings, leveraging machine learning language processing techniques to manipulate the shapes indirectly. Methods have been developed that infer grammars or short shape-generating programs

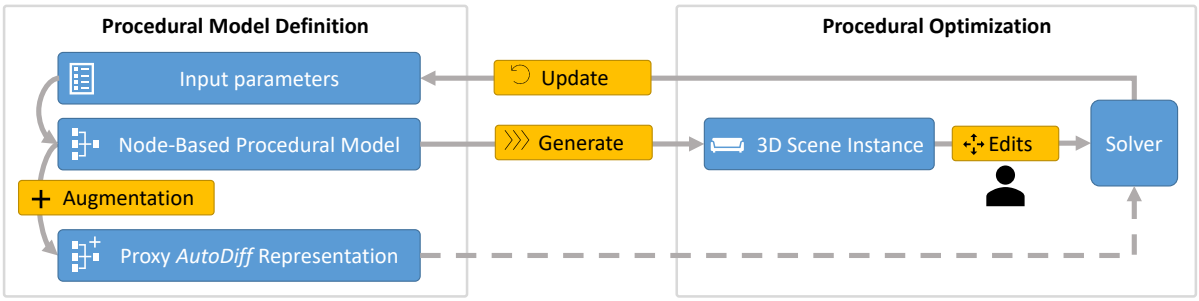


Figure 2: ***ADPM overview:*** *The input is a procedural graph that defines a parameterized Node-Based Procedural Model. When executed, it generates a 3D scene and a proxy autodiff representation of the scene. The user modifies the 3D scene directly in the viewport and changes are picked up by the solver, which optimizes the procedural model parameters to match edits. The optimization is accelerated by the proxy autodiff representation. Finally, the Node-Based Procedural Model's input parameters are updated by the solver to reflect the user's edits.*

from images. The method of Sharma et al. [SGL*18] learns to predict Constructive Solid Geometry (CSG) programs from an image using reinforcement learning with image difference loss. Du et al. [DIP*18] presented a method to convert a 3D model to a CSG tree. Ellis et al. [ERSLT18] showed program inference from hand-drawn images of 2D shapes, Tian et al. [TLS*19] from 3D shapes and Liu and Wu [LW19] from images of simple 3D scenes. L-system grammar from a 2D image of branching structures has been inferred by Guo et al. [GJB*20], and both Hu et al. [HDR19] and Guo et al. [GHYZ20] use learning methods to infer parameters of procedural materials. Jones et al. [JBX*20] learn to generate programs written in a domain-specific language that generate 3D shapes. Liu et al. [LVW*15] generate variations of an existing model by splitting it, and reassembling its parts into a plausible model. Mathur et al. [MPZ20] provide a way to translate edits made in the viewport back to the CAD parametric program that generated the displayed 3D model. More generally, Mayer et al. [MKC18] presented a bidirectional programming language that can update the program to account for edits made on its output.

The inference of the procedural parameters can aid the creation and editing of the procedural model. An early work of Ijiri et al. [IOI06] used sketching to control the growth of L-systems. Longay et al. [LRBP12] later used space colonization [PHL*09] to grow a procedural model based on user input. Smelik et al. [STdKB11] applied sketch to the generation of virtual worlds, including terrain and roads. More recently, Huang et al. [HKYM16] used deep learning to predict procedural models from sketches and Nishida et al. [NGDA*16] presented a similar system for sketching buildings. While these methods allow for some control of the procedural generation, the edits are still indirect. In contrast, Lipp et al. [LSL*19] showed a method that can automatically generate handles for "good edit locations", enabling direct local editing of architectural, procedural models. Our method is similar in that it allows direct local edits. However, we focus on general 3D geometry generated by node graphs.

Auto differentiation: (autodiff) efficiently computes derivatives of a function defined by a program [Spe80, G*89]. Autodiff gained popularity in various applications. In particular, it has been used to accelerate backpropagation in deep learning frameworks such as PyTorch [PGC*17] and Tensorflow [AAB*16]. Autodiff has also been recently applied to rendering for computing per-pixel gradients with respect to input parameters like shapes, materials, and lighting. Loper and Black [LB14] and Kato et al. [KUH18] presented differentiable renderers with simple materials and shading. Li et al. [LADL18] implemented a Monte Carlo differentiable path tracer and Laine et al. [LHK*20] presented high performance rasterization based differentiable renderer. For the latest overview, see the differentiable rendering survey by Kato et al. [KBM*20].

Currently, there is a growing interest in auto differentiating physics simulation. Both de Avila Belbute-Peres et al. [dABPSA*18], and Degrave et al. [DHD*19] built differentiable physics engines on top of existing deep learning frameworks, Liang et al. [LLK19] presented a differentiable cloth simulation, and Schenck and Fox [SF18] showed a differentiable fluid dynamics simulation. A specialized differentiable language for physical simulations has been developed by Hu et al. [HAL*19]. Other recent autodiff tools include Kornia [RMP*20] for computer vision and for graph and geometry procesing [FL19] .

Gleicher [Gle94] introduced an early method for controlling 3D objects interactively. His system expresses the manipulations as differential equations, solved using a technique similar to auto differentiation. A work close to ours has been recently presented by Michel and Boubekeur [MB21]. Their method allows for interactive direct manipulation of a parametric shape defined by a procedural graph. They use a brush metaphor for editing and propose a new way of handling under-determined edits based on the brush radius, whereas our work uses 3D gizmos for interaction and proposes multiple configurations if the edit is under-determined. Another difference is the level of edits; their edits can be more local: at the vertex-level, whereas our system can only backpropagate edits at the object-level, which means that the shape of individual components of the model cannot be interactively edited. This is the tradeoff that comes with our formulation that focuses on differentiability of higher-level primitives. However, one major limitation of their approach is that they use finite-differences, which is slower and can cause numerical problems, whereas we use auto-differentiation. Cascaval et al. [CSQ*22] present another method close to ours. Their differentiable procedural model is ex-

pressed as a CAD program, which is conceptually equivalent to our graph-based representation. However, they offer editing at a vertex granularity, which has better expressiveness, at the expense of optimization speed. To deal with ambiguous edits, they implement a few local-optimization heuristics that minimize the same objective function, but with different regularization terms. Shi et al. [SLH*20] presented a differentiable version of a procedural material graph, which allows them to optimize and match a photo of existing material. Our method extends this concept to a general procedural node graph generating 3D geometry and combines it with interactive manipulation.

3. Overview

Figure 2 shows an overview of ADPM. At its core is a parameterized Node-Based Procedural Model defined by a procedural graph and its associated parameters. Nodes in the procedural graph take data as input, execute operations, and output new data. Data traversing edges of the procedural graph can be 3D objects or variables of any type. A special input node allows parametrizing the Node-Based Procedural Model (see Section 4). The nodes in the procedural graph are connected and generate a 3D scene when executed.

The key contribution of our work is the procedural optimization. We augment the 3D scene tree with automatically differentiable parts, which leads to an Automatically Differentiable Node-Based Procedural Model (ADPM). The ADPM executes its nodes and generates a 3D scene. The generated geometry is an instance of a 3D procedural model that the user can easily edit (translate, rotate, or scale parts, change some elements, etc.). These changes are processed by a solver that optimizes the procedural model and modifies the input parameters of the ADPM to match the edits.

4. Procedural system

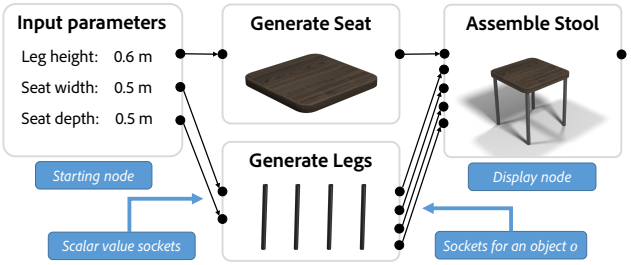


Figure 3: ***Node-based Procedural Model:*** *Nodes represent high-level operations. The "input parameters" node defines the size of parts of the output model, the "generate seat" and the "generate legs" nodes create the geometry of different parts of the model. The "assemble stool" node connects parts to form the stool. Note that there may be multiple input nodes, but only one active display node. See Figure 17 for the actual graph implemented in our system.*

ADPM is a directed acyclic multi-graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ composed of nodes $v_i \in \mathcal{V}$ and edges $e_i \in \mathcal{E}$. Each node v_i has an input set of parameters $x \in \mathcal{R}^n$ and data D, that is most often a 3D geometric object. The node transforms D (e.g., applies a geometric transformation on a 3D object) and outputs the transformed version $v_i(x, D)$. We call this operation a node execution or production. The entire ADPM is executed by executing all its nodes $\mathcal{V}$ in topological order. Note that since the graph in ADPM is acyclic, each node is executed only once per execution of the ADPM.

The procedural graph $\mathcal{G}$ can be thought of as a function that takes an n-dimensional vector $x \in \mathcal{R}^n$ as input and outputs a 3D scene $\mathcal{S}$:

$$\mathcal{S} = \mathcal{G}(x) \quad \text{with} \quad x \in \mathcal{R}^n. \tag{1}$$

An example in Figure 3 generates a stool of varying size. As with any procedural model, the user can modify the generated model by changing the input parameters, and the 3D geometry is consequently updated. ADPM also allows the inverse operation, that is, to automatically propagate any modification of the generated model back to the input parameters.

Inputs and outputs of nodes $\mathcal{V}$ are symbolized by sockets connected via graph edges. Nodes can have any number of input and output sockets that send and receive different types of data. Input sockets can be connected to only one edge, whereas output sockets can be connected to many edges. An edge can connect only sockets of the same type. Data flowing through edges $\mathcal{E}$ can be of any type, including scalars, vectors, and 3D objects.

There are two special types of nodes (see Figure 3): the end node, which is also called the display node, and the starting nodes. Starting nodes do not have input sockets and are not dependent on any other nodes; they are typically input variables nodes. The end node does not have any dependent nodes, and generates the output 3D scene geometry, which can be displayed. The ADPM production ends when the display node is executed.

5. Differentiable representation

The current node-based procedural models (e.g., [Sid21, Ado21, MA21]) allow only for forward procedural generation, i.e., knowing $\mathcal{G}$ and the input x, it generates the 3D model $\mathcal{S} = \mathcal{G}(x)$, because the inverse operation $x = \mathcal{G}^{-1}(\mathcal{S})$ is not generally known. We model the inverse operation as a data fitting problem. We cast it as an optimization problem and solve it by using gradient-based optimization. The setup of a differentiable computational graph model is motivated by the ability of auto differentiation to accelerate optimization.

5.1. Automatic Differentiation (autodiff)

Autodiff is an efficient and numerically stable method for computing the gradient of a function [G*89]. It creates an expression graph of computation for the evaluated function, and it keeps track of arithmetic operations (addition, multiplication, etc.) while the function's output value is computed. The algorithm applies the chain rule recursively on the expression graph to obtain partial derivative values. There are two main ways to apply the chain rule. In forward mode, derivatives are propagated in the expression graph starting from the input variables towards the function result. In reverse mode, derivatives are backpropagated in the expression graph, starting from the result towards the input variables.

We use reverse autodiff mode [Lin76] that has a major advantages for our use case. It is efficient for evaluating partial derivatives of functions $f : \mathcal{R}^n \mapsto \mathcal{R}$ with $n \gg 1$, which are common during gradient-based optimization of a single-valued objective function. An alternative to reverse autodiff would be to directly compute analytical differences using the procedural graph $\mathcal{G}$, although it is conceptually equivalent, it would have to be implemented from scratch. Using an existing auto-differentiation library makes our approach more accessible and general.

Converting a procedural graph to a differentiable computational graph allows modifications made in the final model $\mathcal{S}$ to be efficiently backpropagated to the input parameters x (Section 6). A straightforward way to make the procedural graph differentiable would be to convert all generated mesh vertices into an auto-differentiable type. However, this would make the problem slow for interactive editing. The gradient estimation complexity would depend on the number of vertices in the scene, which is often high. Instead, we work with the hierarchy of bounding boxes (proxies) of objects in the scene $\mathcal{S}$. Our framework tracks affine transformations of the objects generated by the procedural graph $\mathcal{G}$ making the objective function simpler to optimize because it uses fewer terms.

5.2. Automatic Differentiable Node-Based Procedural Model

In parallel to the 3D scene, ADPM maintains a differentiable representation of the scene tree. To each object in the scene tree is associated 1) an auto-differentiable local transformation matrix that is expressed in the object's parent frame of reference, and translated, rotated, or scaled as the objects undergo transformations, and 2) an auto-differentiable oriented bounding box (OBB). The OBBs are generated as axis-aligned, but they get modified when objects go through the apply transform node. Matrices and OBBs in the differentiable representation are organized hierarchically to mirror the hierarchy of the scene tree.

When the procedural graph generates an object o_i, ADPM assigns it a default auto-differentiable local transformation identity matrix $M_i = I_4$ and a default axis-aligned OBB B_i. The OBB B_i is represented by its center, three vectors forming an orthogonal frame, and its size along the three directions. The components of B_i are expressed relative to the reference frame defined by the local transformation matrix M_i, expressed in the local frame of reference defined by the object's o_i parent in the scene tree. To get the position of the oriented bounding box B_i in the world coordinates, one needs to compose the hierarchy of local transformation matrices $M_{root} \times \cdots \times M_{parent} \times M_i$ from the root of the scene tree to the object o_i and apply the final transform matrix on the OBB B_i. Once the OBB B_i is expressed in the world coordinates, we get the coordinates of its eight corners.

Our framework implements nodes that correspond to the smallest set of operations needed to create our example procedural models, such as the one shown in Figure 1. The set of nodes is deliberately minimalist yet powerful. In fact, applying the basic nodes successively is equivalent to applying a high-level operation. In that spirit, to ease the creation process, we implemented a sub-graph mechanism that allows the technical artist to combine several basic nodes into a higher-level node that can be reused later. Any additional node could be implemented in a production-ready system as long as its operation is differentiable. Section 10.1 of the appendix is a list of auto-differentiable nodes implemented in ADPM.

6. Inverse Problem and Optimization

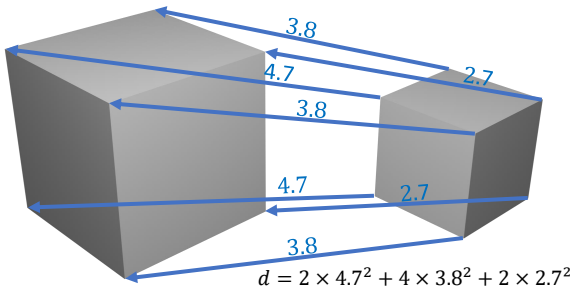


Figure 4: ***Objective function example:*** *The objective function to minimize is the sum of the squared distances between the eight corners of (left) the initial oriented bounding box and (right) the target oriented bounding box.*

6.1. Optimization Problem

The *forward problem* is denoted by $x \mapsto S = \mathcal{G}(x)$, indicating that the 3D scene S is generated by executing the procedural graph $\mathcal{G}$ with the input parameters x. Let $\mathcal{I} \subset \mathcal{R}^n$ be the set of possible values of the input parameters $x \in \mathcal{I}$ and $\mathcal{S}$ the set of possible outcomes of the procedural model $\mathcal{S} = \{\mathcal{G}(x) | \forall x \in \mathcal{I}\}$. Mathematically $\mathcal{G}$ is called the forward operator [YRKC17].

The *inverse problem* consists in finding the parameter $x \in \mathcal{I}$ that gives a certain $S \in \mathcal{S}$ without knowing the inverse application $x = \mathcal{G}^{-1}(S)$. The *inverse problem* can be ill-posed because the forward operator $\mathcal{G}$ can be non-injective (i.e., two or more different input parameters generate the same scene). It is also possible that one tries to find input parameters for a target 3D scene $\hat{S}$ that is not part of $\mathcal{S}$. In this case a desirable outcome of the inverse problem would be an input parameter $x \in \mathcal{I}$ that gives an orthogonal projection of $\hat{S}$ on $\mathcal{S}$. We express the inverse problem as an optimization problem that aims to find the input parameters $x \in \mathcal{I}$ that best fit the expected outcome $\hat{S}$. We introduce an objective function (i.e., a cost function) $\Psi_{\mathcal{G}} : (\mathcal{I} \times \mathcal{S}) \mapsto \mathcal{R}^+$ (see Section 6.2) that measures how well input parameters x fit a 3D scene $\hat{S}$. We attempt to minimize $\Psi_{\mathcal{G}}$

$$\min_{x \in \mathcal{I}} \Psi_{\mathcal{G}}(x, \hat{S}). \tag{2}$$

To build the target configuration $\hat{S}$ that is the input to the inverse problem, ADPM watches the interactive 3D modifications made by the user. When the procedural graph $\mathcal{G}$ is executed with initial parameters x_0, the initial configurations $\mathcal{G}(x_0)$ of all OBBs in the scene are retained as a reference. While the user is editing the 3D scene, ADPM tracks changes made to the OBBs. With each change, the positions of OBBs in the viewport are compared to the reference to the initial configuration. Any OBB that the user modified is added to the target configuration $\hat{S}$ so that the optimization target is a map of new positions of OBBs with their identifiers. Note that if an OBB was not edited, it is not part of the target $\hat{S}$ and can freely

move during optimization. When solving the inverse problem, the optimizer only tries to match the edits made by the user. Therefore, if something has to stay in place, the user must manually annotate it by adding the fixed OBB to $\hat{S}$.

6.2. Objective Function $\Psi_{\mathcal{G}}$

The objective function $\Psi_{\mathcal{G}}$ is defined so that:

$$\Psi_{\mathcal{G}}(x,\hat{S}) = 0 \iff \mathcal{G}(x) = \hat{S}. \tag{3}$$

That implies if one finds a global minimum of $\Psi_{\mathcal{G}}$, the inverse problem is solved, the intent of the user is fulfilled, and the model is visually similar to what the user had in mind.

The objective function $\Psi_{\mathcal{G}}$ is defined as the sum of squared distances between corners of target OBBs in $\hat{S}$ and the corresponding OBBs generated by $\mathcal{G}(x)$ (see Figure 4)

$$\Psi_G(x,\hat{S}) = \sum_{s_i \in \hat{S}} d(s_i, \mathcal{G}_i(x)), \tag{4}$$

where $d(s_i, s_j)$ is the squared distance between the OBBs s_i and s_j.

It has been shown in the context of 6D pose estimation [LWJ*18] that the optimization can be simplified by matching corners of OBBs instead of matching their centers, Euler angles, and extents. Directly matching Euler angles may cause ambiguities because of the gimbal lock.

Note that while editing, the user moves parts of the ADPM, which may end up disconnected (see Figure 8 left). However, this is fixed automatically by the solver as it maintains the procedural model's internal consistency (Figure 8 right). Hence, even if the target configuration is unreachable (i.e.,$\hat{S} \notin \mathcal{S}$) or if the solver fails to reach the global minimum, the output is still a plausible model that is also as similar as possible to the target.

6.3. Optimizer

The input to our optimizer consists of 1) the objective function $\Psi_{\mathcal{G}}$, 2) the initial parameters x_0, and 3) a maximum budget of evaluations of the objective function. Our novel optimizer for inverse procedural modeling works in two phases. First, it uses local optimization to reach the nearest optimal region. If the problem is under-determined, it uses the remaining budget to explore the optimality region. Otherwise, it runs the global optimization until the budget is used up.

We adapted the Basin-hopping method [WD97], which is a two-phase method that combines global stepping and local minimization. As such, the actual method used for local minimization is not critical, and different methods work depending on the properties of the problem. We detail the algorithms we use for local and global optimization below.

6.4. Local Optimization

We optimize a continuous and single-valued objective function that can be non-linear. Our differentiable representation allows for cheap evaluation of $\Psi_{\mathcal{G}}$ and its gradient. The Hessian matrix is available but can be expensive to compute relatively to the gradient (it has $\mathcal{O}(n^2)$ complexity). The set of parameters is small (10-100), and there are no constraints apart from the fact that parameters can be bounded.

We use quasi-Newton methods for the local optimization, which only evaluate the objective function and its gradient. Depending on whether the problem was unbounded or bounded, we used BFGS [NW06] or L-BFGS-B [BLNZ95] respectively. We could also use a trust-region method [CGT00], but depending on the implementation, it can be slower than quasi-Newton methods because the Hessian matrix can be prohibitively more expensive to compute than the gradient.

6.5. Check if Problem is Under-determined

To check if the problem is under-determined, we model the objective function using its second-order Taylor series expansion approximation and look for directions in which the function is constant. The local optimization phase finds an optimal point. Per definition of the second-order necessary condition for optimality, we know that the gradient is zero and the Hessian matrix is semi-definite positive at a locally optimal point. Since the gradient is zero, the second-order Taylor series expansion defines a quadratic form, whose principal axes are the eigenvectors of the Hessian matrix. The Hessian matrix is symmetric. Therefore, its eigenvalues are real-valued. If some of the eigenvalues are close to zero (below a defined threshold), we consider that the solution is under-determined. It may be possible to find other optimal solutions in the direction of the eigenvector associated with near-zero eigenvalue.

If the Hessian matrix is unavailable or expensive to compute, it is still possible to use its approximation. Since BFGS maintains an approximation of the inverse of the Hessian matrix, it is possible to use its pseudo-inverse to determine if the problem is under-determined and in which direction we need to explore the optimality region.

6.6. Global Optimization

The objective function can become non-linear and non-convex with multiple local minima e.g., for a complex model or if the user edits are far from the initial configuration. A practical example is a robotic arm when the user decides to move the end effector very far from its current location. To account for the difficult inverse problems, we implemented a global optimization algorithm.

The global optimization uses the Basin-hopping method [WD97] and adapts it to work with bounded objective functions. Basin-hopping is a two-phase approach that combines global random steps with local minimization. Global steps are accepted or discarded using a temperature parameter, similar to simulated annealing [KGV83]. We adjust the Basin-hopping method by changing the way global steps are taken. We normalize the length of steps according to the range of parameters and make sure steps that are out of bounds are discarded.

This solution is suitable for our tasks because the minimized objective functions are mostly continuous and smooth, as we only track affine transformations, and minimize the squared difference to the target configuration.

6.7. Exploring the Optimality Region

A regularization term is usually used during optimization to address the problem of multiple solutions when solving the inverse problem with a non-injective forward operator (i.e., an underdetermined edit), but this strategy privileges one solution over others [Gle94, MB21]. Instead, we explore the optimality region and present the user with various configurations to choose from. This is made possible by the differentiable representation, which significantly improves the speed of optimization.

We adjust the Basin-hopping algorithm to allow for the optimality region exploration. The step size is decreased to favor random steps that stay close to the optimality region. The temperature is decreased so that there is a low probability of accepting a jump to a less optimal region. Finally, we bias the global random steps to be more likely to go in the direction of the optimality region. When taking a global random step from an optimal point, we find the eigenvectors of the Hessian matrix and fine-tune the step size in the direction of eigenvectors. An eigenvector associated with a small eigenvalue will be assigned a bigger step size than an eigenvector associated with a bigger eigenvalue (see Section 7 for values of hyper-parameters).

The adjustments made to the Basin-hopping method ensure that the algorithm is more likely to stay close to the optimality region and minimize the number of local optimization steps needed to return to the optimality region if we randomly jump too far from it. The adjusted parameters have an impact on the speed of exploration. Thus, a bad set of parameters will not make the algorithm fail. Simply, all things equal, the set of optimal configurations will be less diverse with non-optimal parameters because the algorithm will not have time to reach as many interesting configurations in the budgeted time.

Exploring the optimality region can be viewed as a random walk, and there is a chance that the algorithm will come back to its starting point. However, the probability of coming to the starting point quickly decreases for higher dimensions, so our algorithm will most likely find more diverse solutions [Pól21].

6.8. Grouping and Ordering Solutions

After the global or exploration phases are completed, the optimizer's output is either a unique optimal solution or several solutions. If multiple solutions are found, we group and order solutions for the user to review them quickly.

First, we use a kd-Tree to remove duplicate optimal points. Then we go through the list of all optimal points and find a set of recommendations for the user to look at in priority. Following is the list of particular solutions that we present to the user. See Figure 14 in the appendix for images corresponding to the recommended solutions.

The nearest solution (L2 norm) to the starting point x_0. This is the solution that has the overall least amount of change in parameters. Note that even though it can be a good approximation, this solution is not the one with the least change in 3D shape. Finding the solution that minimizes the Chamfer distance [FSG17] would be prohibitively expensive as it would require to generate the 3D shape for all optimal points.

The farthest solution (L2 norm) from the starting point x_0. This is the solution that has the largest amount of change in parameters. It probably looks very different from x_0, yet still satisfies the edit.

The solution with the most delta-like change: a constant amount is added to all coordinates of x_0. A geometrical interpretation of this solution is the optimal point closest to a line $p(t) = x_0 + t(1,1,\ldots,1), \forall t$, i.e., starting from x_0 and going in the direction of the all-ones vector.

The solution with the most proportional change: a constant multiplication factor is applied to all parameters. A geometrical interpretation of this solution is the optimal point closest to a line $p(t) = t \cdot x_0 \forall t$, i.e., passing through the origin and x_0.

The solution with the least amount of change over one parameter keeps one variable of the problem constant and satisfies the edit only using other parameters.

Once we find a set of recommended configurations, we go over all optimal points and group them into independent optimality regions. This step is important in the case that the global optimization finds more than one optimality region (see Figure 12 in the appendix). To find groups of optimal points, we use the DBSCAN [EKS*96] clustering, with an epsilon set to a multiple of the step size used during global optimization. Once optimal points are grouped, we find an order to present them to the user. This step is important because the random walk over the optimality region does not guarantee that optimal points are ordered in a semantically meaningful way. We chose to use a heuristic solution to this problem to maximize the response time of our interactive system at the expense of raw precision. For each group of optimal points, we run an approximate k-medoid clustering [Mar64, PJ09] and find eight representative points. These are representative points that are uniformly spread among all optimal points.

We then find the shortest Hamiltonian path from the eight representative points, which gives us the shortest path going through the eight points without a cycle. To order all optimal points, we project them on the line segment formed by the shortest Hamiltonian path and sort them according to their parametric coordinate (See Figure 18). This ordering has more semantic meaning than a random order since it takes a path from one point in the optimality region to another point. See Figure 14 and Figure 15 in the appendix for an example of ordering. Experimentally, this strategy works particularly well when the optimality region is a 1D subspace. In this case, the order will follow the manifold defined by the optimality region. Suppose the optimality region is a subspace of higher dimensions. In that case, our heuristic will not disentangle the dimensions, but it will still most likely follow the dimension that has the longest extended and pack the other dimensions in-between.

6.9. Discontinuous Parameters

Not all models are generated by a fully differentiable procedural graph. An example is when non-differentiable nodes are used or when a legacy graph is not upgraded to differentiable nodes. As a fallback, our system implements a derivative-free optimizer if at least one of the target OBB is not auto differentiable. In this case, the system is still functional, but it is considerably slower and thus not interactive.

7. Implementation

We implemented ADPM in Python as an add-on to Blender® version 2.93 LTS. Results were generated on a laptop with an Intel Core i7-10875H (8 cores @ 2.30 GHz, turbo up to 5.1 GHz), 32 GB RAM with an Nvidia RTX 2060 Max-Q. The Blender® Cycles engine was used to render the images. Our code and example graphs are hosted on GitHub: https://github.com/mgaillard/Sorcar

The implementation of our inverse modeling feature is based on the existing procedural modeling add-on Sorcar [Aac21] that uses the node system to call Blender's API and generate the procedural models. Sorcar provides only the forward procedural model generation, and we extended Blender's scene tree and its objects with differentiable transformation matrices and OBBs (Section 5). Sorcar still keeps its full features and functionality, in addition to the inverse modeling capability for editing.

Sorcar originally features hundreds of different nodes for procedural modeling. Besides, high-level nodes were added, which combine many operations that make modeling faster (see Section 5). We also added our input variable nodes that allow the optimizer to interact with parameters in the procedural graph $\mathcal{G}$. We implemented dedicated input variable nodes because it makes it simpler for the optimizer to list all individual input parameters $x_i \in \mathcal{I}$ of the procedural graph $\mathcal{G}$. Input variable nodes also let the user add box constraints on the parameters during optimization. By default, individual parameters are unconstrained $x_i \in \mathcal{R}$, but they can also be bounded $min \leq x_i \leq max$. It is possible to set an input variable as constant to freeze parts of the model when editing.

We use the optimization package from SciPy [VGO*20] to solve the inverse problem. We customized their Basin-hopping implementation by replacing routines for taking random global steps, accepting steps, and local minimization. Different methods are used for local optimization, depending on the characteristics of the inverse problem to solve. When all variables are unbounded, we use BFGS [NW06]. If at least one input parameter is bounded, we use L-BFGS-B [BLNZ95]. If parts of the model are not augmented with auto differentiation, we use the derivative-free optimization method Nelder-mead [NM65] for unbounded problems, and a modified version of Powell [Pow64, PTVF07] for bounded problems. For Basin-hopping, the temperature is set to 1.0 for global optimization and 0.1 for exploration. The step size is set to 0.25 for global optimization and 0.05 for exploration. These values are expressed in terms of the percentage of the range for each parameter. In other words, a value of 0.05 means 5% of the total range in each direction. When exploring with the Hessian activated, we double the step size in the direction of the eigenvector associated with the lowest eigen value. We did not fine-tune the hyper-parameters, and mostly kept default values. The only adverse effect of choosing sub-optimal values is that with equal budget of time, sampling of the optimality region will tend to be less diverse. We use the following values for grouping and ordering of parameters: the threshold for deciding if two points are duplicates is $1e^{-2}$. For DBSCAN, $\varepsilon = 1.0$, and the minimum number of points per cluster is one.

We used the CasADi [AGH*19] auto differentiation library that is a C++ backend for automatic differentiation, but it has a Python frontend to interface with Blender®. Various auto differentiation libraries exist, but CasADi implements the possibility of building the objective function once and then evaluating it multiple times with different values. This allows ADPM to make the differentiable representation only once when the procedural graph $\mathcal{G}$ is executed. Then the objective function can be compiled just-in-time and optimized separately. Optimizing the function separately is efficient because while solving the inverse problem, the optimizer can focus only on the necessary operations instead of also handling the 3D geometry. Thanks to this feature, the time required to optimize any model presented in this paper was in the order of milliseconds (see Table 1).

Building a differentiable procedural graph requires comparable effort to traditional non-differentiable graphs. We prevent invalid configuration by correctly setting types of data that can go through nodes via sockets and edges. The technical artist designing the ADPM does not need to understand how the differentiable representation works, and the end-user does not need to edit the procedural model. At any moment, the differentiable representation is invisible to the user.

8. Results and Evaluation

8.1. Results

We show various examples created by a technical artist that demonstrate the usability of our approach. The procedural graphs are fully editable with ADPM. In all examples, except for materials, everything is procedurally generated by ADPM.

Table 1 shows statistics and a performance breakdown for all examples from this paper. The number of nodes of the procedural graph ranges from 31 to 164, the number of links from 43 to 234, and the number of editable parameters from one to 30. The generation time for the most complex scene in our examples (Figure 9) is 553 ms on average. The time for generation and optimization of the simple examples (Figure 7 and Figure 8) is under a second.

The procedural **stool** (Figure 6) shows the versatility of the editing capabilities of ADPM. It is composed of a wooden seat and four metal legs. Its editable parameters include the seat size, thickness, the length and radius of legs, and the distance between the top of the legs and the side of the seat. The user can reconfigure the model with a few mouse clicks, using a transformation gizmo common in 3D modeling software to manipulate individual parts of the model (see Figure 1). The resulting model will adhere to constraints defined by the procedural graph, for example, the leg thickness is preserved regardless of the size of the top. We extended the procedural model of the stool to lay a Suzanne head from Blender® on top of it. By raising the head, we trigger an under-determined edit because the stool can be made taller by either adjusting the legs' height, or the seat thickness, or a combination of both. Figure 5 shows the suggestions made by ADPM. This type of under-determined edit is encountered when two variables equally contribute to the same value. We show a detailed analysis of an abstraction of this example with a stack of three cubes in Figure 13 of the appendix.

The **sofa** in Figure 7 is composed of a base wooden plank, four metal legs, two armrests on each side, a wooden back, and four blue

Figure 5: ***Stool:*** *a) A procedural stool with the Suzanne head from Blender®. b) The user edits the model by raising the Suzanne head. Image of the c)* nearest*, d)* farthest*, e)* most delta-like change*, f) most proportional change, and g)* constant legs' height *solution. These images were used in the user study.*

	Input			Output		Optimization [ms]				
Model	# of nodes	# of links	# of params	# of objects	Scene depth	Gen	Local	Under-determined	Global / Exploration	Clustering / Ordering
Three cubes (Figure 13)	27	30	2	3	1	51	2.0	1.3	410	310
Stool (Figure 3)	63	89	5	5-6	1	113	1.7	1.6	353	410
Sofa (Figure 7)	86	110	9	12	3	513	3.0	2.1	943	N/A
Robotic arm (Figure 10)	95	108	7	8	8	489	8.6	4.7	6,084	26.8
Succulent (Figure 11)	32	44	12	48	2	163	10.0	1.4	2,858	1,149
Centipede (Figure 9)	164	234	30	22	7	553	11.9	2.7	3,526	401

Table 1: ***Statistics for the generated models:*** *Input includes the number of nodes and links that form the procedural graph. The column number of parameters gives the number of input variables of each model. The output shows the complexity of the generated 3D scene: number of objects and scene depth, which is the maximum number of parents an object can have in the scene. Optimization shows timings decomposed into generation (i.e., time to build the model, the differentiable representation, and display it on the screen), local optimization (see Section 6.4), identification of the under-determined nature of the edit (see Section 6.5), global optimization or exploration of the region of optimality (see Section 6.6), clustering and ordering of points (see Section 6.8). The clustering and ordering column for the sofa model is N/A because it has been designed to make it impossible to trigger an under-determined edit. Note that the budget for global optimization/exploration was set to 400 local minimization steps. Hence it takes roughly 400 times longer than local optimization.*

Figure 6: ***Stool:*** *A procedural stool on the left can be replicated and the scene can be arranged by a few editing operations so that it becomes a table with two stools.*

Figure 7: ***Sofa*** *is parameterized to be symmetric. The two armrests are parameterized by a single parameter to allow for larger changes with a simple operation. See Figure 1 for an example of an edit of the sofa.*

leather cushions. It is editable by several parameters that include: the overall width and depth of the sofa, the thickness of all wooden parts, the size of armrests, the size of cushions.

Some parts of the model are constrained by the same input parameters. For example, the two armrests must have the same size to be symmetric. By re-using the same input parameters at different locations in the procedural model and adding box constraints on the input parameters, we constrain the set of possible models $\mathcal{S}$ to include only visually pleasing models. This ensures that the model will always keep its consistency, after the inverse problem is solved (see Figure 8). Cushions were procedurally generated with a cloth simulation that inflates the cushion meshes with internal springs. A custom node triggers the cloth simulation as a post-processing step. The edits were made using a transform gizmo, as shown in Figure 1 and the supplementary video.

Figure 9 shows a simplified **centipede** (hexipede) with three pairs of legs. It consists of a head and three blocks, each composed of a body element and two legs. This example shows that ADPM can have multiple procedural graphs interacting with each other. A graph generates the terrain, and another graph generates the centipede. To ensure that the centipede is always touching the ground, we designed a node to trigger the update and optimization of the other graph automatically. Thus, after the generation of the terrain and the centipede, the centipede is optimized with 1) the target for its head that stays fixed, and 2) the target for each leg is set to touch the ground and be oriented according to the terrain's normal (see the supplementary video).

This example has been designed to be an adversarial case for optimization and demonstrates the speedup enabled by the differ-

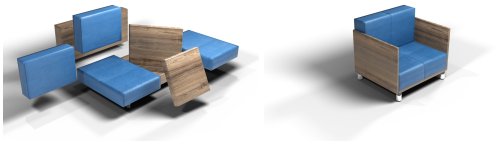


Figure 8: ***Sofa:*** *The sofa from Figure 7 was edited by a careless user and set as a target for optimization (left). The same sofa after the optimization (right). Solving the inverse problem with even the most adversarial edits will always yield a plausible model thanks to bounded optimization and the fact that the procedural graph generates only valid models.*

Figure 9: ***Centipede:*** *scene includes two procedural models, one for the terrain and one for the centipede. The head of the centipede is fixed and its legs are automatically optimized to touch the terrain (see the supplementary video).*

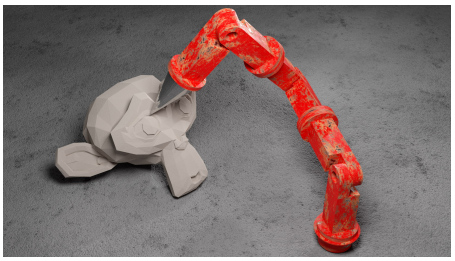


Figure 10: ***Robotic arm:*** *Suzanne head from Blender® (left-hand side) and a fully procedural robotic arm constrained to point at the head (right-hand side).*

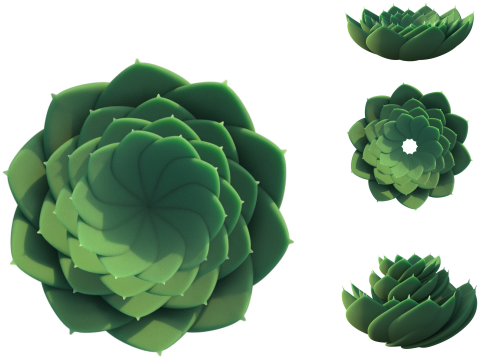


Figure 11: ***Succulent:*** *A plant generated using the Radius Scatter node of our system.*

entiable representation. The model consists of 30 different input parameters (two DOF per body element, four DOF per leg). The objective function is highly non-linear, as it includes rotations in the joints that are bounded. For instance, the last leg segments depend on a hierarchy of seven rotations. When the user edits a single leg, solving the inverse problem takes 11.9 ms on average. When optimizing all legs to touch the ground, the time increases to about 100-200 ms (on average: 128 ms). An artist would need much longer to manually change the parameters of the graph.

Another example of a procedural model created using ADPM is the **robotic arm** shown in Figure 10. Each element is procedurally generated from scratch using nodes implemented in ADPM. Basic elements are later duplicated and assembled to form the arm. The robotic arm is automatically animated to point its end effector (a metal cone) at the nearest vertex on Suzanne's head. It is easy to add elements to the robotic arm by copying the nodes to make it longer. This example shows that our system can be used to solve inverse problems similar to inverse kinematics. Additionally, the robotic arm has enough degrees of freedom to trigger underdetermined edits. Figure 12 from the appendix shows how global optimization finds two different solutions. Figure 16 from the appendix shows different suggestions made by our system. See the supplementary video for an animation of optimal points sampled from the optimality region.

Figure 11 shows a **succulent** plant. The procedural graph replicates and transforms a single leaf object to make a plant. Procedural parameters include: number of layers, number of leaves per layer, initial radius, offset and phase of successive layers. See the video for an interactive editing session of this procedural model.

Timing: The evaluation of a Blender procedural graph takes 20-600 ms (see Table 1). The time needed to evaluate a procedural graph depends on the complexity of the model. Increasing the geometric complexity and object transformations requires longer evaluation times. Graph-based procedural modeling is only used in forward mode, that is why evaluation speed is usually not essential, because graphs are rarely re-executed. Sorcar, which we build upon, is a traditional graph-based procedural engine, as such it is not optimized for maximum execution speed, because it relies on the internal Blender® API. If we were to implement a similar tool as ADPM without auto-differentiation, evaluation speed would be critical since solving the inverse problem requires thousands executions of the graph in a short period of time. Optimizing the internal node engine of Blender® is an engineering challenge that we chose not to tackle. Instead, we focused on the acceleration of optimization by building a separate differentiable representation on top of Sorcar. This way, we can solve the inverse problem separately from Blender® with automatically computed gradient, which is order of magnitude faster than finite differences (see Table 1 for optimization times). With ADPM, since the model is updated after optimization, the total time between the modification and the display of the updated model is the time for optimization plus the time for execution. We implemented an interactive editing tool within Blender®.

As soon as the user changes the model, our inverse procedural optimizer is triggered and parameters are modified to best match the edit. See supplementary video for a recording of an editing session.

8.2. User Studies

Qualitative User study was conducted with five users. After a short introduction to procedural modeling and the motivation behind our approach, we let the users practice on a simple model (the three cubes example). Users were first shown how to navigate, edit, and change the procedural model. Then they were allowed to practice and ask questions to ensure that they understood the system.

After the introduction, they were tasked to perform sofa editing from Figure 7. We placed an existing configuration as an overlay image on the viewport with transparency. The users were asked to edit the model to match it. After the study, the users were asked to fill a survey asking a question about themselves and evaluate their experience on a four point Likert scale (2 strongly agree, 1 somewhat agree, -1 somewhat disagree, -2 strongly disagree). The scale does not include a neutral option thus forcing the users to lean either towards positive or negative. The participants identified themselves as *experts in 3D modeling*: 0.4 on average and *experienced in procedural modeling*: -0.2; In other words, they were, on average inexperienced with procedural modeling. Overall user qualifications regarding 3D modeling and procedural modeling were balanced, with both expert and beginners profiles.

When asked if ADPM were easy to use, the user responded 0.8 on average. Regarding whether edits matched their expectations, users answered 1.4 on average. When asked if it were easy to match the target configuration, users replied 0.8 on average. One participant reported that it is very easy to get a result similar to the target quickly, but fine-tuning the parameters to exactly match the target from a single projected view is difficult. Finally, we asked users which way of editing they preferred (1) only manual editing of parameters, 2) mostly manual + some automatic with ADPM, 3) mostly automatic with ADPM and some manual, 4) fully automatic with ADPM). Three of them replied that their preferred option is 3) two answered 2). The user study shows that non-experts users were able to quickly and effectively use an advanced procedural model to achieve rather complex modeling tasks.

Quantitative User Study was conducted with 21 users. Users were presented with a several procedural models (Stool, Cubes, Robotic arm, and Succulent) along with an under-determined edit made to them. Images of the various suggestions from our optimizer were shown and users were asked which one they visually preferred (See Figure 5 for examples of images used for this user study). This study shows that in case the edit is under-determined, users are interested in suggestions other than the default solution (i.e., the solution found during the first local optimization phase). Results are shown in Table 2 in the appendix. We ran a Chi-squared statistical analysis to show that the distribution is not uniformly random, thus users tend to have preferences for certain solutions. Finally, we can see that users do not always pick the result of the local optimization phase, which justifies that exploring the optimality region is interesting for users.

9. Conclusion, Limitations, and Future Work

We introduced ADPM, an auto differentiable procedural modeling system for node-based procedural models. We construct a proxy graph that operates on a simplified representation of generated object-oriented bounding boxes (OBBs). We track all operations performed on the OBBs during procedural generation and construct a computational graph. The user is then free to manipulate the generated output model directly in the viewport. When the edit is done, we define an objective function, whose minimization implies that OBBs are located were moved by the user. This allows us to find the parameters of the procedural model that match the user edit. If the edit is under-determined and has degrees of freedom, our optimizer detects it and makes various suggestions. The user performs non-destructive interactive modifications of a procedural model in a what you see is what you get workflow. The user does not need to know procedural modeling yet can use it. We have shown ADPM on several examples, and we performed a user study that found that users could use the system effectively and preferred it over manually adjusting parameters. We also show that users are interested in getting various suggestions for edits that are under-determined.

Our work has several limitations. The augmentation of a graph-based procedural model is done manually, and it would be interesting to implement an automatic conversion procedure. We limit ourselves to a high-level primitive hierarchy and a finer vertex-level granularity could be achieved by trading off speed and implementation complexity. An autodiff representation lends itself well to integration into novel machine learning pipelines. It would be interesting to explore ways to increase the granularity of edits and support more geometric operations. Our system does not homogenize parameters when identifying recommended solutions. For instance, angles can be expressed in radians or degrees, and a unit of 1 may mean something different whether it represents an angle, a length or an area. Currently it is up to the technical artists to provide a meaningful parametrization by converting parameters using the math operation node. As a future work, we could add a semantic meaning and a unit to each input parameter, and make custom recommendations based on that. For example, we could favor proportional changes for length parameters, and delta-like changes for angle parameters. Also, we could add support for periodic boundary conditions, useful for angle parameters. Another possible future work would be to speed up global optimization and exploration of the region of optimality by running multiple instances of the algorithm in parallel with different random seeds. In addition to 3D gizmos, we could implement other editing metaphors like brush strokes or sketching. Having a UI that better exposes solutions sampled from the region of optimality is an interesting future direction. There might be other ways to identify interesting solutions: e.g., solutions with inverse proportional dimensions, solutions close to a canonical pose, and solutions with similar input parameters. Regarding the grouping and ordering of points, our system cannot identify the dimensionality of the optimality region. It would be interesting to try more powerful algorithms to explore and map the structure of the optimality region, like non-linear PCA [SSM98].

Acknowledgments: This research was funded in part by National Science Foundation grant #10001387, *Functional Proceduralization of 3D Geometric Models*.

References

[AAB*16] ABADI M., AGARWAL A., BARHAM P., BREVDO E., CHEN Z., CITRO C., CORRADO G. S., DAVIS A., DEAN J., DEVIN M., ET AL.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016). 3

[Aac21] AACHMAN P.: Sorcar. https://github.com/aachman98/Sorcar, 2021. 8

[Ado21] ADOBE: Substance. https://www.substance3d.com/products/substance-designer/, 2021. 2, 4

[AGH*19] ANDERSSON J. A. E., GILLIS J., HORN G., RAWLINGS J. B., DIEHL M.: CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation 11*, 1 (2019), 1–36. 8

[AK84] AONO M., KUNII T. L.: Botanical tree image generation. *IEEE Computer Graphics and Applications 4*, 5 (May 1984), 10–34. 2

[BLNZ95] BYRD R. H., LU P., NOCEDAL J., ZHU C.: A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing 16*, 5 (1995), 1190–1208. 6, 8

[BŠMM11] BENES B., ŠT'AVA O., MĚCH R., MILLER G.: Guided procedural modeling. In *Comp. Graph. Forum* (2011), vol. 30, Wiley Online Library, pp. 325–334. 2

[BWS10] BOKELOH M., WAND M., SEIDEL H.-P.: A connection between partial symmetry and inverse procedural modeling. In *ACM SIGGRAPH 2010 papers*. 2010, pp. 1–10. 2

[CGT00] CONN A. R., GOULD N. I., TOINT P. L.: *Trust region methods*. SIAM, 2000. 6

[CSQ*22] CASCAVAL D., SHALAH M., QUINN P., BODIK R., AGRAWALA M., SCHULZ A.: Differentiable 3d cad programs for bidirectional editing. vol. 41, Wiley Online Library. 2, 3

[dABPSA*18] DE AVILA BELBUTE-PERES F., SMITH K., ALLEN K., TENENBAUM J., KOLTER J. Z.: End-to-end differentiable physics for learning and control. In *Advances in neural information processing systems* (2018), pp. 7178–7189. 2, 3

[DHD*19] DEGRAVE J., HERMANS M., DAMBRE J., ET AL.: A differentiable physics engine for deep learning in robotics. *Frontiers in neurorobotics 13* (2019), 6. 3

[DIP*18] DU T., INALA J. P., PU Y., SPIELBERG A., SCHULZ A., RUS D., SOLAR-LEZAMA A., MATUSIK W.: Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Transactions on Graphics (TOG) 37*, 6 (2018), 1–16. 3

[EKS*96] ESTER M., KRIEGEL H.-P., SANDER J., XU X., ET AL.: A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd* (1996), vol. 96, pp. 226–231. 7

[EMP*03] EBERT D. S., MUSGRAVE F. K., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modeling*, 3rd ed. Academic Press, Inc., Orlando, FL, USA, 2003. 1

[ERSLT18] ELLIS K., RITCHIE D., SOLAR-LEZAMA A., TENENBAUM J.: Learning to infer graphics programs from hand-drawn images. In *NIPS 31*, Bengio S., Wallach H., Larochelle H., Grauman K., Cesa-Bianchi N., Garnett R., (Eds.). Curran Associates, Inc., 2018, pp. 6060–6069. 3

[EVC*15] EMILIEN A., VIMONT U., CANI M.-P., POULIN P., BENES B.: Worldbrush: Interactive example-based synthesis of procedural virtual worlds. *ACM Trans. Graph. 34*, 4 (July 2015), 106:1–106:11. 2

[FFC82] FOURNIER A., FUSSELL D., CARPENTER L.: Computer rendering of stochastic models. *Commun. of ACM 25*, 6 (June 1982), 371–384. 2

[FL19] FEY M., LENSSEN J. E.: Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428* (2019). 3

[FSG17] FAN H., SU H., GUIBAS L. J.: A point set generation network for 3d object reconstruction from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 605–613. 7

[G*89] GRIEWANK A., ET AL.: On automatic differentiation. *Mathematical Programming: recent developments and applications 6*, 6 (1989), 83–107. 3, 4

[GGP*19] GALIN E., GUÉRIN E., PEYTAVIE A., CORDONNIER G., CANI M.-P., BENES B., GAIN J.: A review of digital terrain modeling. *Comp. Graph. Forum 38*, 2 (2019), 553–577. 2

[GHYZ20] GUO Y., HAŠAN M., YAN L., ZHAO S.: A bayesian inference framework for procedural material parameter estimation. In *Comp. Graph. Forum* (2020), vol. 39, Wiley Online Library, pp. 255–266. 3

[GJB*20] GUO J., JIANG H., BENES B., DEUSSEN O., ZHANG X., LISCHINSKI D., HUANG H.: Inverse procedural modeling of branching structures by inferring l-systems. *ACM Trans. Graph. 39*, 5 (2020), 1–13. 3

[Gle94] GLEICHER M. L.: *A differential approach to graphical interaction*. Carnegie Mellon University, 1994. 3, 7

[HAL*19] HU Y., ANDERSON L., LI T.-M., SUN Q., CARR N., RAGAN-KELLEY J., DURAND F.: Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935* (2019). 3

[HDR19] HU Y., DORSEY J., RUSHMEIER H.: A novel framework for inverse procedural texture modeling. *ACM Trans. Graph. 38*, 6 (2019), 1–14. 3

[HKUT20] HOLL P., KOLTUN V., UM K., THUEREY N.: phiflow: A differentiable pde solving framework for deep learning via physical simulations. In *NeurIPS Workshop* (2020). 2

[HKYM16] HUANG H., KALOGERAKIS E., YUMER E., MECH R.: Shape synthesis from sketches via procedural models and convolutional networks. *IEEE TVCG 23*, 8 (2016), 2003–2013. 3

[HSS17] HAUBENWALLNER K., SEIDEL H.-P., STEINBERGER M.: Shapegenetics: using genetic algorithms for procedural modeling. In *Comp. Graph. Forum* (2017), vol. 36, Wiley Online Library, pp. 213–223. 2

[IOI06] IJIRI T., OWADA S., IGARASHI T.: *The Sketch L-System: Global Control of Tree Modeling Using Free-Form Strokes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 138–146. 3

[JBX*20] JONES R. K., BARTON T., XU X., WANG K., JIANG E., GUERRERO P., MITRA N., RITCHIE D.: Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Transactions on Graphics (TOG), Siggraph Asia 2020 39*, 6 (2020), Article 234. 3

[JHM04] JOHNSTON W. M., HANNA J. P., MILLAR R. J.: Advances in dataflow programming languages. *ACM computing surveys (CSUR) 36*, 1 (2004), 1–34. 2

[KBM*20] KATO H., BEKER D., MORARIU M., ANDO T., MATSUOKA T., KEHL W., GAIDON A.: Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057* (2020). 2, 3

[KGV83] KIRKPATRICK S., GELATT C. D., VECCHI M. P.: Optimization by simulated annealing. *science 220*, 4598 (1983), 671–680. 6

[KK11] KRECKLAU L., KOBBELT L.: Procedural Modeling of Interconnected Structures. *Comp. Gr. Forum* (2011). 2

[KLMK19] KALOJANOV J., LIM I., MITRA N., KOBBELT L.: String-based synthesis of structured shapes. In *Comp. Graph. Forum* (2019), vol. 38, Wiley Online Library, pp. 27–36. 2

[KMG*20] KRS V., MECH R., GAILLARD M., CARR N., BENES B.: Pico: Procedural iterative constrained optimizer for geometric modeling. *IEEE TVCG* (2020), 1–1. 2

[KUH18] KATO H., USHIKU Y., HARADA T.: Neural 3d mesh renderer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 3907–3916. 3

[LADL18] LI T.-M., AITTALA M., DURAND F., LEHTINEN J.: Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph. 37*, 6 (2018), 1–11. 3

[LB14] LOPER M. M., BLACK M. J.: Opendr: An approximate differentiable renderer. In *European Conference on Computer Vision* (2014), Springer, pp. 154–169. 3

[LBZ*10] LI Y., BAO F., ZHANG E., KOBAYASHI Y., WONKA P.: Geometry synthesis on surfaces using field-guided shape grammars. *IEEE TVCG 17*, 2 (2010), 231–243. 2

[LHK*20] LAINE S., HELLSTEN J., KARRAS T., SEOL Y., LEHTINEN J., AILA T.: Modular primitives for high-performance differentiable rendering. *ACM Transactions on Graphics 39*, 6 (2020). 3

[Lin68] LINDENMAYER A.: Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of theoretical biology 18*, 3 (1968), 300–315. 2

[Lin76] LINNAINMAA S.: Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics 16*, 2 (1976), 146–160. 5

[LLK19] LIANG J., LIN M., KOLTUN V.: Differentiable cloth simulation for inverse problems. In *Advances in Neural Information Processing Systems* (2019), pp. 772–781. 3

[LRBP12] LONGAY S., RUNIONS A., BOUDON F., PRUSINKIEWICZ P.: Treesketch: interactive procedural modeling of trees on a tablet. In *Proc. of the Intl. Symp. on SBIM* (2012), pp. 107–120. 3

[LSL*19] LIPP M., SPECHT M., LAU C., WONKA P., MÜLLER P.: Local editing of procedural models. In *Comp. Graph. Forum* (2019), vol. 38, Wiley Online Library, pp. 13–25. 3

[LVW*15] LIU H., VIMONT U., WAND M., CANI M.-P., HAHMANN S., ROHMER D., MITRA N. J.: Replaceable substructures for efficient part-based modeling. In *Computer Graphics Forum* (2015), vol. 34, Wiley Online Library, pp. 503–513. 3

[LW19] LIU Y., WU Z.: Learning to describe scenes with programs. In *International Conference on Learning Representations* (2019). 3

[LWJ*18] LI Y., WANG G., JI X., XIANG Y., FOX D.: Deepim: Deep iterative matching for 6d pose estimation. In *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 683–698. 6

[MA21] MCNEEL R., ASSOCIATES: Grasshopper. https://www.grasshopper3d.com, 2021. 2, 4

[Man82] MANDELBROT B. B.: *The fractal geometry of nature*, vol. 2. WH freeman New York, 1982. 2

[Mar64] MARANZANA F.: On the location of supply points to minimize transport costs. *Journal of the Operational Research Society 15*, 3 (1964), 261–270. 7

[MB21] MICHEL E., BOUBEKEUR T.: Dag amendment for inverse control of parametric shapes. *ACM Transactions on Graphics 40*, 4 (2021), 173:1–173:14. 2, 3, 7

[MKC18] MAYER M., KUNCAK V., CHUGH R.: Bidirectional evaluation with direct manipulation. *Proceedings of the ACM on Programming Languages 2*, OOPSLA (2018), 1–28. 3

[MPZ20] MATHUR A., PIRRON M., ZUFFEREY D.: Interactive programming for parametric cad. In *Computer Graphics Forum* (2020), vol. 39, Wiley Online Library, pp. 408–425. 3

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*. 2006, pp. 614–623. 2

[NGDA*16] NISHIDA G., GARCIA-DORADO I., ALIAGA D. G., BENES B., BOUSSEAU A.: Interactive sketching of urban procedural models. *ACM Trans. Graph. 35*, 4 (2016), 1–11. 3

[NM65] NELDER J. A., MEAD R.: A simplex method for function minimization. *The computer journal 7*, 4 (1965), 308–313. 8

[NW06] NOCEDAL J., WRIGHT S.: *Numerical optimization*. Springer Science & Business Media, 2006. 6, 8

[Pat10] PATOW G.: User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications 32*, 2 (2010), 66–75. 2

[PGC*17] PASZKE A., GROSS S., CHINTALA S., CHANAN G., YANG E., DEVITO Z., LIN Z., DESMAISON A., ANTIGA L., LERER A.: Automatic differentiation in pytorch. 3

[PHL*09] PALUBICKI W., HOREL K., LONGAY S., RUNIONS A., LANE B., MĚCH R., PRUSINKIEWICZ P.: Self-organizing tree models for image synthesis. *ACM Trans. on Graph. 28*, 3 (2009), 58:1–58:10. 3

[PJ09] PARK H.-S., JUN C.-H.: A simple and fast algorithm for k-medoids clustering. *Expert systems with applications 36*, 2 (2009), 3336–3341. 7

[PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic topiary. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), pp. 351–358. 2

[Pól21] PÓLYA G.: Über eine aufgabe der wahrscheinlichkeitsrechnung betreffend die irrfahrt im straßennetz. *Mathematische Annalen 84*, 1 (1921), 149–160. 7

[Pow64] POWELL M. J.: An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal 7*, 2 (1964), 155–162. 8

[PTVF07] PRESS W. H., TEUKOLSKY S. A., VETTERLING W. T., FLANNERY B. P.: *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007. 8

[RMGH15] RITCHIE D., MILDENHALL B., GOODMAN N. D., HANRAHAN P.: Controlling procedural modeling programs with stochastically-ordered sequential monte carlo. *ACM Trans. Graph. 34*, 4 (2015), 1–11. 2

[RMP*20] RIBA E., MISHKIN D., PONSA D., RUBLEE E., BRADSKI G.: Kornia: an open source differentiable computer vision library for pytorch. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision* (2020), pp. 3674–3683. 3

[RTHG16] RITCHIE D., THOMAS A., HANRAHAN P., GOODMAN N.: Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances in Neural Information Processing Systems 29*. 2016, pp. 622–630. 2

[ŠBM*10] ŠT'AVA O., BENES B., MĚCH R., ALIAGA D. G., KRIŠTOF P.: Inverse procedural modeling by automatic generation of l-systems. In *Comp. Graph. Forum* (2010), vol. 29, Wiley Online Library, pp. 665–674. 2

[SEBC15] SILVA P. B., EISEMANN E., BIDARRA R., COELHO A.: Procedural content graphs for urban modeling. *International Journal of Computer Games Technology 2015* (2015). 2

[SF18] SCHENCK C., FOX D.: Spnets: Differentiable fluid dynamics for deep neural networks. In *Conference on Robot Learning* (2018), PMLR, pp. 317–335. 3

[SG71] STINY G., GIPS J.: Shape grammars and the generative specification of painting and sculpture. In *Segmentation of Buildings for 3D Generalisation. In: Proceedings of the Workshop on generalisation and multiple representation , Leicester* (1971). 2

[SGL*18] SHARMA G., GOYAL R., LIU D., KALOGERAKIS E., MAJI S.: Csgnet: Neural shape parser for constructive solid geometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 5515–5523. 3

[Sid21] SIDEFX: Houdini. https://www.sidefx.com/products/houdini, 2021. 2, 4

[SLH*20] SHI L., LI B., HAŠAN M., SUNKAVALLI K., BOUBEKEUR T., MECH R., MATUSIK W.: Match: differentiable material graphs for procedural material capture. *ACM Trans. Graph. 39*, 6 (2020), 1–15. 4

[SM15] SCHWARZ M., MÜLLER P.: Advanced procedural modeling of architecture. *ACM Trans. Graph. 34*, 4 (2015), 1–12. 2

[SMBC13] SILVA P. B., MÜLLER P., BIDARRA R., COELHO A.: Node-based shape grammar representation and editing. In *Proceedings of the Workshop on Procedural Content Generation in Games (PCG'13)* (2013), pp. 1–8. 2

[SP16] SANTONI C., PELLACINI F.: gtangle: A grammar for the procedural generation of tangle patterns. *ACM Trans. Graph. 35*, 6 (2016), 1–11. 2

[Spe80] SPEELPENNING B.: *Compiling fast partial derivatives of functions given by algorithms*. Tech. rep., Illinois Univ., Urbana (USA). Dept. of Computer Science, 1980. 3

[SPK*14] STAVA O., PIRK S., KRATT J., CHEN B., MĚCH R., DEUSSEN O., BENES B.: Inverse procedural modelling of trees. In *Comp. Graph. Forum* (2014), vol. 33, Wiley Online Library, pp. 118–131. 2

[SSM98] SCHÖLKOPF B., SMOLA A., MÜLLER K.-R.: Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation 10*, 5 (1998), 1299–1319. 11

[STBB14] SMELIK R. M., TUTENEL T., BIDARRA R., BENES B.: A survey on procedural modelling for virtual worlds. In *Comp. Graph. Forum* (2014), vol. 33, Wiley Online Library, pp. 31–50. 2

[STdKB11] SMELIK R. M., TUTENEL T., DE KRAKER K. J., BIDARRA R.: A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics 35*, 2 (2011), 352 – 363. 3

[TLL*11] TALTON J. O., LOU Y., LESSER S., DUKE J., MĚCH R., KOLTUN V.: Metropolis procedural modeling. *ACM Trans. Graph. 30* (April 2011), 11:1–11:14. 2

[TLS*19] TIAN Y., LUO A., SUN X., ELLIS K., FREEMAN W. T., TENENBAUM J. B., WU J.: Learning to infer and execute 3d shape programs. *arXiv preprint arXiv:1901.02875* (2019). 3

[TYK*12] TALTON J., YANG L., KUMAR R., LIM M., GOODMAN N., MĚCH R.: Learning design patterns with bayesian grammar induction. In *Proceedings of the 25th annual ACM symposium on User interface software and technology* (2012), pp. 63–74. 2

[VAB10] VANEGAS C. A., ALIAGA D. G., BENES B.: Building reconstruction using manhattan-world grammars. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2010), IEEE, pp. 358–365. 2

[VGDA*12] VANEGAS C. A., GARCIA-DORADO I., ALIAGA D. G., BENES B., WADDELL P.: Inverse design of urban procedural models. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 168:1–168:11. 2

[VGO*20] VIRTANEN P., GOMMERS R., OLIPHANT T. E., HABERLAND M., REDDY T., COURNAPEAU D., BUROVSKI E., PETERSON P., WECKESSER W., BRIGHT J., VAN DER WALT S. J., BRETT M., WILSON J., MILLMAN K. J., MAYOROV N., NELSON A. R. J., JONES E., KERN R., LARSON E., CAREY C. J., POLAT İ., FENG Y., MOORE E. W., VANDERPLAS J., LAXALDE D., PERKTOLD J., CIMRMAN R., HENRIKSEN I., QUINTERO E. A., HARRIS C. R., ARCHIBALD A. M., RIBEIRO A. H., PEDREGOSA F., VAN MULBREGT P., SCIPY 1.0 CONTRIBUTORS: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17* (2020), 261–272. 8

[WD97] WALES D. J., DOYE J. P.: Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *The Journal of Physical Chemistry A 101*, 28 (1997), 5111–5116. 6

[WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant architecture. *ACM Trans. Graph. 22*, 3 (2003), 669–677. 2

[YRKC17] YE N., ROOSTA-KHORASANI F., CUI T.: Optimization methods for inverse problems. 5

10. Appendix

10.1. List of procedural nodes implemented in our system

Following is a list of nodes implemented in ADPM:

The **input variable node** has no input and only one output: an auto-differentiable value. This node is used to parametrize the procedural model. The node displays a few input widgets for the user to edits the properties of the input value. A name can be given to the variable. A check box sets the value to be a constant or a variable. Another check box sets the value to be bounded within a minimum and a maximum values. Finally, a slider allows the user to easily set the value of the variable.

The **convert object node** takes as input an object o_i and adds it to the auto differentiable representation of the scene. Upon execution, the node will assign an auto differentiable OBB to the object and track any subsequent transformation made on it.

The **transform node** takes as input an object o_i and a 3D vector and applies an affine transformation (translation, rotation, scaling) to o_i. The autodiff local transformation matrix M_i of the object o_i is also transformed.

The **apply transform node** takes as input an object o_i and applies its local transformation matrix M_i to o_i and its OBB B_i. Then 1) each vertex v of the object o_i becomes $M_i v$, 2) the OBB is transformed $M_i B_i$ and finally, 3) the local transformation matrix is reset to identity i.e.,$M_i = I_4$. This node does not transform the object, but it has an action on the scene tree. An example of its usage is when an object is scaled, but the user does not want the scaling to be propagated to the object's children.

The **joint node** takes as input two objects: a child and its parent. It translates the child object on one side of the parent's oriented bounding box. The user can choose one of the six sides with a drop-down menu $(\pm X, \pm Y, \pm Z)$. If the two bounding boxes have different sizes, it's also possible to align the child bounding box according to two other directions (left, center, right). An example of using this operation is connecting parts of an object, and we used this for the arms and legs of the centipede examples (See fig. 9).

The **radius scattering node** takes as input an object o_i, and duplicates it n times in a circle around the origin of its parent object. To control the layout of objects, the node takes additional input: 1) the radius of the circle, 2) the scale to apply to the object, 3) the phase of the rotation for the object. The radius scattering node is demonstrated with the procedural succulent in Figure 11.

The **matrix scattering node** takes as input an object o_i, and duplicates it $n \times m \times k$ times respectively on X, Y, Z axis. It is similar to the radius scattering node, except that it replicates the object on a grid instead of a circle.

The **math operation node** takes as input two auto differentiable values a and b and applies an operation to them. Operations that are available include: $a+b$, $a-b$, $a*b$, a/b, a^b. Some operations require only one input to be connected: $\log(a)$, $\sqrt{a}$, $-a$.

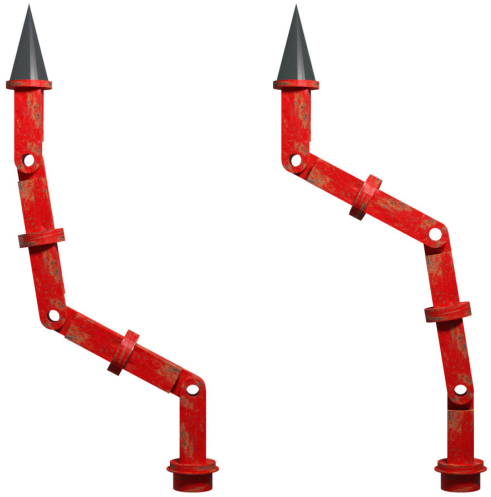


Figure 12: ***Robotic arm with two solutions:*** *We show the robotic arm from Figure 10 and made an edit with some joints fixed to force the system to only be able to find two solutions. In this case the edit is not under-determined but there are two distinct solutions to the inverse problem and our system is able to find them during the global optimization phase. The solution on the left-hand side is the nearest solution i.e.,the solution that changes the least the angles. The solution on the right-hand side is the farthest solution i.e.,the solution with the biggest changes in angles. Note that the change that makes the right-hand side solution farther from the initial configuration is very subtle: the first joint (on the bottom) is rotated by 180 degrees.*

	Stool	Rob. arm	Cubes	Succ1	Succ2
Local opt.n only	0	6	2	5	6
Nearest		4	2		
Farthest	3	3	3	0	3
Delta-like change	2	1	0	0	5
Prop. change	14	0	0	11	4
Least change X1	2	7	1	5	3
Least change X2	N/A	N/A	13	N/A	N/A
Chi-squared value	29.71	10.71	35.29	23.76	5.05
Chi-squared sign.	0.00	0.06	0.00	0.00	0.28

Table 2: ***Quantitative user study results:*** *For each procedural model: Stool, Robotic arm, Cubes and two succulents, users where asked to pick their preferred suggested solutions after an under-determined edit. Each cell shows the number of times a user preferred one solution over the others. The Chi-squared significance is the probability that the user preferences are following a uniform random distribution.*

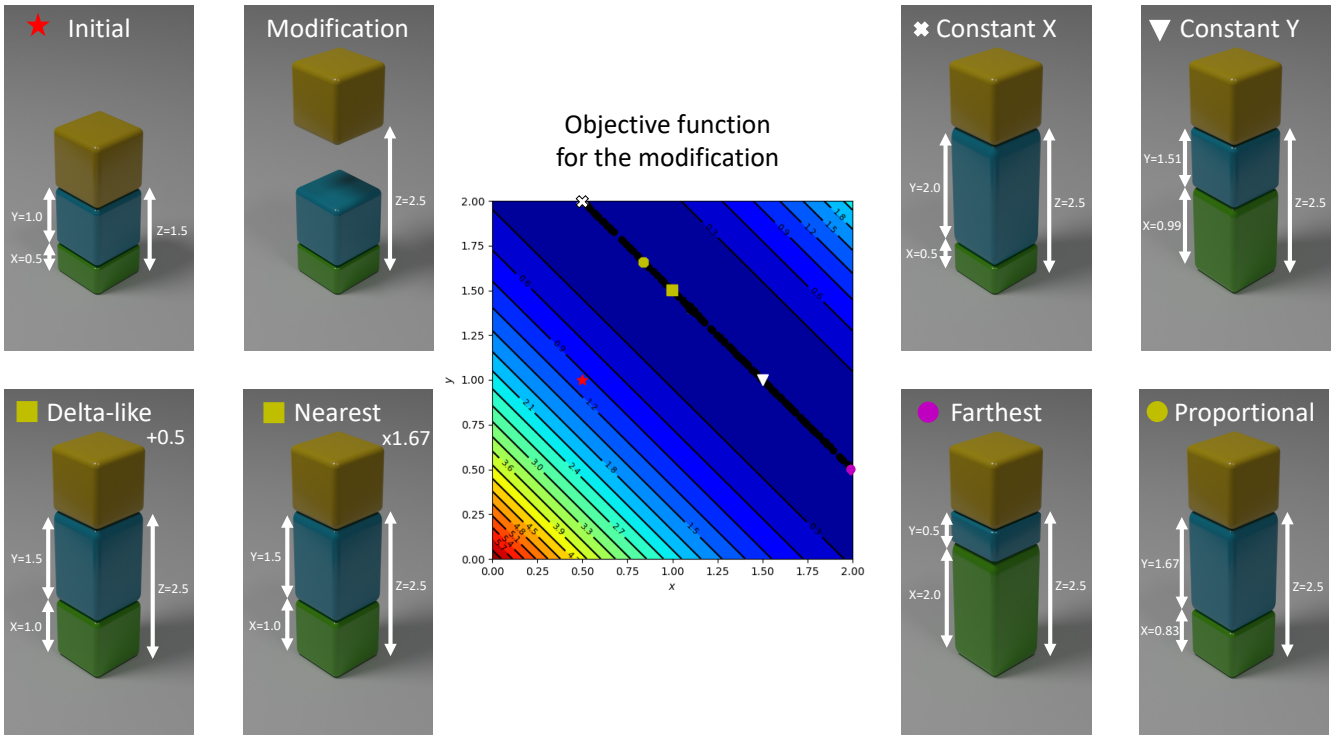


Figure 13: ***Objective function for the modification of a cube stack with recommended models:*** *The cube stack is a typical example of an under-determined system. The bottom cube (green), the middle cube (blue) and the top cube (yellow) are stacked on top of each other. The height of the bottom cube is controlled by the variable X, the height of the middle cube is controlled by the variable Y and the altitude of the top cube is $Z = X + Y$. If we want to raise the altitude of the top cube, the system is under-determined and there are many ways to propagate a change in Z back to X and Y. Our system finds many optimal solutions and present some recommendations to the user. The* ***Initial*** *image shows the initial configuration with $X = 0.5$ and $Y = 1.0$, thus $Z = 1.5$. The* ***Modification*** *image shows the modification asked by the user: Z becomes $Z = 2.5$. In the middle, we present the contour plot of the objective function induced by the modification made by the user. There are two variables to the problem X and Y, which are both bounded between 0.0 and 2.0. The initial configuration is indicated by the red star ($X = 0.5$ and $Y = 1.0$) on the contour plot. Per the modification asked by the user, we want to find X and Y so that $Z = X + Y = 2.5$. To achieve this modification, we minimize the following loss function $f(X,Y) = (X + Y - 2.5)^2$. We can see on the contour plot that the region of optimality is a line segment from (0.5, 2.0) to (2.0, 0.5) materialized by the set of black dots. Images of some of the optimal configurations can be seen in Figure 14. Relevant configurations found by our optimizer are shown on the contour plot with symbols, and the corresponding models are shown on the sides of the figure. The* ***Delta-like*** *image (symbolized by the yellow square) shows the solution that adds a constant value ($+0.5$) to both X and Y. The* ***Nearest*** *image (symbolized by the yellow square) shows the solution that is the closest to the initial in L2 norm. Note that for this particular problem, the delta-like configuration happens to be the same as the nearest configuration, but it is not always the case. The* ***Constant X*** *image (symbolized by the white cross) shows the solution that best keep X constant. The* ***Constant Y*** *image (symbolized by the white triangle) shows the solution that best keep Y constant. The* ***Farthest*** *image (symbolized by the magenta dot) shows the solution that is the farthest from the initial in L2 norm. The* ***Proportional*** *image (symbolized by the yellow dot) shows the solution that multiply both X and Y by a constant factor ($\times 1.67$).*

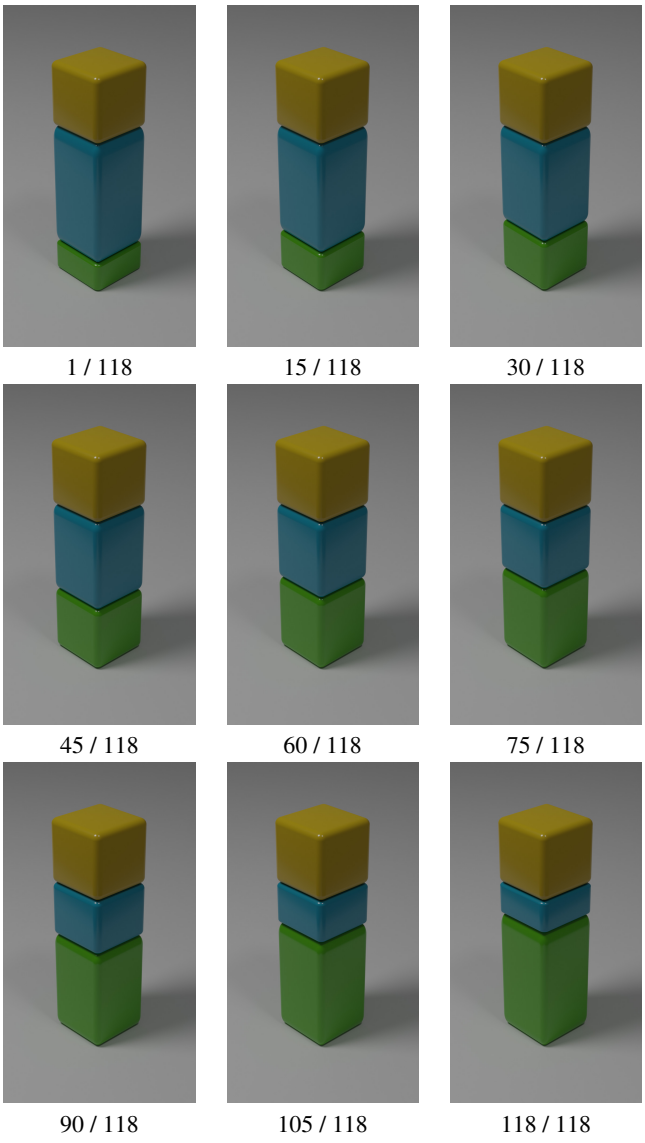
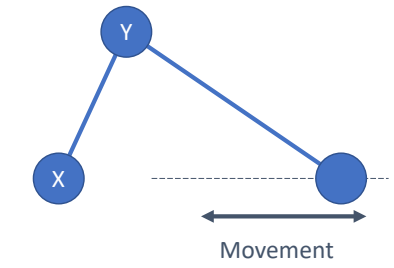
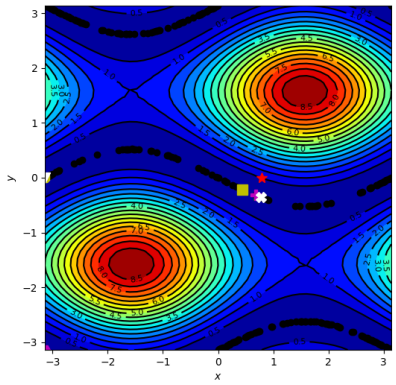


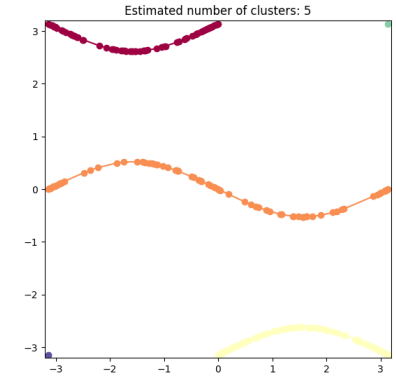
Figure 14: ***Cube stack ordering:*** *We show samples of the ordered optimal points. See Figure 13 for the description of the model and the modification that lead to this set of optimal solutions. A total of 118 non-duplicate optimal solution were found. All of them were grouped into one group and ordered using the heuristic detailed in Section 6.8. From the upper-left corner to the bottom-right corner of this figure, we present 9 samples from the 118. We can see that the order is going from small to large values of X (height of the bottom cube).*



2D robotic arm



Objective function



Optimal solutions

Figure 15: ***Optimization of a 2D robotic arm:*** *Example of a 2D robotic arm with two degrees of freedom. The end effector is edited to be sliding on the horizontal line, which defines the following objective function:* $f(x,y) = (\sin(x) + 2\sin(y))^2$. *Our optimizer explores the optimality regions and finds different solutions to the problem. For reference on the meaning of symbols in the objective function plot, see the caption of Figure 13. These solutions are then clustered into 5 regions of optimality and ordered as show in the bottom sub figure. For more details on optimization and ordering, see Figure 18.*

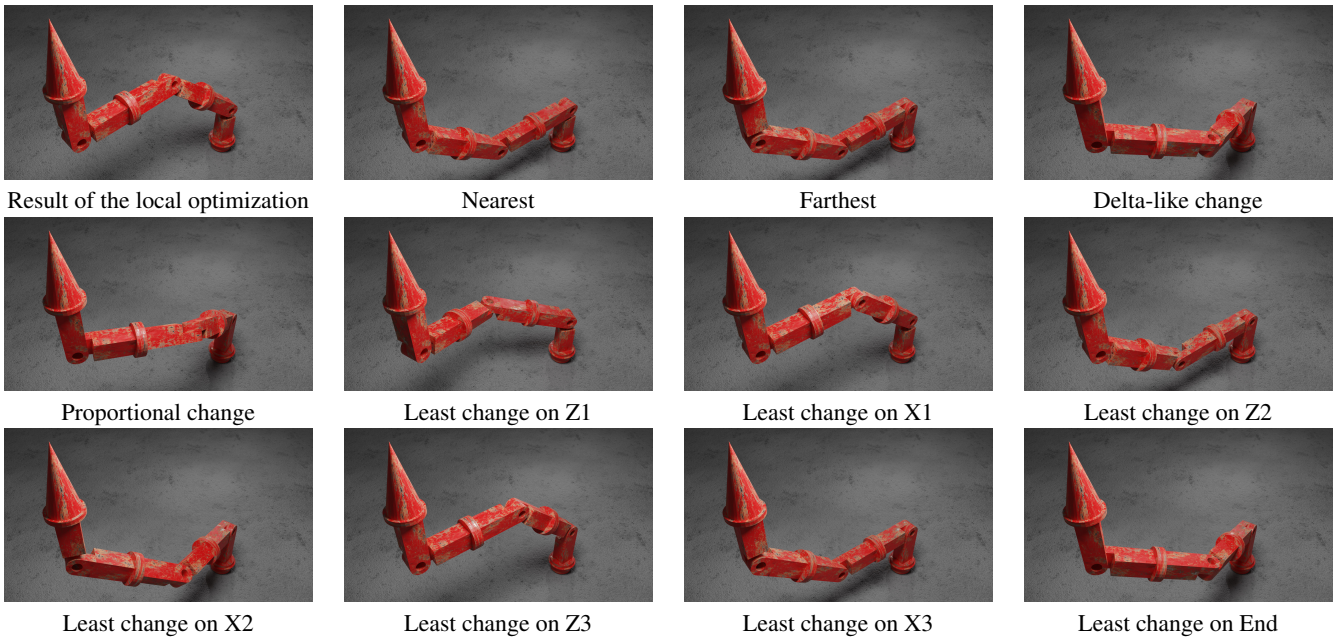


Figure 16: ***Robotic arm recommendations:*** *This figure shows suggestions made by our system for an under-determined edit of the robotic arm. The initial configuration (not shown here) is a straight robotic arm going up, with all angles set to 0 degrees. The* ***Result of the local optimization*** *is the result after the first local optimization phase. The* ***Nearest*** *solution is the most similar to a straight robotic arm, in other words it is the solution that changes angles the least. The* ***Farthest*** *solution is the most different solution, in this case it looks similar to the Nearest solution, except that some joints are flipped 180 degrees. The* ***Delta-like change*** *solution is the solution that adds a constant value to all angles. The* ***Proportional change*** *solution is the solution that applies a constant multiplication factor to all angles. All other solutions are the ones with the least change on a certain joint in the robotic arm: Z1 is the first joint, X1 is the second joint, Z2 is the third joint, X2 is the fourth joint, Z3 is the fifth joint, X3 is the sixth joint, End is the last joint. See the supplementary video for an animation of the optimal solutions of the robotic arm for this edit.*

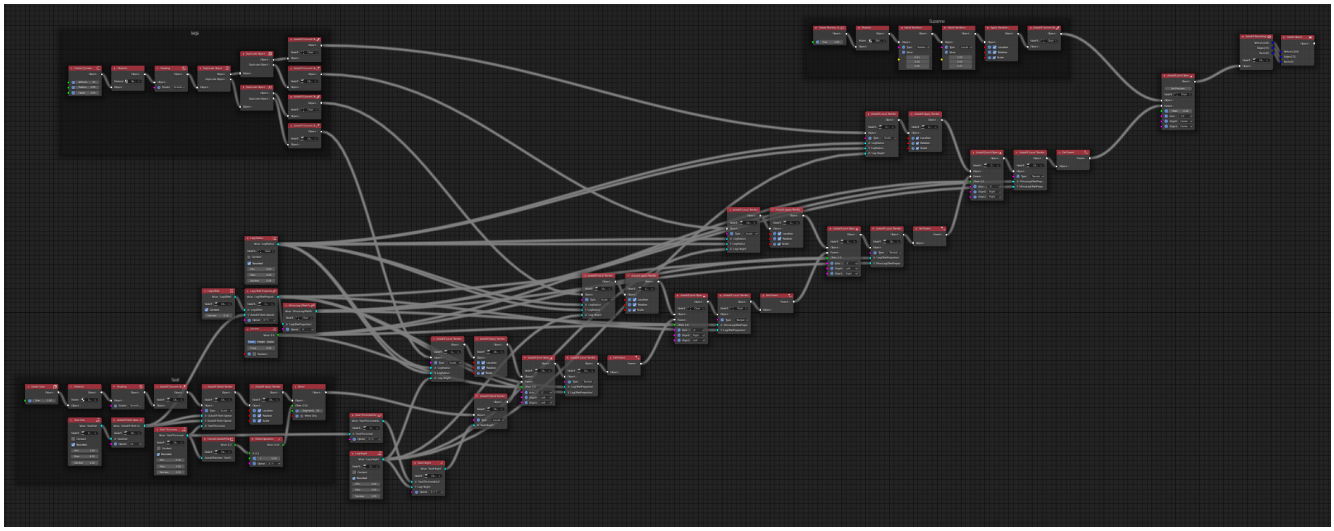


Figure 17: *Blender® procedural graph for the Stool shown in Figure 5. We recommend using our add-on to visualize graphs directly in Blender®.*

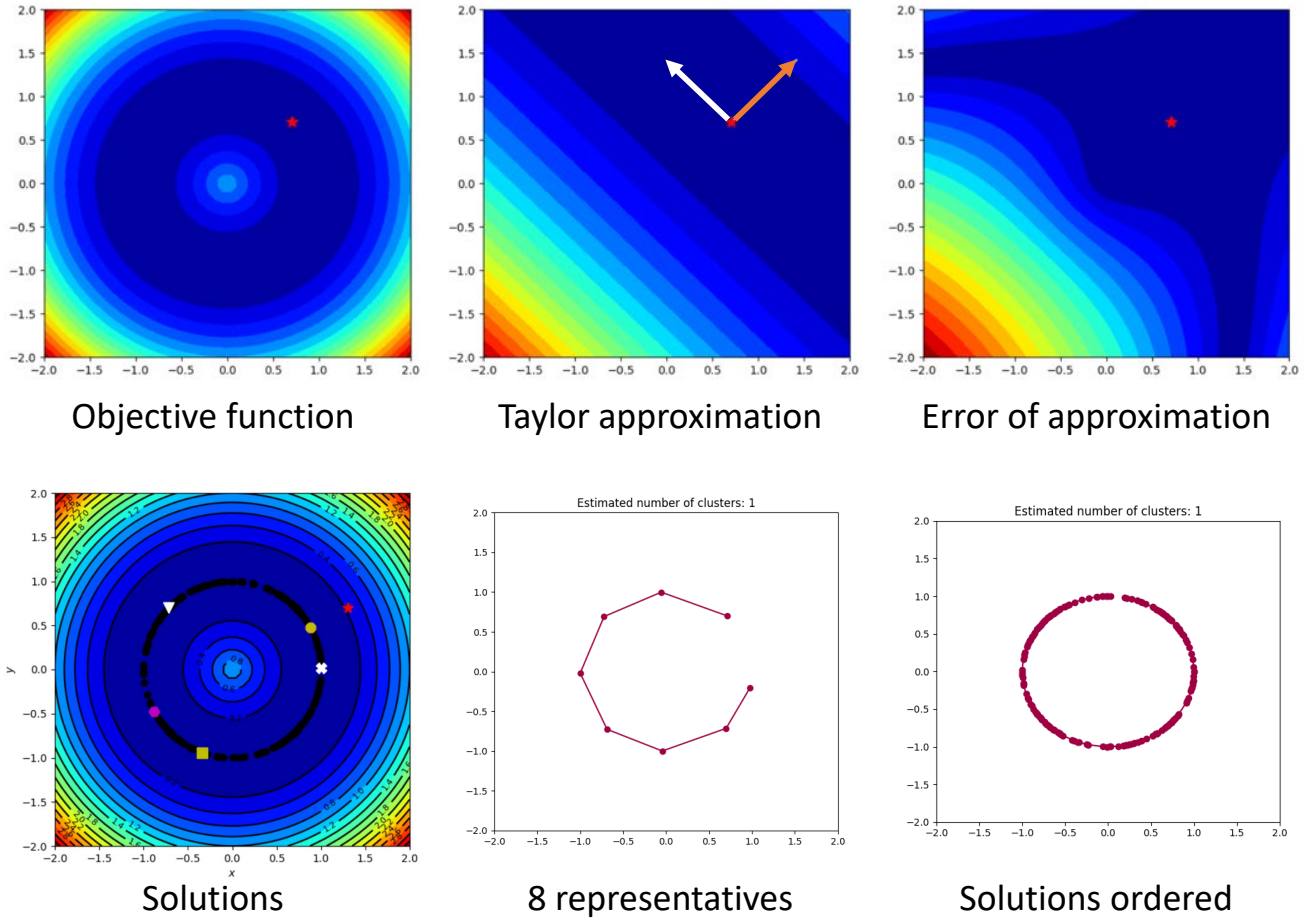


Figure 18: ***Optimization:*** *This Figure shows how our optimizer works on an objective function with 2 variables. Each sub figure shows one aspect of the optimization. The* ***Objective function*** *plot shows a contour plot of the example objective function:* $f(x) = (\|x\| - 1)^2$. f *is of particular interest because its region of optimality is the unit circle. The* ***Taylor approximation*** *plot shows the second order Taylor approximation of the objective function at the point* $p = (\sqrt{2}, \sqrt{2})$ *(red star in the plots), which is a local minimizer. The two arrows show the eigenvectors of the Hessian matrix at point* p*. The white eigenvector has an eigenvalue of* 0.0*, and the orange eigenvector has a an eigenvalue of* 2.0*. The white eigenvector is tangent to the region of optimality, thus going in this direction rather than in the direction of the orange vector ensures that we stay close to the optimality region. The* ***Error of approximation*** *plot is the absolute difference between the Taylor approximation in* p *and the value of* f*. It shows the region in which we can trust the second-order Taylor approximation. The* ***Solutions*** *plot shows optimal solutions found during the exploration of the optimality region. For reference on the meaning of symbols, see the caption of Figure 13. The* ***8 representatives*** *plot shows the 8 K-Medoids of the optimal solutions, and the shortest Hamiltonian path going through them. The* ***Solutions ordered*** *shows all optimal solutions in the order presented to the user.*