TECHNISCHE UNIVERSITÄT DRESDEN

Computer Science, Institut für Technische Informatik, Chair of Compiler Construction

Automatic compiler-based differentiation of distance functions

Tendsin Mende

Matriculation number: 4680252

Master Thesis

to achieve the academic degree

Master of Science

First referee

Prof. Dr. Jeronimo Castrillon

Second referee

Prof. Dr. rer. nat. Stefan Gumhold

Supervisor

Karl F. A. Friebel

Submitted on: 29th November 2024

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Automatic compiler-based differentiation of distance functions* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Tendsin Mende

Signed by: Tendsin Mende

Date: 2024-11-29 15:52:57 GMT+1

Dresden, 29.11.2024
Ort, Datum

Unterschrift

Abstract

Domain specific compilers can be leveraged to optimize implicit distance functions, like signed distance functions. Widely employed automatic differentiation techniques, combined with domain specific knowledge can increase the fitness of such compilers for an even wider array of tasks. Based on a broad survey of automatic differentiation techniques in different scientific domains, as well as a review of existing methods, a distance-function domain specific approach on the compiler level is realised. It is then compared against other existing methods as well as tested for fitness in different applications.

Zusammenfassung

Domänenspezifische Compiler können zur Optimierung impliziter Distanzfunktionen eingesetzt werden. Weit verbreitete automatische Differenzierungstechniken, kombiniert mit domänenspezifischem Wissen, können die Eignung solcher Compiler für ein noch breiteres Spektrum von Aufgaben erhöhen. Auf der Grundlage einer umfassenden Übersicht über automatische Differenzierungstechniken in verschiedenen wissenschaftlichen Bereichen sowie einer Überprüfung bestehender Methoden wird ein domänenspezifischer Ansatz für Distanzfunktionen auf Compilerebene realisiert. Dieser wird mit anderen existierenden Methoden verglichen und auf seine Tauglichkeit in verschiedenen Anwendungen getestet.

Contents

1. Introduction

Automatic differentiation (AD) is a technique with roots that stretch back several decades, playing a crucial role in fields such as optimization, machine learning (ML), and scientific computing. It is often essential for scientific areas that require efficient computation of derivatives. In this thesis, we provide a comprehensive survey of the current state of the art in AD, focusing on both the theoretical underpinnings and the practical techniques that have emerged over the years.

Alongside the AD techniques themselves, we explore applications with relations to distance function (DF) that use, or would profit from readily accessible derivatives. We explore recent advantages in image rendering, modeling, and semantic model reconstruction that fit this description.

To bridge the gap between the AD techniques and their application areas, we highlight the connection between the two fields and how a domain specific compiler fills this gap. We discuss how already existing development in AD can be tailored to meet the needs of our specific application and how a deeper understanding of the domain at derivative-creation time enables us to simplify the process itself. This synergy enables us to implement a comparatively simple algorithmic differentiation step in a distance function domain specific compiler.

Our approach leads to a compiler-based AD implementation that leverages high-level partially domain specific knowledge to statically compile derivatives that are fit for modern graphics processing unit (GPU) execution. We compare our approach to Enzyme, another in-compiler AD implementation. It is shown that we obtain comparable runtime performance at much shorter compile times for our small, domain specific cases. We also give several examples for improvements to artistic workflows and rendering algorithms.

1.1. Motivation

Distance functions provide a concise, resolution-independent way to mathematically describe matter, capturing spatial relationships with elegance and precision. Using a domain specific language (DSL) compiler, we can keep those properties all the way to their actual evaluation as part of some algorithm.

Derivatives, a fundamental mathematical tool, align naturally with distance functions by allowing us to explore gradients in those fields. A logical next step after creating a distance function DSL is to extend it with support for derivatives, combining both concepts to create a more expressive and versatile tool to represent and use distance function in computing.

1.2. Distance functions

DFs in this thesis are all functions that lie in some kind of metric space. *Metric space* meaning any space, with a notion of distance. An example would be the 3-dimensional Euclidean space, where we can distinguish locations by measuring the distance with a ruler for instance.

Distance functions encode this simple fact. In 3D, we can calculate the distance by plugging in the point in space, from which we want to know the distance to whatever is encoded in the function.

A popular enhancement to that idea is the signed distance function (SDF). Additionally, to the distance, it also encodes if the evaluated point lies within something (via a negative sign), or outside of something, via a positive sign.

A mathematical formulation of SDFs is to define a distance of 0 as the boundary or surface:

$$f(p) = \begin{cases} > 0 & outside \\ < 0 & inside \\ = 0 & on\ boundary \end{cases}$$

Picturing the 2D plot of a SDF helps the intuition. A circle can be defined as:

$$f_{sphere}(p) = ||p|| - radius$$

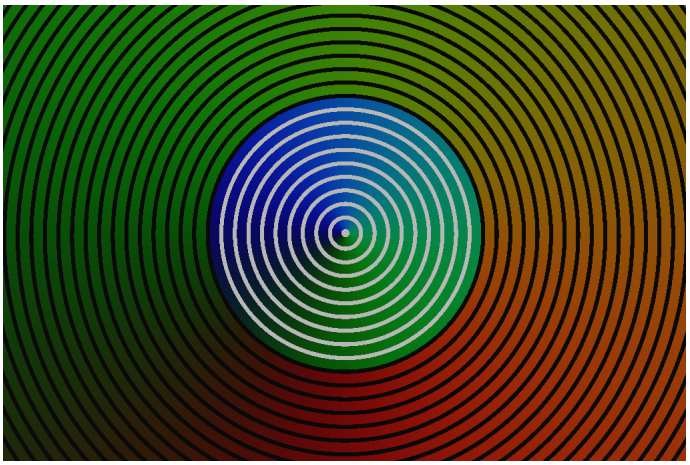
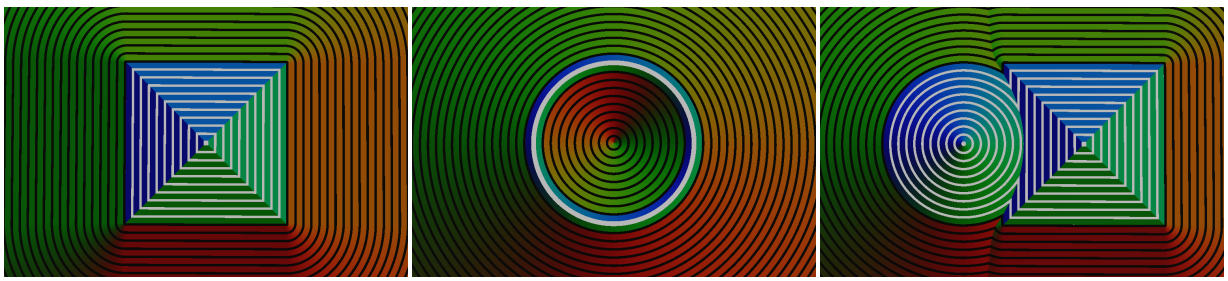


Figure 1.1.: Signed distance function f_{sphere} around $(0, 0)$.

Plotted around the origin of a 2D coordinate system the function can be seen in fig. 1.1.



(a) Box (b) Ring (c) Union box-sphere

Figure 1.2.: Selected SDFs

Similarly, other primitives like boxes, rings, lines, and even splines can be constructed (fig. 1.2).

Combining primitive shapes is comparatively simple for DFs and SDFs. Throughout the years, multiple operations and transformations were established in the SDF community [73, 66, 56].

Operations that are sometimes harder to compute in other volume representations, like cutting exact holes into a volume, are comparatively simple to represent in SDFs/DFs.

To give an example, consider merging two primitives:

A simple *union* operator might take two signed distances $f_a(p)$ and $f_b(p)$ and choose the smaller of the two distance (see eq. (1.1)):

$$f_{union}(p) = min(f_a(p), f_b(p)) \tag{1.1}$$

Visualizing this in fig. 1.2c one can see that the boundary surface now lies on the union of both primitives.

SDFs are not constrained to those simple operations. *Higher level* operation like smooth set operations, edge-avoiding set operations, twisting, mirroring etc. exist.

It is to note, that not all of those operations respect all properties of a scalar field, like continuity, differentiability etc.

1.3. Vola: The volume language

The AD of distance fields is implemented in the Vola's compiler stack. Vola is an experimental volume description language by the author, that focuses on GPUs as a compilation target.

1.3.1. Introduction to the language

Vola's language allows a user to define trees of operations, similar to constructive solid geometry (CSG) trees. fig. 1.3 shows a simple model expressed as a subtraction of the left arm and the right arm. The left arm forms the rounded box, while the right arm constructs a cross of cylinders that are then subtracted from the rounded box in the top subtraction node. The full Vola source code for the model can be found in fig. A.1.

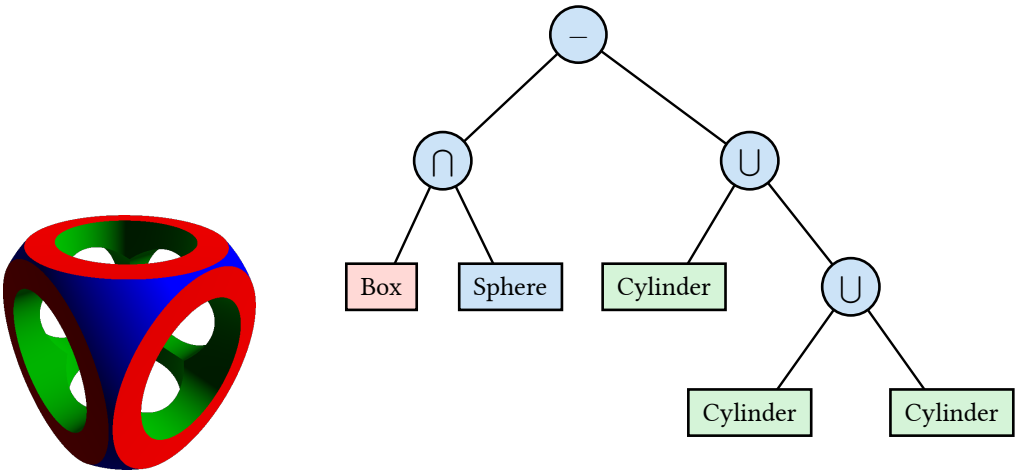


Figure 1.3.: CSG-tree of a simple model.

At some point, the DSL has to communicate to the *real world*. We do this by defining *exports* of our field with an explicit signature. Taking the example from fig. 1.3, we export the calculation of the signed distance field in fig. 1.4.

```
export eval_sdf(position: vec3){
    csg myfield = my_model();
    eval myfield.Sdf3d(position)
}
```

Figure 1.4.: Exporting a distance function from Vola.

Notice the `eval myfield.Sdf3d(position)` expression, which hints a distinct feature of Vola compared to other CSG languages. It allows you to evaluate a CSG tree under different *concepts*. For instance, to access the color of your CSG at some position in space, one can evaluate a *color concept* via `eval myfield.Color3d(position)`.

Concepts are defined as transformations of one data type into another (cf. fig. 1.5).

$$Sdf3d : \mathbb{R}^3 \to \mathbb{R}$$

(a) Mathematical notation

```
concept Sdf3d: vec3 -> s;
```

(b) Notation in Vola

Figure 1.5.: Mathematical and Vola's notation

The question remains how one defines the color, or generally behavior of operations in the CSG. This introduces the last building block: In Vola operations and entities can be defined from *within* the language. For instance, the `Union` operator described in eq. (1.1) is implemented in fig. 1.6 for a concept `Sdf3d`.

```
operation Union();

impl Union<l, r> for Sdf3d(at) {
    let left = eval l.Sdf3d(at);
    let right = eval r.Sdf3d(at);
    min(left, right)
}
```

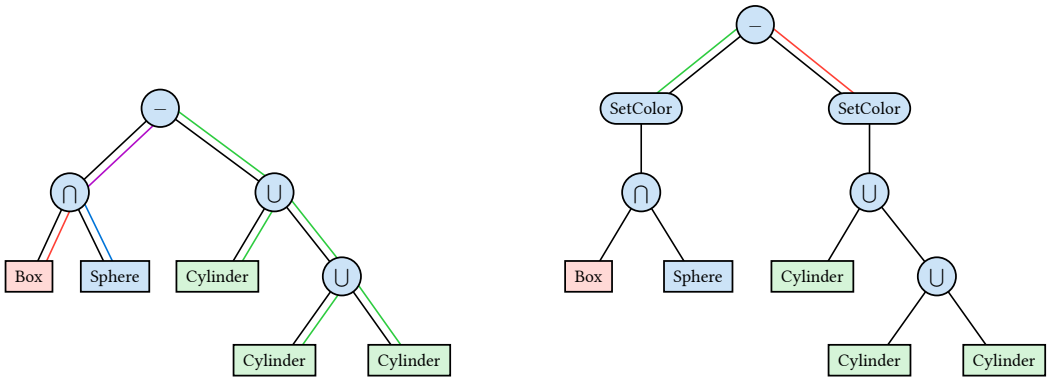
Figure 1.6.: Implementation of min-union from eq. (1.1) in Vola

Returning to the simple model, we can implement the *Color* concept for all nodes, which gives us the CSG tree in fig. 1.7a.

Note that not all nodes need to implement all concepts. Only nodes that are in use, under a concept's interpretation of a tree need to implement the concept. To elaborate: one can define an operation `SetColor`. A simple operation that just returns a color. This effectively renders all children to such a node invisible, when the CSG tree is evaluated for `Color`. Illustrated in fig. 1.7b the tree remains the same size for the black `Sdf` interpretation of the tree, but shortens considerably under the `Color` interpretation.

The compiler makes sure, that the tree is valid under a given interpretation at compile time.

Practically speaking this makes sure, that the concept, that is being evaluated is implemented everywhere in the tree. So for instance, if the tree uses an operation `Foo`, and is evaluated for a concept `Bar`, the compiler makes sure, that `Foo` is implemented for `Bar`.



(a) CSG tree with the color concept implemented for all nodes.

(b) CSG tree with the SetColor node shortening the tree for the Color concept.

Figure 1.7.: Using CSG-operand to shorten CSG-tree under interpretation.

1.3.2. Compiler technology

The frontend is the DSL, shown in section 1.3.1, that is parsed into a high-level abstract syntax tree (AST). The AST is kept simple in order to be able to use other languages as a frontend, like OpenSCAD [63]. This also allows building the AST not just from source code, but possibly a graphical interface like a node-graph as well.

Vola's optimizer is based on a graph intermediate representation (IR) called Regionalized Value State Dependency Graph (RVSDG) [96]. Internally, the compiler employs the multilevel / dialect idea central to newer compiler frameworks like MLIR [69] or xDSL [10].

The output of Vola's backend is SPIR-V [54], an intermediate representation for GPU code used by OpenCL and Vulkan. Multiple translators to other proprietary formats like DirectX's DXIL, or Metal-IR for the Metal graphics API make this a suitable format for cross-platform, cross-vendor applications.

1.3.3. Regionalized Value State Dependency Graph introduction

The implementation is described in the IR terms of the compiler which is an unmodified version of a RVSDG described in [96]. We introduce the concepts informally to allow the reader to build an intuition, when later referring to implementation details.

The task of the IR is, to describe a program wholistically, that makes analyzing and transforming it possible. The RVSDG belongs to the graph-based IRs[1] family. There are three fundamental parts to the graph: Nodes, edges, and regions.

A node describes an *action*, for instance an addition, accessing a value in memory etc.

An edge describes a dependency between those nodes. There are two types of edges, value edges, and state edges. An addition might be connected to the two values it adds via value edges. A memory-read might be connected to a memory-write via state edge, to signal that the memory-write has to be executed before the memory-read.

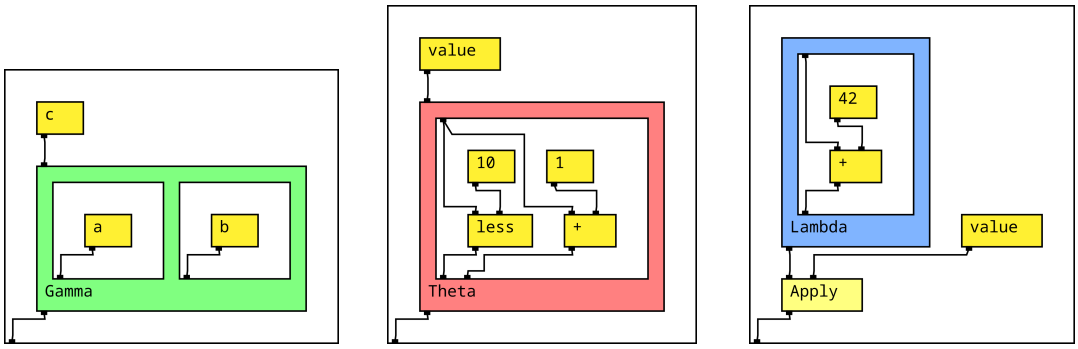
At this point we can already describe small programs by connecting nodes via state or value edges. We can, however currently not make decisions, loops, or call functions. The remaining part are inter- and intra-procedural nodes. This is also where the regions come into play. Inter-procedural means that the control-flow won't leave the current *location*, while intra-procedural nodes might move to another region entirely.

[1] In this case a directed acyclic graph (DAG)

There are two types of intra-procedural nodes, Gamma and Theta nodes. Gamma nodes can be understood as n-way *if-then-else* construct. It is a node with n-inputs, where the first input is an integer describing which sub-region is taken (cf. fig. 1.8a), the remaining inputs are fed to the respective sub-regions of each branch as *entry-variables*. All branches produce the same amount of outputs, called *exit-variables*. The Theta node describes a loop that repeats execution until the first result, the theta-predicate, signals finishing of the loop (cf. fig. 1.8b). Any input to that Theta node is also an output, called a *loop-variable*. Whenever the loop is repeated, each result of the loop-body is mapped back to the input, if the loop finishes, the result is instead mapped to the Theta node's output.

Inter procedural nodes describe whenever control-flow leaves, or moves out of its current procedure, i.e., function calls, recursion or similar. For our implementation we only need two constructs, a function-call, called *Apply node*, and a function, called a *Lambda node*. Apply nodes take a value dependency that is connected to a Lambda node as their first input, and the Lambda's arguments as the remaining inputs. The Lambda node contains a single sub-region, the function-body.

There are more constructs, and details to the IR, which we don't mention in this short introduction, please refer to [96] for more information. We are however now able to describe simple programs with control-flow and non-recursive function-calls.



(a) Deciding if a value is a or b, depending on c

(b) Looping until the value becomes 10

(c) Calling a function that adds 42 to an argument

Figure 1.8.: Usage of introduced RVSDG nodes

2. Background and related work

2.1. Introduction

In the first part, we show that there are many use cases that profit from differentiation of distance functions. There are machine learning like search optimizations, enhancements to distance functions as well as more engineering focused applications like CSG reconstruction from unordered data.

Turning to AD specifics, the project profits from a long history of research into that matter. First AD systems were described in the 1950s - 1960s. Since that time, AD found its application in many scientific and engineering contexts. This makes it possible to find already tested techniques for specific problems.

Previous research has already addressed whether and how post-optimization AD should be applied to multi-level compiler toolchains. By combining this with the compiler-like implicit modeling kernels described in section 2.2.2, we can make informed decisions about implementing differentiable signed distance functions in Vola's compiler.

2.2. Distance field implementations

Before focusing on the use-cases of AD, we review similar projects to Vola. While the amount of distance function domain specific languages is small, we can find two distinct techniques. One embraces the algebraic nature of the topic and tries to create machine executable code of some form. The other focuses on the creation of volumes, or 3D models, which is already the interpretation of that code into some representation. One could argue that the former approach is leaning more into the implicit aspect of distance functions than the other.

2.2.1. Code targeting domain specific languages

The closest SDF compiler to Vola [85] is Raumkünstler [64]. It is a LLVM based just-in-time-compiler (*JIT*) that transforms a visual node graph into native code. Contrary to Vola that code is used to calculate a traditional triangle based mesh. Vola in contrast outputs GPU native code, that can be used to evaluate the function directly.

Bauble [45] is an embedded DSL within the Janet [16] language. Similarly to Vola, and contrary to Raumkünstler a program is represented by a string. Contrary to both, Vola and Raumkünstler the target is not executable, platform dependent code, but high level GLSL [55] shader code. This compilation target makes it similar to other experimental projects, which compile to high level shader languages.

Notably, none of those languages have the ability to differentiate or otherwise analyze the generated distance function.

3D model targeting DSL

A family of DSLs that do not compile to executable code, but employ compiler related optimizations is the OpenSCAD family of DSLs, as well as related projects. OpenSCAD describes itself as something like a 3D-compiler that reads in a script file that describes the object and renders the 3D model from this script file [63]. It could be seen as a domain-specific compiler that focuses on generating 3D object files. A notable feature is that OpenSCAD supports integrating traditional polygonal 3D models into its workflow.

ImplicitCAD [75] has a similar approach, but develops out of a functional-language concept. Compared to OpenSCAD it only works with implicit functions.

2.2.2. Distance field modeling kernels

A related field is DF based modelling kernel. *Modelling kernel* being the terminus used in the computer aided design (CAD) application space to describe the core components that transforms the high level *modelling operations* into the final model. Specifically, a niche family of kernels exists, that internally functions similar to the DSL compilers mentioned before.

nTop [49] is a full-fledged CAD application based on such a compiler-like modelling kernel. From version 5 it has its own proprietary implicit modeling kernel [50]. The software is interesting because of its sophisticated use of implicit modeling, as well as fielding some novel engineering focused applications like field-driven-design and unit gradient field (UGF) [17] in practice.

Before switching to its proprietary implicit modeling kernel, nTop was based on LibFive. First developed by Matt Keeter [58] it traces its lineage back to Antimony, which was developed at MIT's Center for Bits and Atoms [59].

The latest development of Matt Keeter et al. is Fidget [81] which itself is based on an experimental strategy for implicit surface rendering using interval arithmetic and runtime code refinement called MPR [60]. All kernels of this lineage share an optimization approach that is similar to traditional compilers. Distance functions are combined on an algebraic level. The resulting code is optimized for size, for instance via dead-code-elimination, or for execution speed via common-subexpression-elimination. MPR and Fidget additionally focus on exploiting the massive parallelism of GPUs, by optimizing the code-tape via interval arithmetic.

Both MPR and Fidget have the ability to calculate the first derivative via forward AD.

2.3. Applications of automatic differentiation in the distance field domain

To get a better understanding of what a distance field compiler needs to be able to differentiate, we explore the applications of that domain. The following represents a non-exhaustive list of cases, in which differentiation of distance fields is needed, or useful.

It ranges from improvements to the distance functions itself, improvements to the evaluation of distance function based rendering and collision detection in general, to specific professional applications in the CAD space and machine learning.

2.3.1. Unit gradient fields

An interesting field definition related to SDFs, which are common, are UGFs. Intuitively, they can be understood as a distance field, but the gradient of the field is 1 everywhere. A better definition can be found on Blake Courter's blog [17] as well as a more formal definition in Luo et al.'s Analysis on SDF [76].

Courter explains why the unit-gradient property is important when using DFs in the CAD sector [18]. AD comes into play when trying to convert a common signed-distance-field to a unit-gradient-field. Paul D Sampson argues that an approximation of a unit gradient field can be found for any distance field [99], simply by dividing the distance by the magnitude of the first order derivative at the same point[1].

In conclusion AD of a signed distance field[2] would allow us to turn that field into a UGF, fit for CAD application. A user of the system thus can use the well explored field of SDFs, but can still benefit from the advantages of UGFs.

2.3.2. Differential engineering and design space exploration

Differential engineering extends the concept of parameterized engineering by enabling optimization of sufficiently parameterized models using gradient descent instead of brute-force design-space exploration. This approach requires access to the gradient, which, for distance functions, corresponds to the partial derivative with respect to the space parameter(s).

The advantages of such processes have been researched in aerospace engineering [19, 35] but also other parameterized computer aided design [11]. While the cited works do not just consider modelling characteristics, but a wealth of other physical properties, it is safe to say that AD is needed for automated optimization of engineering efforts.

An interesting detail about differentiable 3D CAD programs is the creation of a parameter DAG [11], that is subsequently used for AD. On the surface this looks similar to systems like Fidget [81] or Libfive [58]. However, the system operates on the explicit, vertex-based representation of the model rather than its implicit representation.

2.3.3. Distance field (re)construction

Two main avenues for distance field reconstruction from other data sources exist. Recognition-based strategies, that try to recognize patterns in attributes of input data and construct a CSG-trees from those. These techniques are often ML based. On the other hand search-based techniques try to fully explain the reconstructed input in terms of CSG operations, which typically makes them slower [24].

When focusing on the application of ML in the context of DF reconstruction, promising results for creating distance fields from boundary-representations [40], point-cloud and voxel-grid to CSG-Tree construction [113] can be found. It is also possible to convert tessellated meshes to distance fields [100] and reconstruct them from images [115] as well.

All the ML based approaches benefit from, or require differentiation for a better informed back-propagation step while training. Some mentioned techniques even need to be able to differentiate not just the model boundary property, but other attributes like color, or shading properties too.

On the *search-based* reconstruction side, we see advancements in image to model reconstruction [110], including the color characteristics of a model. Additionally, reconstruction from point-cloud [24], or via iterative refinement of CSG-Trees [73] are explored. Friedrich et al. [31] show that techniques can be mixed in multistage hybrid reconstruction.

[1] Formula 3 in [99]

[2] It still must be differentiable and have non-vanishing gradients, see [99, 17]

For an overview of reconstruction approaches, see *A Survey of Methods for Converting Unstructured Data to CSG Models* [28].

This shows that differentiation of distance functions and other encoded data in distance fields are beneficial when reconstructing distance fields or CSG-Trees from other data sources.

2.3.4. Implicit shape rendering

In computer graphics, a popular collision detection and rendering algorithm for signed distance fields is a method called sphere-tracing [43]. This method suffers from bad scaling when encountering ray-parallel surfaces, and near-misses. Several improvements to the original algorithm exist [102] [61] [5] that try to fix this problem via over-stepping, heuristics and other approaches. An interesting recent development is the inclusion of local information, called *local Lipschitz bound*[3] of the function. This allows [32] to formulate a robust overstepping of the *global* Lipschitz bound, which is synonym to the signed distance function in that case. Obtaining that local Lipschitz bound, however, is not as simple as just relying on the global Lipschitz bound, which might attribute to the comparatively low usage of this algorithm.

A further development of this technique approximates (among other approaches) the local Lipschitz bound as the local bounded derivative of the global Lipschitz bound [4] (Cf. *5.1 Linear Taylor Inclusion Functions* in [4]). The advantage over the former technique is the simpler acquisition of a first derivative compared to a local Lipschitz bound.

AD would help users of the distance function compiler to implement recently developed advancements in implicit surface rendering.

2.4. Automatic differentiation survey

Differentiation as a fundamental tool of calculus is in use in almost any math related scientific field. The following survey will focus on *automatic differentiation* as a form of differentiation. While symbolic differentiation can be precise in its description of a derivative, in practice, when working with programming languages it becomes challenging to describe constructs like *loops* in a mathematical sense.

Numerical differentiation, that is often based on some form of finite-differences [7][4], can be implemented efficiently on computer generated code [79]. However, it suffers from imprecision and bad scaling characteristics for partial derivatives and higher-order derivatives [79, 7].

Automatic differentiation is a technique that originates in the 1950s to early 1960s [52] [7]. In contrast to symbolic methods, it always produces a value rather than an expression. It can also be applied to algorithms rather than just mathematical expressions. Compared with numeric differentiation, it does not exhibit the same imprecision and scaling characteristics.

The basis for AD forms the observation that all numerical computations are ultimately compositions of a finite set of elementary operations. If one knows the derivative of each operation, combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition [7].

It is to note, that symbolic, numeric, and automatic differentiation methods can be mixed [77]. When doing so, the characteristics of each method are introduced into the result which are compared in table 2.1.

[3] Lipschitz boundedness comes from *Lipschitz continuity*. Intuitively, this describes functions that are constraint in how much it can change in a given interval. Another formulation is, that the first derivative is bound for a given interval.

[4] Cf. 2.1 AD Is Not Numerical Differentiation in [7]

Name	Output	Pro	Contra
Symbolic	Expression	Exact	Hard/impossible for general purpose code
Numeric	Value	Simple to implement	Bad scaling, approximation, possibly high error range
AD	Value	Exact, Simple to implement	Needs careful implementation for good performance

Table 2.1.: Comparison of the techniques

2.4.1. On structuring this field

AD approaches and implementations can be structured differently:

A common method is by-mode, which is either forward- or backward-AD. Both describe a way of accumulating parts of the final result in different ways. A third technique are dual-numbers, which we will also categorize as a *mode*.

AD implementation greatly differ based on strategic implementation details and optimizations. We will cover recurring optimizations and classify them based on memory, compile-time and run-time optimizations.

Lastly, implementations can be structured based on the level of code transformation they are applied on. This means distinguishing domain specific implementation from general purpose implementations in general purpose languages, down to *post-optimization* implementations that work on some form of compiler related intermediate representation.

2.4.2. Basics of AD

When working with scalar or vector fields, which is our main goal, we have to understand the concept of the *Jacobi-Matrix*.

Given a SDF we observe, that the function is defined as:

$$f_{sdf} \ : \ \mathbb{R}^n \to \mathbb{R}$$

When differentiating the scalar field, we effectively create a vector field, of that scalar field's gradient:

$$\begin{aligned} &f : \mathbb{R}^3 \to \mathbb{R} \\ &grad\ f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}) = \nabla f \end{aligned} \tag{2.1}$$

But what happens, if we try to differentiate a vector field, for instance as a second derivative of a scalar field?

If we rewrite a vector field:

$$\begin{aligned} &\vec{v} \ : \ \mathbb{R}^3 \to \mathbb{R}^3 \\ &\vec{v}(\vec{x}) = (v_x(\vec{x}), v_y(\vec{x}), v_z(\vec{x})) \end{aligned} \tag{2.2}$$

$\vec{v}(\vec{x})$ is now understood as three scalar fields. By applying the directional derivative from section 2.4.2 we get three vectors of three partial derivatives each. By assembling the partial vectors as row vectors in a matrix, we obtain the *Jacobi-Matrix* $J_{\vec{v}}$:

$$J_{\vec{v}} = \begin{pmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{pmatrix}$$

Generalized to $f : \mathbb{R}^n \to \mathbb{R}^m$ we obtain the Jacobi-Matrix

$$J_f = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

2.4.3. By Mode

Forward mode

The basis of the forward AD approach forms the observation, that we can break down the calculation of any expression by applying the chain rule recursively, until the whole expression consists only of known derivatives of elementary operations. As an example, given the SDF of a sphere with constant $radius \in \mathbb{R}$ in eq. (2.3):

$$f_{sphere}(p) = ||p|| - radius \tag{2.3}$$

For $p \in \mathbb{R}^3$ an implementation might expand this function to only consist of elementary operations (eq. (2.4)):

$$f_{sphere}(p) = \sqrt{p_x^2 + p_y^2 + p_z^2} - radius \tag{2.4}$$

We can now build a computational graph for f_{sphere} of eq. (2.4) in fig. 2.1.

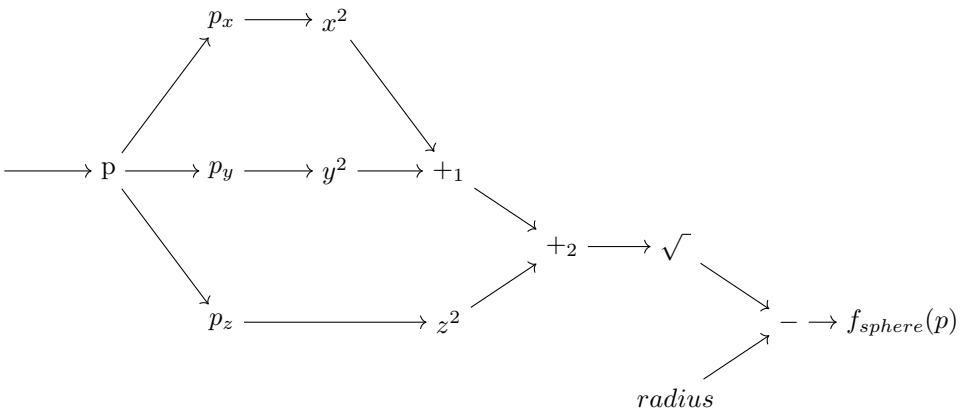


Figure 2.1.: f_{sphere} computational graph

This graph and the ability to introduce intermediate values, form the basis of AD. The advantage over symbolic differentiation is, that such a computational graph can be built for any value in a program, regardless of the taken control-flow during execution.

Forward (accumulation) mode is the naive application of the chain rule to each node in fig. 2.1. As an example: When calculating the partial derivative for x, we apply the derivative $\frac{\partial_f}{\partial_x}$ to the graph in fig. 2.2.

The computational graph of f_{sphere} is now transformed to include $\frac{\partial f_{sphere}(p)}{\partial_x}$. Note how the $\sqrt{}$ node is applied via the chain rule, since the inner part of the square root is an expression containing the variable x. Note that the figure does not include the expansion of g' for the constants x & y, which results in the subsequent reduction to $2 * x$ for the expression g'.

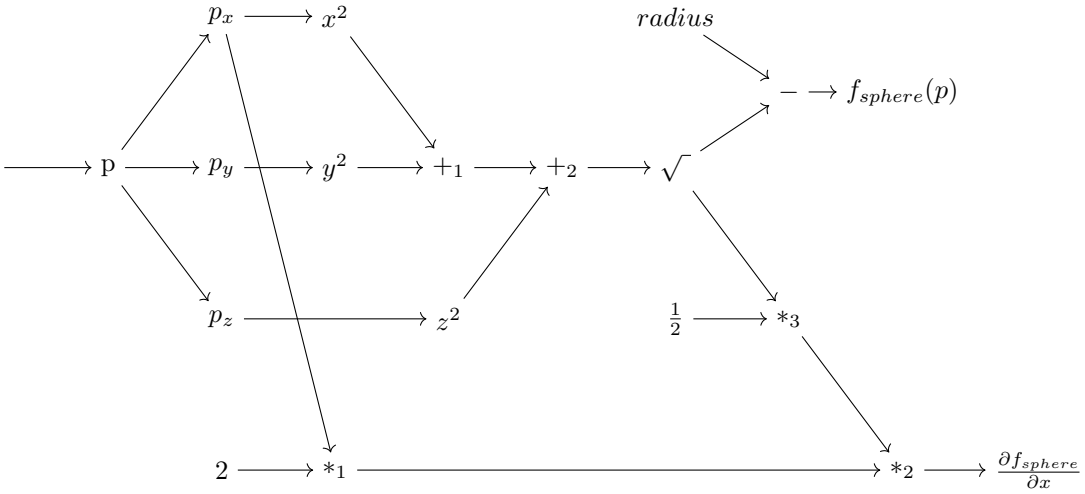


Figure 2.2.: Forward differentiation applied to f_{sphere}

$$\frac{\partial f_{sphere}}{\partial x} = \frac{1}{2} * \sqrt{x^2 + y^2 + z^2} * 2 * x = \frac{x}{\sqrt{x^2 + y^2 + z^2}} \tag{2.5}$$

The resulting graph then encodes eq. (2.5), which is the expected partial derivative for x.

We only had to know the derivation rules for x^2, $+$ and $\sqrt{}$ (which are our chosen *elementary* operations) to calculate the partial derivative for any parameter.

At run-time x, y and z are substituted to calculate f_{sphere} and $\frac{\partial_f}{\partial_x}$.

To get the full 3D gradient of the scalar field describe by f_{sphere} we can derive $\frac{\partial_f}{\partial_y}$ and $\frac{\partial_f}{\partial_z}$ the same way (Cf. eq. (2.2)). This way forward AD can be generalized over n for any scalar field $f : \mathbb{R}^n \in \mathbb{R}$.

The technique can be generalized to $f : \mathbb{R}^n \in \mathbb{R}^m$ while still needing only n evaluations, see [30] for a real world implementation in Matlab, as well as Elliott's Beautiful differentiation [25] paper and related blog post on generalized derivatives. The initial discovery is described by Khan et. al. [62].

Forward AD works best whenever the number of independent inputs is small, since the number of partial derivative evaluations scales with n. For the reverse case, backward-mode is more suitable.

A note on the signed distance domain For standard distance fields, which are usually scalar fields in 3D or 2D space, forward AD could be considered powerful enough. When introducing more information into the field, like color (typically 3D or 4D), physical parameters, or custom parameters, the number of independent input variables can increase, which makes forward AD increasingly unsuitable. In practice a heuristics based decision of the AD mode, or profiling can circumvent the problem.

Dual numbers

Dual numbers is an idea discovered in 1873 by William Clifford [14] to describe *biquaternions* in the context of rotation and parallel rotation traces. Later the concept was generalized, to

expressions of the form:

$$a + b\varepsilon \; where \; a, b \in \mathbb{R}, \varepsilon^2 = 0, \varepsilon \neq 0$$

Figure 2.3.: Definition of dual numbers

In the context of AD, the basic idea is, that for a function f in dual number space, the following *untangling* operation can be defined:

$$f(a + \varepsilon b) = f(a) + f'(a)b\varepsilon$$

Figure 2.4.: Dual number derivative split off

The already known chain-rule applies in the same way to dual numbers. We therefore can express the following transformation:

$$f'(g(a + b\varepsilon)) = f(g(a) + g'(a) * b\varepsilon) = f(g(a)) + f'(g(a)) * g'(a)b\varepsilon \tag{2.6}$$

Figure 2.5.: Applying first *untangling* rule followed by the chain rule.

When recursively applying eq. (2.6) to a function $f(x)$ we get both, the value $f(x)$ and $f'(x)$. This is in the same behavior as demonstrated by standard forward AD in fig. 2.2.

Rules used in dual-number based differentiation encode the same steps that are taken to transform the computational graph in forward AD [114]. This makes dual-number based AD a mathematical interpretation of the idea expressed in section 2.4.3.

The example above only considers one derivative. This approach can be extended by using hyper-dual-numbers for second order derivatives [29]. The disadvantage of forward AD, that the process needs to be done for each input variable, remains however.

Mature implementations exist in different programming languages like DNAD for Fortran [114], Julia [97], C++ [71] as well as a wealth of semi-professional implementations.

Backward mode

Both forward-mode, and the related dual-numbers based approach scale with the number of inputs. If the number of inputs is high, and the number of outputs is low, for instance for the back-propagation step of machine-learning applications or physical science, forward-mode becomes inefficient.

Backward accumulation is first described in 1970 [72] in the context of rounding errors in complex expressions [37].

The idea is similar to back-propagation, as that it tries to propagate the derivative *back* from a given output. This naturally requires, that the differentiation knows the output value, which is also the reason for the two-step nature of that algorithm.

The first step runs the computation in forward-mode[5] to calculate a result of a function $f(x) = y$ (including all intermediate values for each node in the computational graph, i.g. fig. 2.1).

The backward propagation idea originates, again, from the application of the chain-rule to the computational graph.

The central observation is, that for a function $f(g(x)) = y$ the following holds true for the derivative [47]:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial g}\frac{\partial g}{\partial x} \tag{2.7}$$

With that knowledge we can associate each node v_i in the computational graph with an adjoint $\overline{v}_i$:

[5] Sometimes called *primal-trace*.

$$\overline{v}_i = \frac{\partial y_j}{\partial v_i}$$

$\overline{v}_i$ represents the *change* of the output variable y_j when changing the value v_i[6].
By applying the rule from eq. (2.7) to each $\overline{v}_i$, we effectively need to calculate the partial derivative of the *outer* function (f in eq. (2.7)), with respect to the *inner* function's (g in eq. (2.7)) change. We also need to calculate the change of the *inner* function's change with respect to the remaining expression (x in eq. (2.7)).

Coming back to f_{sphere} from eq. (2.4) and its computational graph in fig. 2.1 we first run the expression in forward-mode in table A.2 on the left-hand side. This calculates all intermediate values v_n.
Now we can run the backward trace by applying eq. (2.7) for each dependency of a node v_n, by summing up the contributions. Let us observe the square-root node v_8. The graph fig. 2.6 makes clear, that it only depends on the value from the (in forward-mode) following subtraction of the constant $radius$. We can therefore build the contribution of $\overline{v}_8$ as shown in eq. (2.8):

$$\overline{v}_8 = \frac{\partial y}{\partial v_{10}} \frac{\partial v_{10}}{\partial v_8} = \overline{v}_{10} \frac{\partial v_{10}}{\partial v_8} \tag{2.8}$$

Note that the reverse accumulation calculates the derivative of all inputs, for one output, which practically is one row of the Jacobi-Matrix.
Similar to forward AD, backward AD can be applied to all outputs one after another, to obtain the full Jacobi-Matrix.
The main drawback of reverse AD is, that it needs to save the intermediate variables of the forward-trace. Therefore, memory consumption grows with the number of intermediate variables, which can become a problem in practice.
Implementations can circumvent this drawback by only saving needed variables (through data-flow analysis) [20, 103] or via lazy allocation techniques, which is an area of active research [79].

2.4.4. By Optimization

Retaping

Both forward AD and backward AD use an intermediate computational-graph to apply their respective technique with respect to either an input, or output variable. The idea of retaping stems from the observation, that the graph stays the same, as long as the expression does not change. The step of building the graph, and in the case of reverse AD calculating the intermediate variables can be saved for all executions, following the first one.
The application is comparatively simple, but comes with an added memory cost, as specially to forward AD. Mitigation in the form of tape size minimizing [27], or reuse centric and streamable data structures exist [41].

Checkpointing

For reverse mode AD the memory requirement to compute all adjoint values is in principle proportional to the operation count of the underlying function [39][7].
Checkpointing is the idea to only save the program state at defined *checkpoints*, and recompute the values in between the checkpoints when needed. This exploits the fact that programs can be split up into multiple segments. Checkpointing effectively allows us to trade memory

[6] Sometimes called *sensitivity* in the machine learning space.
[7] Cf. Theorem 3.3 Chapter 12

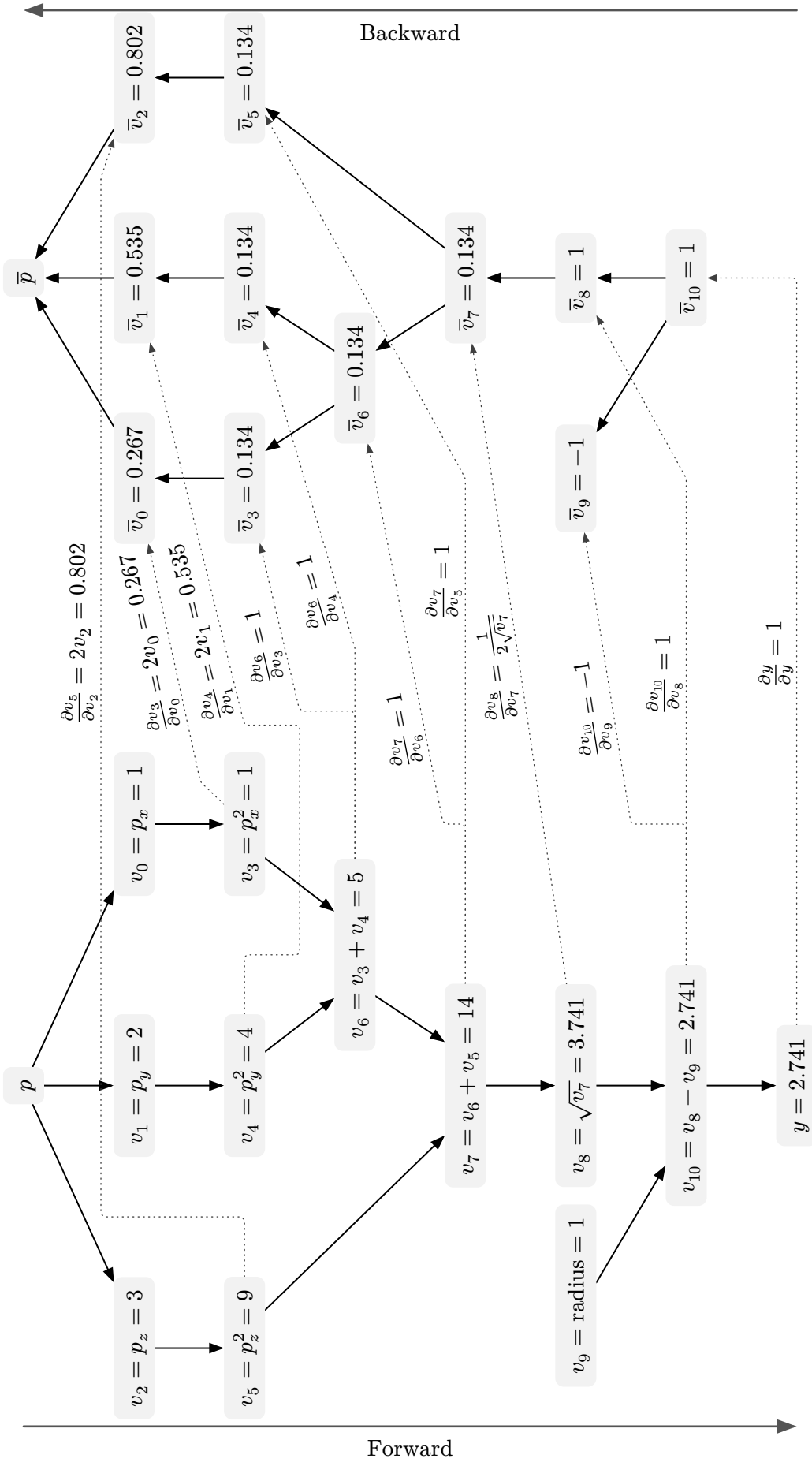


Figure 2.6.: Backward differentiation applied to f_{sphere}

for time by placing more (higher memory requirement) or less (higher recomputation time requirement) checkpoints.
Naturally the question arises where to place those checkpoints, how many, and what to actually save. Naumann proofs that finding checkpoints in a way, that minimizes the runtime for a given amount of memory is NP-complete [91].
If a program can be split into comparable equal-cost steps, logarithmic growth for both the temporal and spatial complexity can be realized [36].
Gebremedhin et al. also mention several advantages that get unlocked if the evaluation of a function is *time-step-like* [33]. Overall, placing checkpoints becomes easier, and possibly optimal [38], the more equal the segments of the function become.

Parallelization

Checkpointing exploits separability of functions to decrease memory usage. Separability can also be exploited to evaluate AD in parallel. Tapernade describes a vectorization technique [44] based on the idea that one can differentiate multiple points on the same function at once. For instance given data points $p_0, p_1, p_3 \in \mathbb{R}^3$: The partial derivative of the x, y, z components of all points can be executed *together* (in forward mode). In practice, this allows building loops that are data independent (since the points do not depend on each other), which then compute in parallel in a multithreaded execution context like OpenMP.
PARAD introduces possibly the first work-efficient parallel reverse mode implementation [57]. The paper describes the determinancy-race-free fork-join AD algorithm, the actual implementation of that algorithm as well as proof for its polylogarithmic scaling.
The current parallelization efforts mostly focus on run-time parallelization, which makes them a run-time optimization effort.

Sparsity exploitation

For large input and output sizes, problems exist, that produce Jacobi-Matrices that contain numerous zero values. Such matrices are also called *sparse*, since only some values contribute to the final derivative. Optimization techniques can exploit that fact by completely skipping the differentiation for such entries, and therefore speed-up the overall calculation time, as well as the time needed to generate the related code.
The challenge in this technique is detecting which entries are zero, before calculating the actual value. According to Gebremedhin et al. the proposed techniques can be split based on run-time and compile-time detection [33]. They identify two main run-time techniques: The *sparse-vector* approach defers allocation of result vectors (of the Jacobi-Matrix) to the first-use of the result. The *bit-vector* technique utilizes a pre-computation step, that treats the computation-graph as a OR-network. It performs a forward sweep tracking whether a result is in-use or not.
Compile-time techniques benefit from typically less strict time constrains to detect zero-values. Coleman et al. show that the problem can be mapped to graph (bi)-coloring [15]. They also point out that the function segmentation mentioned in section 2.4.4 can also be exploited with relatively dense matrices.
Gebremedhin et al. provide an extensive survey of this idea and its application, as well as practical difficulties [34].

Other optimizations

Further, more software engineering adjacent techniques are described in various implementation papers. Memory management techniques like deferred / lazy allocation [87], cache and

streaming friendly ordering of data [41, 53] are a source of real world performance gain, as specially for data intensive application like back-propagation in ML.

2.4.5. By transformation level

Operator overloading

Operator overloading describes the technique of redefining the operators in terms of AD as described in section 2.4.3 & fig. 2.5. This lets a user of that feature define functions as they would usually do, and the AD implementation takes care of applying the given technique on top of a program.
This method is popular with forward-mode AD implementation in operator-overloading capable languages, since it is implemented comparatively easily in a self-contained library. A good guide can be found in *Forward-Mode Automatic Differentiation in Julia* [97] which uses the dual-number technique. By defining a dual-number type, and a set of elementary operations, the library facilitates competitive AD performance in an easily distributed package.
Reverse mode AD is harder to implement, since following the reverse trace of that technique solely from within the host language can become challenging.
The main draw-back of this method is its *in-language* nature. Since it redefines the implementation of operators, the code quality depends on the input program. This does not allow an implementation to optimize based on properties like locality of data, redundant loads etc. Instead it depends on either the interpreter, or a following compilation step to find, and apply those.
Implementations are also language or rather library dependent.

As code transformation

Code transformation contains all techniques that either employ some form of meta-programming, for instance template-programming in C++ [46], or manual code-transformation by parsing code and rewriting it.
Compared to operator-overloading, implementations are more involved, since not only are the operands redefined, but some kind of understanding needs to be built for the program at hand. Either in the form of auxiliary structures for meta-programming, or in the form of parsing source code, and building related intermediate representations. This also introduces the challenge of keeping the AD implementation up-to-date with the language's semantics.
The advantage of this technique is its compiler-like standing in the tool chain: The implementation can exploit context information of the program, can analyze data-flow, rewrite parts of the program etc. As specially when implementing reverse mode AD, this freedom allows for better optimized output, as argued by the Tapenade specification [44].
A source-code independent implementation can be used with programs written in different languages. This allows better reusability of the implemented strategies. For instance, Tapenade is implemented for Fortran and C [44]. Another approach is not using a source language at all. Aesara [21], which is a fork of Theano [106], lets a user build the computational graph directly via a Python application programming interface (API), then optimize it and compile it into C, JAX [9], or the NUMBA [68] runtime.

Post code optimization transformation

A natural extension to source-transformation AD are *post optimization* AD compilers. Nauman et al. show that the already existing intermediate representations of compilers can be used to implement AD within a Fortran compiler itself [92].

Language independent compiler frameworks like LLVM lent them self naturally to this strategy, since multiple language frontends, and machine backends, as well as a wealth of common optimizations are already provided. One such AD framework for LLVM is Enzyme [90]. It operates on the LLVM (LLVM)-IR, which allows it to emit code for numerous central processing unit (CPU) architectures as well as GPUs. Official and unofficial frontends include Fortran, Julia, Rust and C/C++.
Recently multi-level IRs like MLIR or xDSL started acting on the observation that higher level information could be exploited for more informed optimization. Often this means keeping *domain specific* information longer, before lowering into a less specialized/more general format. This idea is also applied to AD in LAGrad [94] for higher level MLIR. While Enzyme is also implemented for MLIR, its AD application happens at the comparatively low-level LLVM abstraction. LAGgrad in turn is applied to high level dialects like *Tensor* or *Linalg*. Peng et al. [94] show that this allows them to implement traditional AD optimization like tape-size-reduction efficiently and achieve speedups of 4.2× in selected benchmarks compared to Enzyme, which is already one of the fastest AD framework at the moment [89].
The observation is, that on one hand postponing the differentiation step to post-code-optimization is beneficial, on the other hand lowering the step *too far*, is not the optimal strategy. The general observation that higher level information can assist in optimization for compilers holds true for AD as well.

2.4.6. Reviewing tool

There are several criteria under which we can group AD tools. We now chose all recent professional tools and compare them in each of those groups.

Selected AD tools

We include tools in active development past 2023, that have more than 100 confirmed users. The development is confirmed either by public code-forge[8] commits, or via public research results. The user-base size confirmed either by public usage records (issues, commits, forks etc.) or via public citation count of related publications since 2023.
Note that we can not guarantee that all tools under those criteria are included. The AD-tool space is big and fragmented among different scientific fields. As specially the case of *few developers, many users* is hard to distinguish from *few developers, few users* from the outside.
Refer to table A.1 for a list of all selected AD tools and a short description of each.

Interpretation

Table 2.2 collects all selected AD implementations into an overview in relation to the implemented mode (green), optimization (blue) and implementation level (red).
Mode wise, we can see that most implementations chose to implement both, forward- and backward-mode AD. There are only two outliers.
ForwardDiff, as the name suggests, implements forward AD only. The benefit is a light AD library compared to more sophisticated implementations. This is also in line with other already mentioned *forward-only* implementations.
The second outlier is Tensorflow that only implements backward-mode. Since Tensorflow is explicitly a ML framework and not a AD tool, this makes sense as well. Forward-mode is only useful for small input sizes, which is almost never the case for ML problems.

[8] e.g. GitHub, GitLab etc.
[9] Any of the marked features is implemented in one of the packages of https://juliadiff.org/
[10] The tool lets you build the computational graph from a Python API.

Tool	Forward Mode	Dual Numbers	Backward Mode	Retaping	Checkpointing	Parralelization	Sparsity exploitation	Operator Overloading	Code Transformation	Post-Opt. Transformation
ADOL-C [111]	■		■		■	■	■	■		
Aesara [21]	■		■		■	■	■		■	
AutoDiff [71]	■	■	■				△	■	▲	
Autograd [78]	■		■					■		
Casadi [2]	■	△	■		■		■		■	
CLAD [109]	■		■	■	■				■	
DiffSharp [6]	■		■					■		
Enzyme [90]	■		■	■		△	■			■
ForwardDiff [97]	■	■				■	■	■		
JuliaDiff[9]	■	■	■	■	■	■	■	■	■	■
Minkowski Engine [12]	■		■			■	■		▲[10]	
PyTensor [22]	■		■		■	■	■		■	
Tensorflow [80]			■		■	■	■		■	■
Zygote [51]	■		■		■		■			■

Table 2.2.: AD tool feature comparison
■: Implemented, ▲: Partially implemented, △: Unknown

The most popular optimization is sparsity exploitation (10/14) followed by checkpointing (8/14) and parallelization (7/14). The least popular being retaping. This does not mean that retaping is unuseful. ML focused frameworks (which is the majority here) often need to optimize for memory usage. Retaping, which improves runtime by using more memory, is not applicable in that case.

There is no clear winner for the transformation level.
An interesting observation is, that the optimization count on average increases when lowering the implementation level. While operator overloading implementations have on average 1.5 implemented optimizations, code-transformation based implementations increase that to 2.7 on average. Post-optimization implementations lead with 3 on average. The hypothesis is, that lower transformation level benefit from more accessible data-flow information and transformation capabilities, which are needed for powerful optimizations.

2.4.7. The case for domain specific compilers

When considering domain specific compilers in the context of AD, we can relate two points. The first one is, that AD libraries and tools are often domain specific them self. This is usually the way optimizations are chosen, and user interaction is modelled. Considering for instance JAX, which focuses on machine learning, the API is Python, which is widespread in that space, and it focuses manly on backward-mode, with memory conscious optimization, parallel computation and sparsity exploitation.
In contrast, DNAD, which originates in computer physics focuses on forward-mode, and uses Fortran. Both choices make sense, if you encounter *small input, big output* problems and already use Fortran code bases, which are wide spread in that space as well.
The compiler-based AD approach in LAGrad [94] shows that an implementation can exploit domain specific knowledge at that level as well. Coming back to f_{sphere} from eq. (2.3), the expansion of the $||p||$ term to $\sqrt{p_x^2 + p_y^2 + p_z^2}$ transformed one operation into nine, in order to arrive at an elementary-operation-only compute graph.
A domain specific compiler might be able to implement the $||p||$ operation as an elementary-operation. This would effectively shrink the compute graph and possibly allow better derivatives for that operation.
Domain specific differentiating compilers can exploit their focus by implementing a suitable interface for the users of AD, use (only) fitting optimizations and implement a customized set of elementary operations.

2.4.8. Distance fields in AD

Distance fields, or scalar fields represent $n - Input,\ 1 - Output$ problems, where n is the dimensionality of the distance field. For a 3-Dimensional field we have three inputs (x, y, z), and one output, the (signed) distance. If those are the only variables the differentiation is interested in, the previous research suggests that forward-mode is powerful enough, since 3 could be considered a *small* amount of inputs. With that technique properties of the spatial parameter can be analyzed, like the normal-vector, or the curvature property. Section 2.2 shows however, that more sophisticated applications, like distance field reconstruction need access to not just the spatial parameter's derivative, but properties of the encoded primitives as well.
For instance, imagine we want to fit a sphere to some collision problem, the solver might have to change $radius$ of f_{sphere} (in eq. (2.4)), in order to find a result. The naive search would just change $radius$ until it finds a result, which potentially takes long, or never ends. An informed search however, could employ gradient descent over the parameter $radius$ to find a result faster.
Transferring this idea to more complex models, the input space becomes much larger than just the spatial parameter. In that case backward-mode AD becomes more viable.

Another consideration are *higher order derivatives*. Referencing the *small-input* problem from before, the first derivative of a signed distance function in 3-dimensions yields three partial derivatives (see section 2.4.2). Each of those needs to be differentiated again, to arrive at the second partial derivative. A naive implementation could implement the second derivative as two applications of the AD pass to the same expression. There are also specialized implementations for forward- [29] and backward-mode [70] [42] as well as mixed techniques [104]. Depending on the intended use case those optimizations might make sense, but are not general enough to make a general recommendation.

3. Methodology

Chapter 2 explores applications of AD for DFs. An observation in other AD implementations is their domain specific choices. The choice of implemented modes, optimization, and the implementation level all depend on the application domain. We now motivate our domain specific considerations, informed by the prior research.
We then introduce Vola's implementation and how the commonly used structures translate into that environment.
We close with the chosen correctness criteria for testing as well as performance criteria for comparison to other implementations.
The implementation of the chosen strategies is discussed in chapter 4.

3.1. Domain specific considerations

The implementation is realized within a domain specific optimizing compiler, Vola's optimizer. The language's objective is generating machine executable code, that represents signed distance functions as well as adjacent, user defined properties. The Vola-DSL lets humans define such functions. Vola's compiler toolchain currently focuses on GPU executable code.
Vola's compiler has relaxed precision rules in favor of higher flexibility while defining signed distance functions and while optimizing high-level code. This applies to floating-point operations as well as the reinterpretation of high-level operations.
Control-flow is structured and deterministic. This means only bounded loops, and no infinite-recursion of functions.

3.1.1. Forward vs. Backward mode

Both modes are good at two distinct input-output pattern. Forward mode usually requires less memory, but is only efficient for *Few Inputs, Many Outputs* cases. Backward-mode in contrast is more effective for *Many Inputs, Few Outputs* problems. Since our domain contains all distance functions, which includes highly complex, highly parameterized functions, we can't rule out the *Many Inputs* case.
In practice any use case that does not use *many* inputs fits the forward-mode. This includes simple applications like a first-derivative of a SDF at any given point in a 3D-space. Therefore, for optimally reasons the *Few Inputs* case, where forward-mode fits best, can't be ruled out as well.
We therefore come to the same conclusion as most (cf. table 2.2) sophisticated AD-tools, that both modes need to be implemented to arrive at an efficient implementation.

In practice the tool needs to decide which mode fits best for a given problem. The main criteria are the input and output size of a given function. Deciding at which point *few* changes to *many* is not a simple task. We propose counting the number of operations and letting those influence the heuristics. Each operation introduces memory overhead for backwards-mode AD[1]. The heuristic should approximate the point, at which repeated backward-mode AD with intermediate values outweighs repeated forward-mode application without intermediate values.
In practice the heuristic could be informed by profiling representative use-cases and encoding the result into a simpler model.
This also opens up the possibility for self-optimizing strategies, possibility based on ML like [107] or [88].

3.1.2. Implementation Level

Section 2.4.5 shows that the compiler level is a promising level for AD implementations. There is an optimal level for that implementation somewhere between high-level, domain specific dialects and low-level, hardware-like dialects. Vola's compiler toolchain already encodes the multilevel dialect approach in its optimizer. The AD implementation can therefore be implemented on an algebraic level, that is already present. This allows domain specific exploitation, like cross-product derivatives, that need to have notions of vectors and a cross-product-operation to be present. The chosen IR also provides a proven framework for data-flow and control-flow analysis that is needed to facilitate the implementation in chapter 4.

3.1.3. Optimizations

The implementation does not use any of the optimizations of section 2.4.4. It exploits compiler native optimizations, like dead-code-elimination, to reduce code size and constant folding, which eliminate statically known operations. Those optimizations can be applied before the differentiation pass, to reduce the amount of operations in the computation graph, and afterward, to optimize the derivative calculation itself.
The implementation can also make use of mixed-mode strategies. *Activity-analysis* has to find values that are related to any x for a AD pass $\frac{\partial f}{\partial x}$. We can exploit this information to not just implement the chain rule, but the product-rule, constant-factor-rule and others. This shortens the generated operation graph for derivatives.

3.1.4. Elementary operations

The choice of elementary operations has a big impact on the AD-implementation's complexity and computational graph complexity (e.g. eq. (2.3) & eq. (2.4)).
The optimizing compiler's high-level dialects encode common arithmetic operations as well as vector-arithmetic operations and common programming related expressions (eg. linear interpolation[2]). We chose to keep all high level operation that have a defined derivative in the elementary operation set. Some higher level operations can be replaced with an equivalent graph of multiple defined operations (again, see eq. (2.3) being rewritten as eq. (2.4)).
The compiler also contains *non-arithmetic* operation, like indexing into a vector. Those operations are transparent to the AD transformation. They are used as-is without changing them at all. If needed, arguments to those operations are transformed, however.

[1] if no memory optimizations like checkpointing is implemented.
[2] Also known as *lerp*, or *mix*.

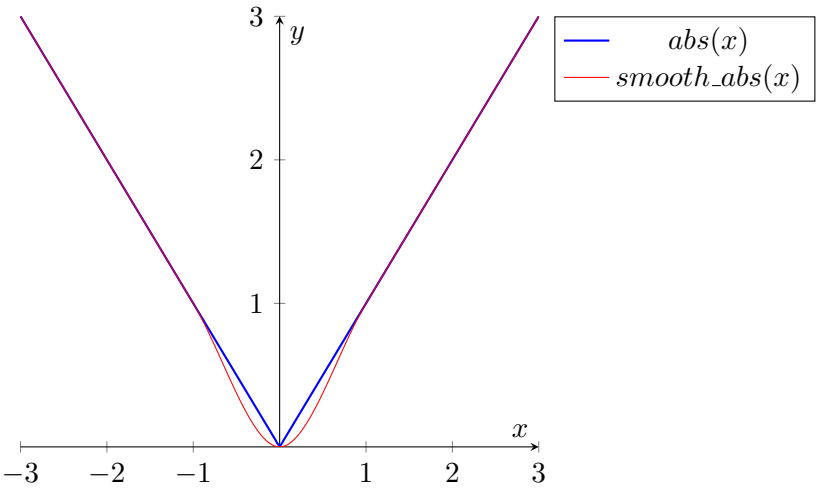


Figure 3.1.: *Smooth_abs* approximation in XAD [105]

3.1.5. Differentiability

A so far unmentioned practical problem are non-differential functions. In practice, computer code is rarely fully differentiable. Things like control flow can, but often don't lead to differentiable functions. Other expressions like *abs*, which returns the positive of any value, is only differentiable in a certain range. In this case, *abs* is differential for non-zero values.
The question arises, how non-differentiable expressions are handled in our case. We introduce two compiler modes, a *mathematical correct* mode, that aborts differentiation when it encounters a non-differentiable expressions, and a *practical* mode, that applies possibly non equality-preserving translations to the expression, to make it differentiable.
The resolution idea to this problem stems from code-transformation tools. A *canonicalization* pass translates non-differentiable functions to differentiable approximations. For instance, *abs* is approximated around[3] 0.0 by a differentiable function in fig. 3.1. This approximation is chosen in a way that makes the whole function differentiable.

A second resolution strategy is explicitly catching non-differentiable cases in the canonicalization. The $x = 0.0$ case in fig. 3.1 could also be handled by an explicit branch, that emits a constant 0.0 for the derivative.

The AD-implementaiton is control flow (i.e. branches and loops) *blind*. For a branch, only the taken branch's derivative is calculated.
Loops in the DSL have known, static bounds[4]. They are treated as sequential re-execution of the same code. The derivative of a loop is simply the derivative of the loop unrolled *loop-iteration-count* times. Better solutions to the problem, that do not need to unroll loops exist [51]. This would lead to drastically less code, especially for high iteration counts.
Control flow always aborts in the *mathematical correct* AD mode. Otherwise, the compiler would have to include proofing capabilities for control flow boundaries. Such a system would have to prove that the boundary of two branches (or loop iteration pairs) is in fact continues.
The *practical* mode can employ approximation at control flow boundaries [13]. The idea is, that control flow is treated as a composition of two differentiable functions. The approximation smoothly interpolates both functions at the control-flow boundary[5]. This leads to a differentiable approximate expression.

[3] In this case the approximation region is $c = 1$ to illustrate. In practice this can be chosen smaller.
[4] In plain words, we know, at compile time, how often a loop is executed.
[5] Given an *if-statement* that chooses a path *A* for $x < 0$ otherwise it chooses path *B*. The approximation can interpolate the values of *A* and *B* for $x = 0 \pm c$.

3.2. Implementation environment

The AD feature needs to integrate with an already existing language and compiler. While not strictly part of the AD technique itself, the syntax used to describe differentiation is the only contact surface from a user to the compiler feature. The following discusses how we interface from the users (syntax) perspective down to the compiler's IR description of a differentiation.

3.2.1. Syntax

Users access AD capabilities of languages either through bespoke functions, or sometimes special syntax in DSLs with AD.
A common syntax is a *function-call* like pattern with three parts (cf. fig. 3.2): The function's name determines the AD mode, the second, and third part signal the differentiated expression and with respect to which[6] parameter differentiation is done.

```
//forward mode
let result = forward_gradient(*expression*, *wrt-argument*);
```

Figure 3.2.: Function-call style syntax

This kind of syntax has the advantage of being *within* most languages. For operator overloading or source transformation implementations, the function call serves as the AD entry point. From here, either the operators of `*expression*` can be modified with respect to `*wrt-argument*`, or a source transformation can start. Some implementations allow `*wrt-argument*` to be of non-scalar type. This effectively makes the differentiation either directional derivative or a vector gradient, depending on the signature of `*expression*` and `*wrt-argument*`.

DSLs can use special syntax to signal the creation of an expression's derivative. In theory the appropriate mathematical notation like $\frac{\partial f}{\partial x}$, $f'(x)$, ∇f can be implemented. In practice, those are symbols that are not present on most keyboards, which makes this rather unpractical. If a none text-notation (i.e. node graphs) or a integrated development environment (IDE) is already given to express the DSL hover, this might be viable.

Vola takes parts of both (cf. fig. 3.3). Since the compiler should use a heuristic to decide for forward or backward mode, the mode is not signaled by the syntax. Instead, a `diff` function call signals the AD entry-point.

```
csg sphere = Sphere(1.0);
let at = [1,2,3];
let result = diff(eval sphere.Sdf3d(at), at);
```

Figure 3.3.: Derivative of a sphere in Vola

The differentiated parameter can either be a variable, or expression. The `*wrt-argument*` can be either a scalar or vector parameter. The result type of the derivative is calculated from both parameters in accordance with the rules established in section 2.4.2. This means taking the derivative of a scalar function returns a scalar derivative. Taking the derivative of scalar field returns a vector of the field's dimensionality, etc.

[6] With respect to (WRT)

3.2.2. Integration into the IR

The central data structure to both AD modes is the computation graph. This is a graph with *elementary* operation as its nodes, and edges from each operand's value producer. This is equivalent to the data dependency graph. The IR employed by Vola's optimizer is the data-flow centric RVSDG. Generating the computational graph of any expression in this graph is as simple as following all *value-dependency* edges of an entry-node[7].
Vola therefore does not have to build a computational graph for the AD implementation, since that is already established within the IR. The AD specific implementation focus on the already established canonicalization pass, the actual AD-mode specific transformation of the graph as well as post-transformation optimization. The latter being standard none-AD specific optimizations.

3.2.3. Code generation concerns

Vola's focus is GPU executed graphics (in contrast to GPU executed compute) code. A CPU targeting code generation backend exists to verify correctness against other AD implementations, as well as making performance comparisons possible. The implementation would be constraint to other GPU capable AD implementations otherwise.

3.3. Testing the implementation

3.3.1. Correctness

The implementation needs to be correct on a mathematical layer as well as on an algorithmic one. This means in practice that the implementation needs to identify non-differentiable expressions correctly, and transform differentiable expressions reliably into the result.
We build a test framework that uses Enzyme [90] to facilitate differential testing. The idea is to use the same test-code on both systems, Enzyme and Vola. The emitted code is then executed and checked for differences.
We can't compare the emitted code, since both systems transform and optimize code differently.
Differential testing does not allow us to catch all errors. Floating point error accumulation is not considered, for instance. Nevertheless, it allows the system to catch basic differentiation mistakes.

3.3.2. Performance

We test performance similarly to correctness in relation to Enzyme. Shared test cases, that are compiled by both compilers, are executed on several data-points. Compile-time, runtime, and code-size for both are measured.
We split the compile-time into two measurements, *overall compile-time*, and *time taken for AD*. This lets us identify whether differences in the compile-time stem from the differentiation pass, or other unrelated compilation details.

[7] Ignoring inter-procedural nodes, like apply-nodes, and intra-procedural nodes like θ-nodes. Integrating those into the computational graph is simple as well.

4. Implementation

The implementation description follows the natural compilation flow. We start by describing the initial compiler state, in which the AD takes place. General preparation of the RVSDG is followed by analysis of partaking values in the derivative (green in fig. 4.1). Next, the actual AD algorithm is applied to the graph, to generate the derivative value of an expression with respect to a singular value (blue in fig. 4.1). In a final step, the partial derivatives are combined into the final value as defined by section 2.4.2 (red in fig. 4.1).
The described implementation lives in the `autodiff` module of the `vola-opt` crate of the `vola` [85] repository.

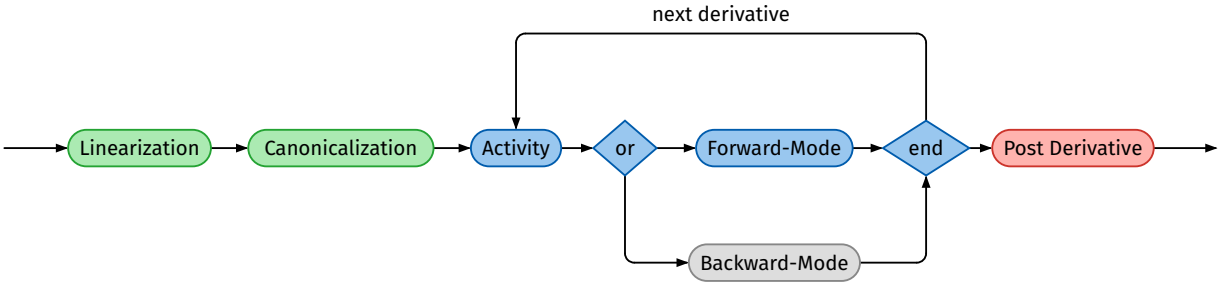


Figure 4.1.: Differentiation pipeline overview

4.1. Preparation

We show the AD calculation based on the already known sphere example from section 2.4.3. We start out with the Vola code in fig. 4.2a. After the compiler has finished all pre-AD work, we end up with the graph in fig. 4.2b. The orange node declares the entry-point to the AD pass. The green nodes are the expression that is being differentiated.

The entrypoint for the AD implementation is `Optimizer::dispatch_autodiff`. The first action is to find all nodes of the `AutoDiff` dialect, in our case the singular green `AutoDiff` node. Before building the derivative of each of the discovered nodes, we build the topological order of the derivatives. This lets us solve the higher-order derivative problem. The idea is to first solve *inner* derivatives, which is equal to being *earlier* in the topological order in the RVSDG. This effectively lets the user write an expression `diff(diff(expr, x), y)`, which is the second derivative of `expr` with respect to `x` and `y`.

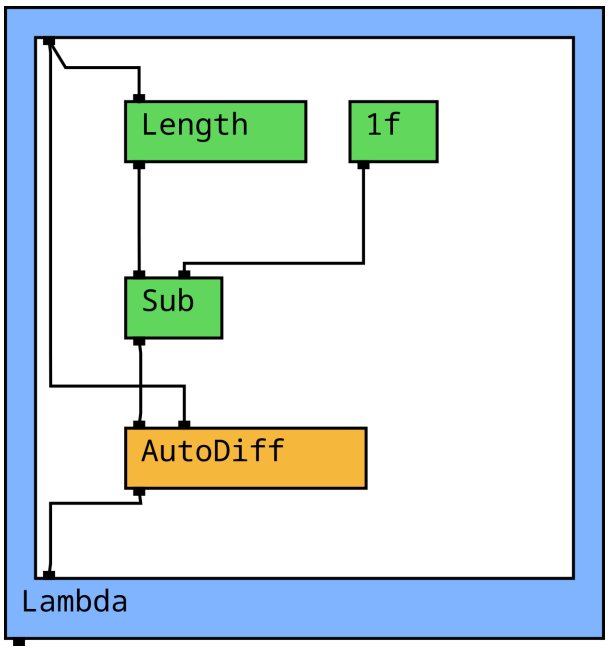
```
entity Sphere(rad: s);
concept Sdf: vec3 -> s;
concept Gradient: vec3 -> vec3;

impl Sphere for Sdf(at){
    length(at) - rad
}

operation MyAdOp();
impl MyAdOp<sub> for Gradient(at){
    let sd = eval sub.Sdf(at);
    diff(sd, at)
}

export mysdf(at: vec3){
    csg sphere = MyAdOp(){
        Sphere(1.0)
    };
    eval sphere.Gradient(at)
}
```

(a) Vola code



(b) RVSDG immediately before starting AD process.

Figure 4.2.: Compiler state before sphere differentiation.

The next step is to dispatch the actual derivative calculation, in that order.

In the future this would be the location, in which a heuristic decides which AD implementation to use, based on the expression, and context information. In practice, we currently only have a forward implementation, we therefore always dispatch the same algorithm.

4.1.1. Linearization

In forward-mode we have a practical problem at this point: `diff` can be used with respect to any expression, including expressions that are vector or matrix valued. Recall that section 2.4.3 established, that forward-mode creates the derivative of an expression with respect to one scalar-valued argument. We fix that issue by *linearization* of the AD entrypoint. Practically this transforms an entrypoint `let d = diff(expr, [a, b, c]);`[1] into `let d = [diff(expr, a), diff(expr, b), diff(expr, c)];`.
Note that the type of `d` in this case still depends on `expr`'s type.
The process can be observed in fig. 4.3. Note how the graph now has three (yellow) `AutoDiff` nodes, each connected to the original expression, as well as one index (red) into the region's argument.

4.1.2. Canonicalization

Next we canonicalize the dependencies of the *to-be-differentiated* expression as described in chapter 2 and section 2.4.3. We again traverse dependencies of the expression in topological order. For each node, we determine whether the node must be canonicalized. If that is the case, based on the before established approximation criterion the node is either transformed into an equivalent, differentiable expression, a differentiable approximate expression, or returns an error.

[1] Where a, b, c, are scalar-valued, and the *with-respect-to* argument is vector-valued in consequence.

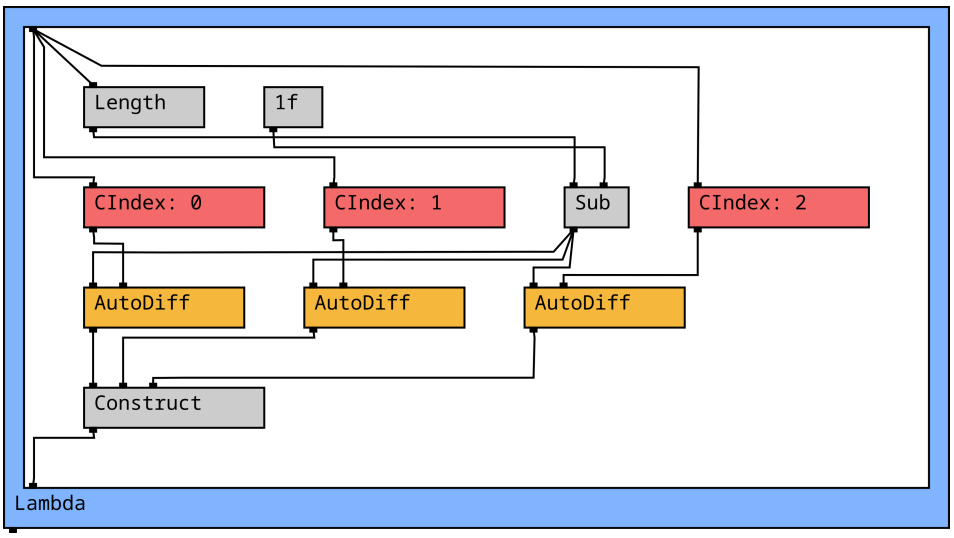


Figure 4.3.: AD entrypoint split into three.

An advantage of this approach is, that we only have to canonicalize an expression once. So even if we had to linearize the AD node before, the canonicalization expense does not grow. In fact, even if another AD node later on uses parts of the same expression, it won't have to re-canonicalize the sub-expression.
A current disadvantage is, that any approximation that is introduced into the expression will also apply to non-differentiated values that use that expression. However, this also makes the result more predictable, since not only is the derivative based on the approximation, but the value itself too.
In our example case (cf. fig. 4.4) only the `Length` node must be canonicalized. In this case, into $\sqrt{x^2 + y^2 + z^2}$, or in other words, the standard Euclidean distance formula. The green nodes mark that expression.

Control-flow

This pass is also the pass that handles control-flow. Internally, the RVSDGs's control-flow nodes (γ- / θ- nodes) are handled the same way as any other node that needs to be canonicalized.
We currently handle branches by making them *transparent* to the differentiation. At runtime we use the same decision value to choose a branch. Instead of returning $f(x)$ however, we return $\frac{\partial f}{\partial}$.
For loops, we simply unroll the loop into one big expression. This is at the expense of code size, but keeps the implementation simple. Otherwise, we would have to implement some kind of intermediate value analysis. This is a well explored field [48, 51]. This approach potentially introduces redundant AD workload. Given an expression e that is *within* a loop, but could be factored out[2], e will be differentiated *iteration-count*-times. This can be mitigated by either pulling out such expressions first, or by unifying common-expressions after unrolling, but before differentiating.
We can use unrolling because Vola's language only allow static loop bounds. This was decided before, because the language first focused on real-time rendering code for GPUs. In that context unbound loops, and big loops are unwanted, since they can lead to unanticipated long runtime of a kernel, or even GPU timeouts.

[2] meaning a loop-invariant expression

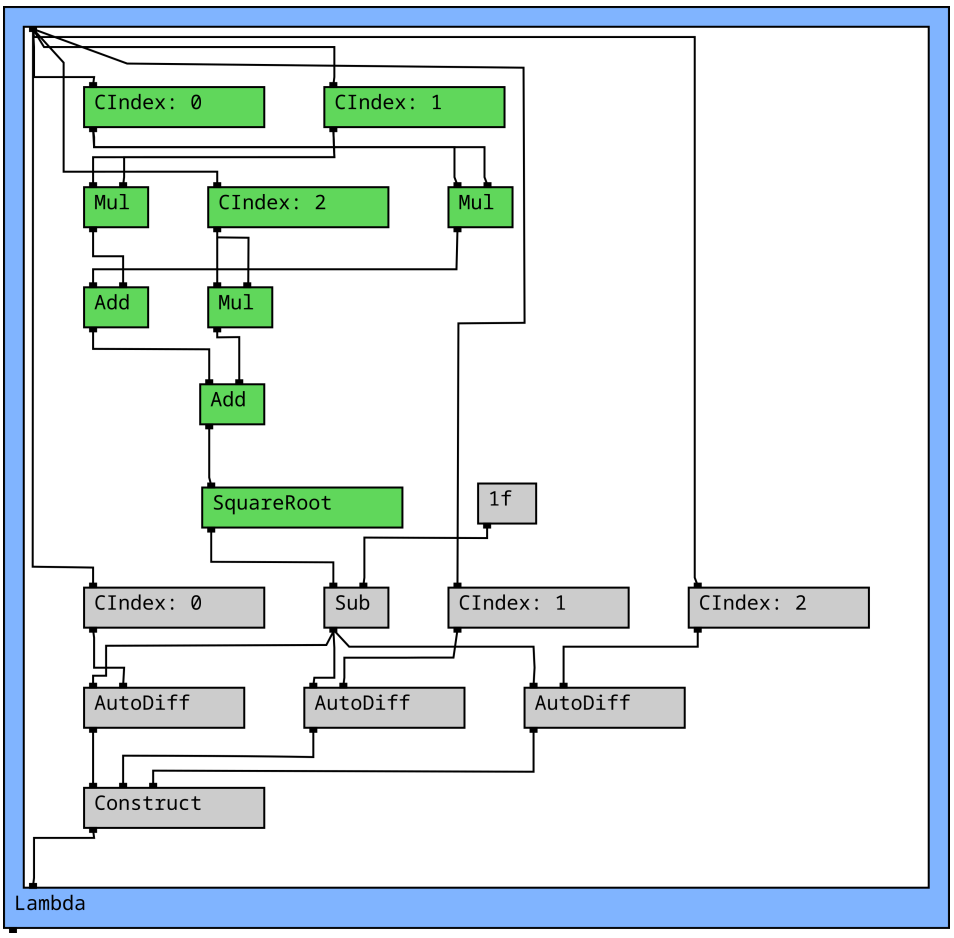


Figure 4.4.: Canonicalized expression

4.2. Activity analysis

We now consider the expression connected to any `AutoDiff` node in *AD-canonical* form. The next step of the algorithm identifies so-called *active* values. This is a terminology we borrow from Enzyme[90]. We consider a node in the graph *active* if it can propagate a differential value[3]. In other words, if we are building the derivative for x, any node that could somehow contain x, or a value that is impacted by x is considered *active*.
It is to note that our analysis can be simpler than Enzyme's, since we do not have a concept of *memory* or *pointers*. We can always reliably infer what values are connected to x.
Coming back to the example, recall that we split the AD with respect to `at` in fig. 4.2a into three in fig. 4.3, which leaves us with three `AutoDiff` nodes, one for `at.x`, one for `at.y` and the third for `at.z`. In the figures x is accessed via `CIndex: 0`. y/z are at index 1 & 2 respectively.

The first part of the activity analysis is a backward search from the `wrt-argument`, so the second input to a `AutoDiff` node. It traces that dependency graph until it ends at any argument to the surrounding region, or a value producing node[4]. This allows us for instance to include a vector $\vec{v}$ if that vector contains x.
We also record which component in that vector contains x. This way, if any other node uses x at some other point in the graph, we know that this value is *active* as well. This process is the same for all shapes of algebraic type system in the DSL.

[3] The ∂x in $\frac{\partial f}{\partial x}$
[4] Meaning anything that *calculates* something. In practice we can just check if a node is of the *algebraic* dialect.

The system now has a set of initial *active* nodes and ports in the dependency-graph of our `AutoDiff` node. Later, whenever activity needs to be considered, the algorithm can *ask* that system whether a node or output-port is active. It can then utilize the just build initial activity state, as well as cached activity state, to derive the asked-for activeness. In other words, the system can lazily update each node's activity once it is needed. fig. 4.5 highlights active nodes and ports in the example graph. The initially active nodes are highlighted in orange, while the derived nodes are highlighted in green. All the colored nodes are considered *active*.
Note how the `CIndex: 1` and `CIndex: 2` nodes are not highlighted. While they do index into the `at` argument, they do not refer to `at`'s x coordinate, but y/z.

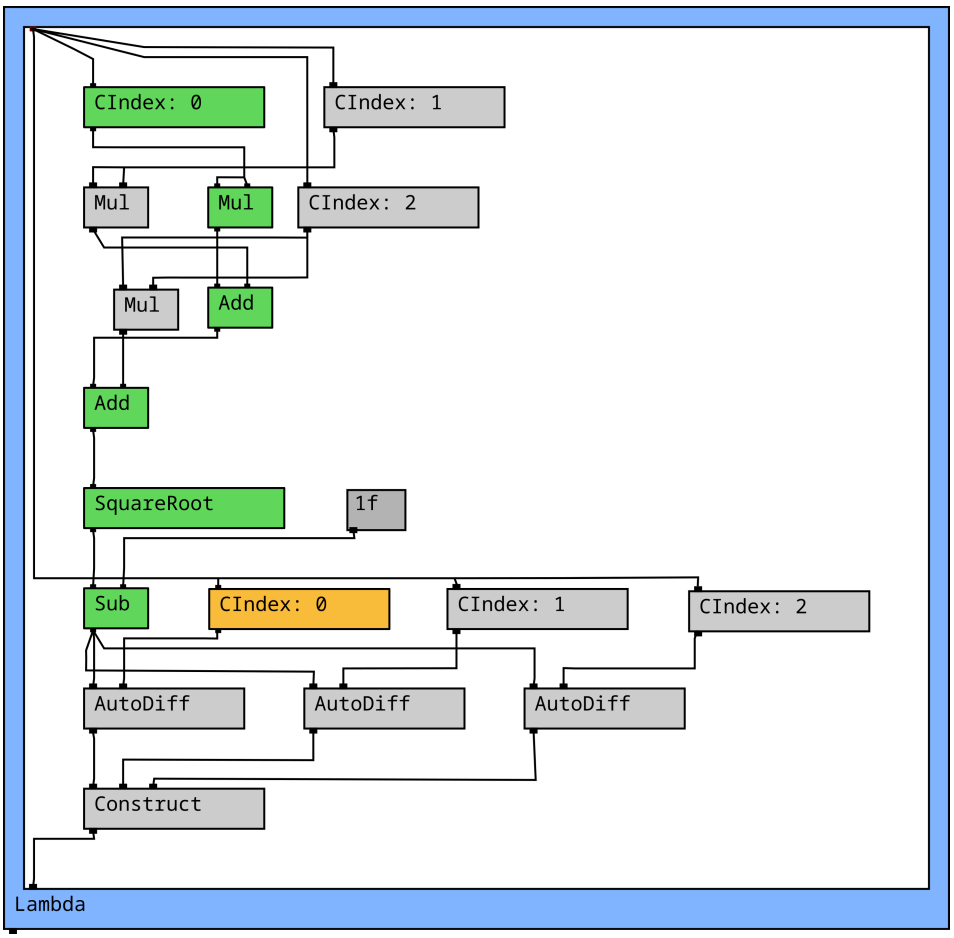


Figure 4.5.: Node activity in AD dependencies

In practice additional care needs to be given to nodes with sub-regions, for example derivatives within a loop. The same tracing idea applies, but across region boundaries.
The initial tracing logic as well as the lazy exploration resides in the `activity` module.

4.3. Generating derivatives

Both modes, forward & backward, rely on the ability to set up a node $f'(x)$ for any $f(x)$ they encounter. For that reason we implement that node creation outside of the mode's logic.

Simple nodes and ports

Most values in our graph are *simple* nodes of the algebraic dialect. We can decide whether they are active or not for each. If the AD algorithm request the derivative of a non-active

node, it emits a *zero* value of the corresponding type. A 3-component-vector typed node would therefore generate a 3-component-vector of zeros.
If the derivative of something that *contains* x is requested, a correspondingly typed value is created, where any element that was x is 1.0, and any non-x element is 0.0. This behavior originates from the application derivative rule for the pure differential value shown in fig. 4.6.

$$let\ f(x) = x \quad then\ \frac{\partial f}{\partial x} = 1$$

$$let\ f(x,y,z) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad then\ \frac{\partial f}{\partial x} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Figure 4.6.: Pure differential value creation

For any active value that does not contain the pure value x, we apply the corresponding derivative rule. At this point we can leverage the activity information to not only implement the chain rule, but quotient-, product-, scalar-rule etc. By adding the type system to that idea, we can even define the derivative for vector functions like dot-product and cross-product. We therefore don't have to break them apart, which lets us save operations for our derivative.
It is to note that in principle only the chain-rule and the derivative of each elementary operation[5] need to be defined. Employing those additional rules only allow us to keep the derivative's computational graph smaller.

Branches

For branches, we duplicate the γ-node[6], which contains both branches. We then import the activity state of each entry-variable of the γ-node into the region. Now we can build the derivative of each exit-variable within that region based on the imported activity state. This effectively creates a new gamma node that outputs the derivative of an exit variable based on the runtime branch criterion.
This makes the γ-node *invisible* to the derivative, since it does not contribute to the derivative. If both branches are not continues in their initial form, then the discontinuity is carried over into the derivative value. We discussed interpolation at the bounds of control-flow nodes (gamma and theta) in section 3.1.5. This strategy is currently not implemented though.

Inter-procedural nodes

Inter-procedural nodes, usually function calls, are handled similarly to branches. We copy the λ-node[7], import the activity state that is connected to the caller into that copy, and execute the derivative creation on that node's body.
The DSL does not permit recursive function calls. We can therefore be sure that the number of copied λ-nodes is bound. We currently don't account for redundant copies. For instance, given a function $g(x)$ that is differentiated with respect to x twice, we don't reuse the already created

[5] Any node that can occur in a canonicalized graph in our case.
[6] γ-nodes can be understood in principle as a switch-case node. For more detail see the RVSDG paper[96] and section 1.3.3.
[7] λ-notes can be understood as functions. For more detail, see the RVSDG paper[96].

$g'(x)$. To be able to directly reuse the function, we would have to make sure, that the activity for both differentiation cases is exactly the same at the call-site. Later common-node-elimination might take care of reuse regardless.

4.4. Forward mode

The forward mode follows the idea described in section 2.4.3 closely. It iterates the derivative expression's nodes in topological order. Non-active nodes are ignored. For each active node, we generate the derivative as described in section 4.3. Each node can request sub-expression derivatives. That allows them for instance to apply the chain rule. Since traversal takes place in topological order, the derivative of any sub-expression will always be present before the expression itself is handled. In math terms, given $f(g(x))$, $g(x)$ will always be handled (and emit $g'(x)$), before $(f(g(x)))'$ is requested. Building $(f(g(x)))' = f'(g(x)) * g'(x)$ therefore only needs to generate $f'(g(x))$ and hook it up accordingly.

Coming back to the example in fig. 4.5, the algorithm traverses all green nodes from the top to the bottom. The `CIndex: 0` node differentiates to 1.
The multiplication uses the product rule, since both inputs are active. The rule $(u * v)' = u' * v + u * v'$ applied to $u = x$ & $v = x$, $u' = 1$ & $v' = 1$ resolves to $1 * x + x * 1 = 2x$.
In the end the graph look like fig. 4.7. The green nodes are newly added nodes to the graph, while the red nodes are reused nodes that partake in the derivative calculation.
Note that this is the raw post-derivative graph. Many of those nodes could be folded, for instance the multiplication by 1 at the top, or the subtraction of 0 at the bottom.

4.5. Post derivative

For forward mode, the process is repeated for the remaining `AutoDiff` nodes. Afterwards it is advisable to run cleanup passes like dead-node-elimination, common-node-elimination and constant folding. It can happen that the original expression is fully transformed, which leaves it dead. We already saw that the simple application of derivative rules creates operations that can be reduced, like multiplication by one. The reason for that lies in the initialization of active values to 1 and passive values to 0. Both values are identity values to multiplicative and additive operations, respectively.
In our case, the unoptimized graph after AD has 56 nodes, while the cleaned final graph has 20 nodes and can be observed in fig. 4.8. Again, the green nodes are nodes added by the AD pass, while the red nodes are part of the original expression. Note that we now treat the canonicalized expression as *original*, since it calculates the original value, not the derivative.

The implementation has an experimental Egg [112] based rewriting system that handles the former described optimization of the graph. It can be found on the `feature-egg-expr-rewrite` branch [86] in the optimizer's `ego` pass. It serializes parts of a region's result into Egg's `SymbolLang`. A `ConstantFold` analysis uses pre-defined rules to apply folding operations. The final extraction step currently just optimizes for smallest AST-size.
If any region argument, or control-flow-depending value is part of an expression, it gets serialized in a way, that the analysis can not change it. That way the e-graph can use such values, but is not allowed to transform them.
It was already shown, that e-graph can be utilized better with RVSDGs in Optir [101] and Eggcc [93]. This is also the reason that feature is not yet merged. We currently only optimize expressions (hence the name of the branch), not the whole graph. We found that the *typedness* of our graphs does not translate well to the standard usage of Egg. E.g. we have

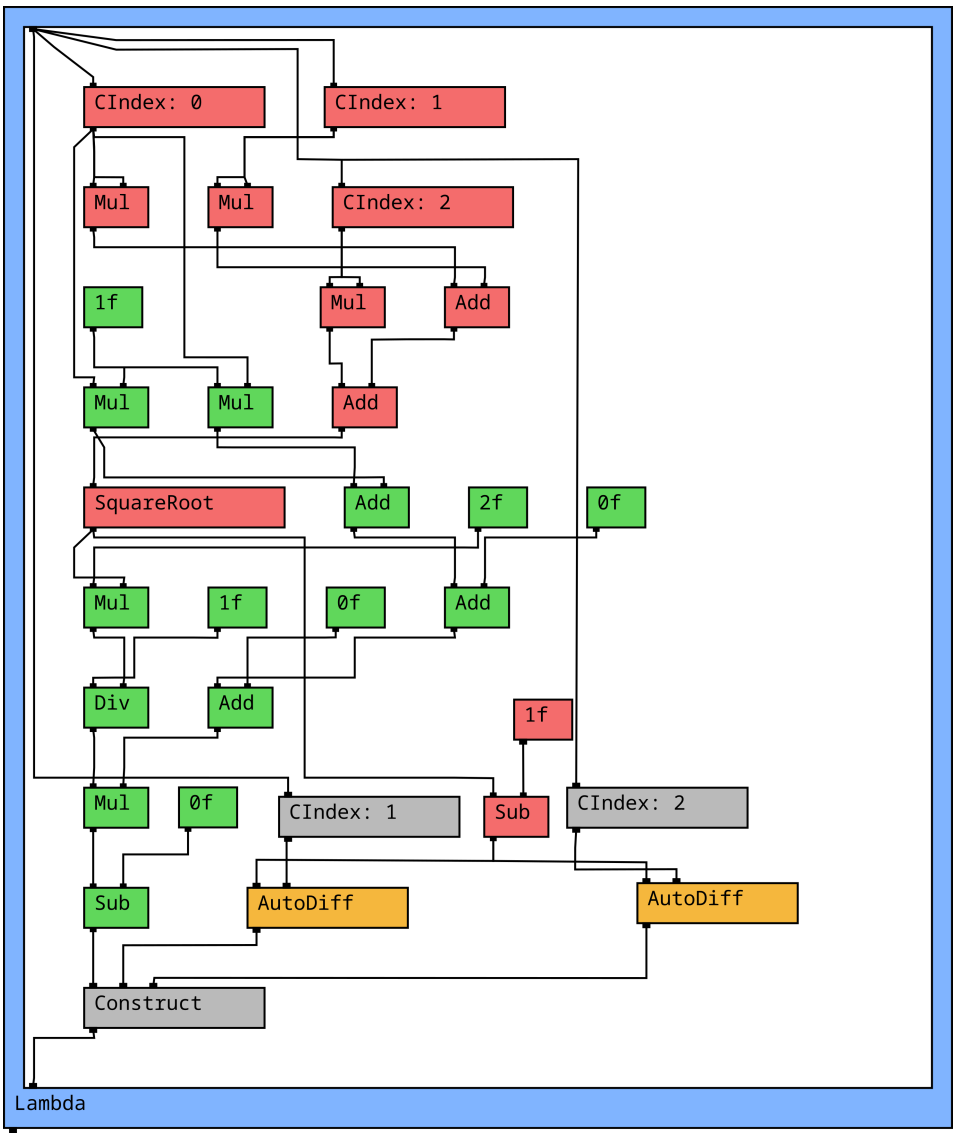


Figure 4.7.: Graph after first forward-mode application

no notion of a *neutral-element*. Reducing multiplication-by-one must be handeled with one rule for each data-type that supports multiplication, which does not scale well. A custom Egg-Language[8], and possibly rule description DSL has to be found, and implement, that supports context-informed rules.

4.6. Future work

There are several improvements that could be made to the implementation. Currently, only algebraic nodes can be differentiated. However, there also exist strategies for boolean differentials. Those could be used to increase the emitted differential quality. Similarly, additional domain specific types, like quaternions to represent rotations (instead of rotation matrices) would increase the differential quality as well, since the derivative is defined for those as well.

The rewrite logic is currently handwritten in the compiler. This process is error-prone and hard to debug. An improvement over this system would be a small, easy to read rewriting DSL.

[8] See Egg's language definition

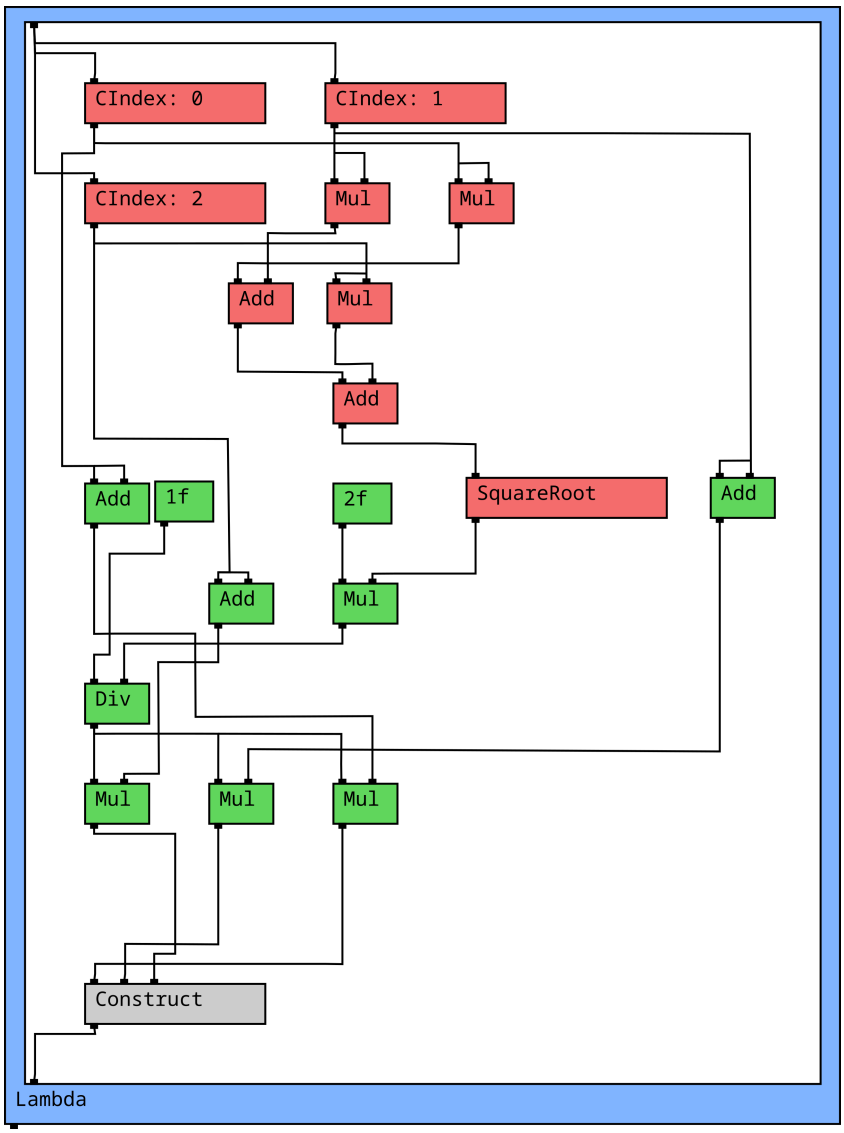


Figure 4.8.: Final graph after AD and optimization

Cranelift showed [26] that such a DSL can be used with equality saturation based optimizers like Egg [112] or Egglog [116] to implement a performant, easy to manage rewriting system. This is not to be confused with the Egg-based optimizing that is already implemented. The E-Graph functions as a common structure for two different tasks, once for AST-size-reduction, and once for graph-transformation.

Whenever an approximation uses a constant value, the value is currently set in the compiler's configuration. Consider the *approximation region* in fig. 3.1. Sizing that region correctly depends on the context that function is used in. In the best case scenario, the compiler would use sensible defaults (as it currently does), but expose a way for the user to set the approximation region, whenever those defaults fail. It is impractical to set such a region on a *per-operation* basis. My proposal is an abstract *sensitivity* value that is applied to an expression. This would act as a multiplier to any default approximation value that is encountered while canonicalizing. The rational is, that users do not necessarily understand what some abstract *approximation value* does to a given CSG. Usually they would want to in-/ decrease precision because the approximation fails somewhere. Therefore, just *dialing up* or *dialing down* approximation precision would lie on a similar level of understanding.

5. Evaluation

We evaluate the automatic differentiation under four different aspects:
First, we compare its results to the existing differentiation of Enzyme [90].
We then benchmark our compiler against Enzyme's Rust integration. The generated code is also compared in its runtime performance.
Afterwards we change the topic to the applications of the compiler feature by showing enabled usage patterns in the DSL that were not possible before. We also classify the usage pattern with regard to other SDF usage patterns, to illustrate the gained usability advantage.
In the end, we combine both topics into four use cases that showcase different applications of the derivative feature, ranging from shading of surfaces to procedural animation of objects.

5.1. Correctness

The implementation is tested for correctness by calculating the same differentiated distance function in Enzyme and Vola. Our implementation generates a web assembly (WASM) module from the Vola code, and a dynamic library from the Rust based, Enzyme differentiated code.
The differential tester then loads the WASM module into the wasmtime [1] runtime, and dynamically links the Rust library as well.
It generates fuzzing points (by default 1000) in 3D-Space. For each point, it is tested whether the Rust-Enzyme generated result is the same as the Vola generated result. The test run can be configured for a permitted deviation. Since both implementations generate different code, different floating-point impressions can occur. By default, we use 0.01. Refer to the vola-test repository[84] for implementation details. Note that the deviation is chosen relative to the tested scenario. The occurring error is fundamentally influenced by the function that is being compared.
There are two caveats that make that system less useful than anticipated. The first being, that Rust's Enzyme implementation can only differentiate unwrapped mathematical, expression-like code. While trying to build a *close-to-Vola* type system in Rust, several patterns were encountered, that crashed Enzyme. You can't use dynamic-dispatch / function pointers, so building a CSG-Tree in Rust is not possible. Using closures/higher-order-functions to represent operations and entities, as Vola does it, is also not possible. In the end, we settled to hand-unwrapping the distance function into simple mathematical expressions, which worked.
A second caveat is a miss compilation problem for debug builds with Enzyme. It was first encountered while testing differentiation of a union operator via Enzyme. The idea of the union operator is simple: Given two SDFs, take the smaller of both values. In practice, this is a *min* operation on both values. While testing both implementations, we found that there were

big differences for a specific sector in the 3D space, all other sectors had comparable results. At first, Vola's implementation was suspected to be at fault. We could reduce the issue to the union of two spheres, which can still be hand derived. It turned out that Enzyme was at fault. After further testing, we found, that Enzyme outputs the correct value in the release profile. The Enzyme user-group was asked about this issue [82]. An answer in the Enzyme-project's issue tracker [98] suggests that this might be related to an overall problem of Enzyme's handling of runtime activity, inlining and composite data types.
To have a second source of truth, an additional differential view was implemented in the test-renderer [83]. The differential view shows the difference of the AD generated derivative compared to the numerical derivative. The view does not capture subtle differences, but is good at signaling big, regional differences. This let us debug and correct a bug in the derivative creation of matrix-vector multiplication.

Finally, we could verify correctness for mathematical expressions using Enzyme and the differential view of the test renderer for a wide variety of use-cases. This includes vector-operations like cross and dot product, matrix operations, widely used scalar operations known from other shader languages and control-flow.

5.2. Benchmarking

We compare the performance of Vola's forward-mode to Enzyme's forward-mode. The benchmarking environment is described in section 3.3.2. The AD timings are taken for Vola's AD entrypoint `AutoDiff::dispatch_ad` and Enzyme's `LlvmCodegenBackend::autodiff`. In both cases, the differentiation happens in isolation to other compilation passes. We only measure the actual differentiation passes, as earlier and later optimization passes do not exclusively work on differential code in both cases.
We measure runtime CPU characteristics on a AMD Ryzen 9 5900x on (Arch) Linux. We isolate one full core for our benchmark process via the `nohz-full` kernel parameter. The runtime task is pinned to one core's threads via `taskset`[1].
The test-runtime introduces an overhead to the overall timing. Since the same runtime is used for both test cases (Vola & Enzyme generated code), the timings are comparable.
We profile both tools in four test cases. The derivative of a sphere, the hard and soft union of a sphere and a translated box, and finally a *blob* scenario that assembles a grid of spheres as well as four hand placed spheres at the coordinate system's center.
We repeat each test three times to verify comparable runtime. Table 5.1 presents the average of all measurements. For all benchmark data, see table A.3.

	Compile-time		AD compile		Runtime		Codesize	
	Enzyme	Vola	Enzyme	Vola	Enzyme	Vola	Enyme	Vola
Sphere	5.50s	207ms	2.50s	1.7ms	53μs	60μs	370.6kb	50.4kb
Union-Hard	5.62s	222ms	2.54s	1.8ms	57μs	60μs	371.1kb	67.6kb
Union-Soft	5.65s	225ms	2.54s	2.8ms	58μs	58μs	371.2kb	70.6kb
Blob	5.63s	225ms	2.61s	4.0ms	60μs	62μs	371.8kb	79.9kb

Table 5.1.: Enzyme/Vola benchmarks

The benchmarking shows a far shorter compile-time (for both, the overall compile time and

[1] Following this profiling guide: https://manuel.bernhardt.io/posts/2023-11-16-core-pinning/

the AD pass) for Vola in all four scenarios. The runtime is comparable, but favors the Enzyme generated code. The difference in runtime is probably rooted in the use of Cranelift to translate the Vola generated WASM module to x86.
The comparatively long compile time of Rust/Enzyme is explained by the use of LLVM's optimizing passes before and after differentiation. Both implementations need type information (and other auxiliary data) to successfully differentiate the given code. Given the simplest AD case, of just a sphere, 97% of the AD compile-time are spent in `LLVMRustOptimize`. The application of the forward-mode pass is less than 0.1%. For the slightly bigger *blob* scenario, 0.2% of the AD compile-time is spent in the forward-mode pass.
Enzyme probably scales better than Vola to more complicated, bigger scenarios. At which point this case needs to be determined.

5.3. Usability advantage

We implement differentiation as a first class concept in Vola's optimizing compiler. This grants the ability to differentiate any value of a program. This is a higher degree of freedom compared to general purpose shading languages like OpenGL shading language (GLSL) and high level shading language (HLSL). Those have intrinsic functions[2] to retrieve the derivative of an input to a shader. However, they do not possess the ability to build the derivative for *any* value. Slang, a newer shading language, posses the ability to declare differential functions. It needs manual declaration of those by hand though, and requires manual handling of the derivative values[3]. Differentiability is handled via an *interface* concept, which needs to be implemented for any differentiable value. This is appropriate for a more general language with AD capabilities. It can be argued that a DSL that does not need any additional work to make values differentiable is more convenient to use in that specific case.
Compared to Enzyme, our implementation allows using derivatives *within* derivatives, without explicit order handling [3]. We also handle derivative calculation as a fundamental part of our compiler toolchain. Anything that can be derived is handled. All other cases are signaled as an error to the user. The current Rust implementation, requires careful handling of derivative expressions and input/output data [3].

For our differential testing (cf. section 5.1) to work, we had to severely lower the abstraction level of the Rust implementation. The initial idea was, to build a CSG tree as a tree of dynamically dispatched trait-objects [23]. A simplified Vola CSG could have looked like fig. 5.1a. This does not permit a node of the tree to take the derivative of its children (which Vola allows), but lets us represent the fundamental *CSG-tree to DF* idea of Vola in Rust.
In practice, however, Enzyme can not build derivatives over trait objects. We also tried to represent the entities and operations via closures/higher-order functions and similar constructs, which did not work as well. In the end, we hand-unfolded each use-case in a simple sequence of operations, which are derivable (cf. fig. 5.1b).
This allows us to use the derivative not just for one use case, but any that involves derivatives. We can use the derivative of the SDF to calculate normal vectors (section 5.4.2), drive animations (section 5.4.3) based on an animation curve's derivative, shading (section 5.4.4) based on a second order derivative, or optimize a rendering algorithm (section 5.4.1).

[2] For instance *dxdy* in GLSL.
[3] see the *DifferentialPair*, and the overall AD user guide for Slang.

```rust
///Traits as "concept".
pub trait Sdf3d{
    fn eval_sdf3d(&self, at: Vec3) -> f32;
}

//Note "Box" is taken, use cuboid
struct Cuboid(Vec3);
struct Sphere(f32);
struct Union{
    left: Box<dyn Sdf3d>,
    right: Box<dyn Sdf3d>
}
struct Offset{
    offset: Vec3,
    sub: Box<dyn Sdf3d>
}

//.. implementation of Cuboid, Sphere, Union, Offset

#[autodiff(autodiff_sdf, Forward, Dual, Dual)]
fn sdf(at: Vec3, res: &mut f32) {
    let csg = Union{
        left: Box::new(Offset{
            offset: Vec3::X,
            sub: Box::new(Cuboid(Vec3::ONE))
        }),
        right: Box::new(Sphere(1.0))
    };
    *res = csg.eval_sdf3d(at);
}
```

(a) Trait object Rust implementation
See fig. A.2 for the whole implementation.

```rust
#[autodiff(autodiff_sdf, Forward, Dual, Dual)]
fn sdf(at: Vec3, res: &mut f32) {
    //sdf of a sphere at "at".
    let sphere = at.length() - 1.0;
    //Offset the box "at" parameter
    let box_offset = at - Vec3::ONE;
    //calculate the box's sdf with an
    //extent of Vec3::ONE
    //at "box_offset".
    let boxsdf = {
        let q = box_offset.abs() - Vec3::ONE;
        q.max(Vec3::ZERO).length()
            + q.max_element().min(0.0)
    };
    //union both parts
    let union = sphere.min(boxsdf);
    *res = union;
}
```

(b) Hand-unfolded derivable implementation

Figure 5.1.: Rust implementation of Vola's CSG-tree.

5.4. Example use-cases

We now evaluate the usage of the system in practical examples. We take a closer look at ray-tracing enhancement and then present several smaller examples that highlight more artistic use-cases for derivatives.

5.4.1. Segement tracing approximation

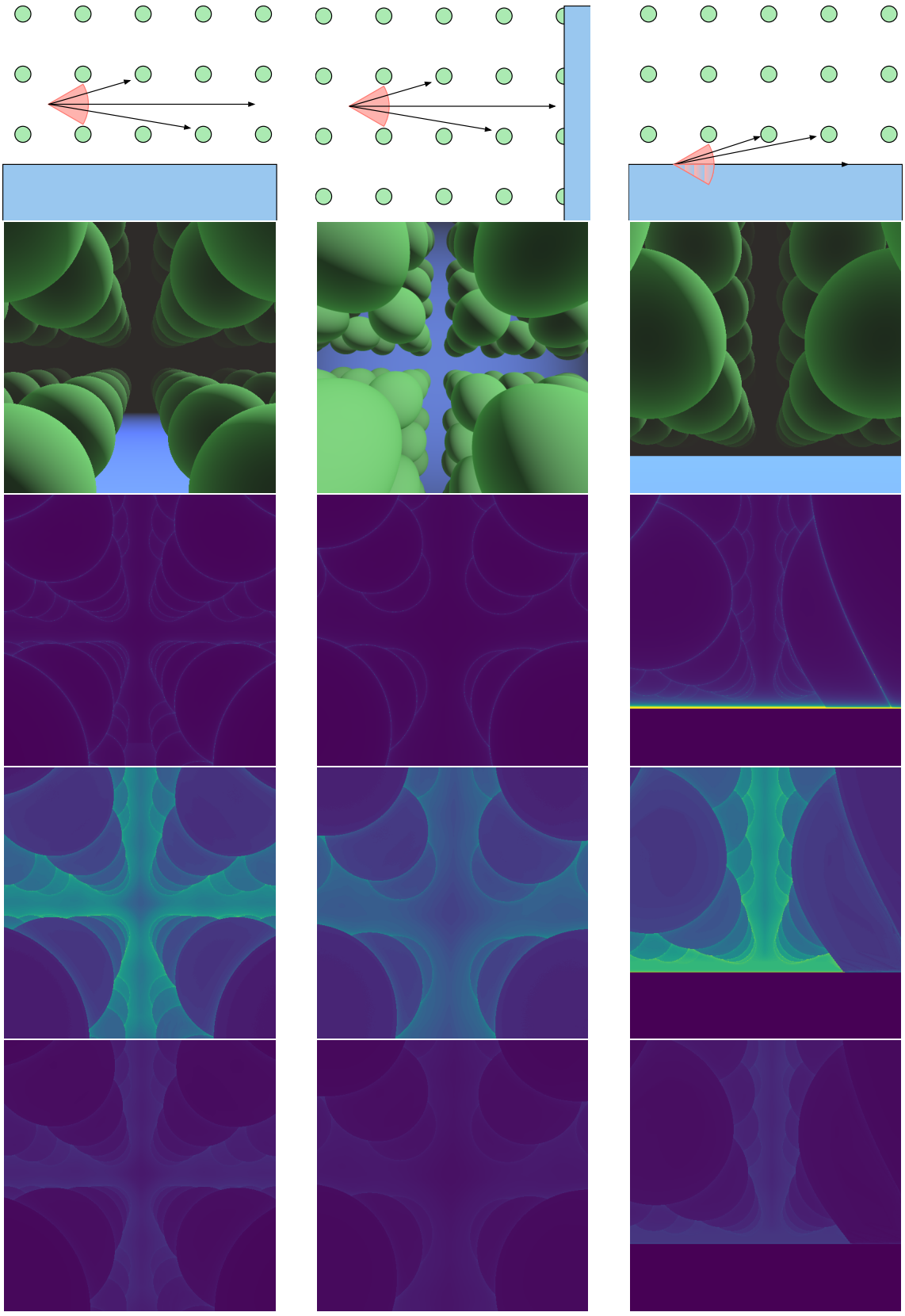
We can optimize the ray tracing algorithm used to render images from SDFs as described in section 2.3.4. We use a segment-tracing [32] approximation, that uses the local derivative of the SDF at the start and end of each segment, to approximate the local Lipschitz bound.
Furthermore, we compare the standard sphere-tracing based rendering to the segement-tracing approximation. We now have four combinations to render the SDF that can be compared:

- Sphere tracing with numeric gradient
- Sphere tracing with AD gradient
- Segment tracing with numeric gradient
- Segment tracing with AD gradient

The differentiation capability has no impact on the sphere-tracing algorithm. We therefore merge the analytic and numeric gradient into one case.
To get a better picture of the overall performance characteristics, we measure the three combinations in three scenarios. A *mixed* (fig. 5.2a) scenario, with far away intersections as well as

close intersections. A *gracing angle* (fig. 5.2b) scenario that includes the worst case for sphere-tracing: an infinite surface parallel ray. Finally, a third case, the *bound* (fig. 5.2c) scenario caps the traveled ray distance at an infinite plane.
The benchmark increases the rendering distance in 5 unit increments on the x-axis. For each distance, we measure the minimum and maximum iteration count per frame. We repeat this measurement across 20 frames and calculate the average. Each time, the benchmark renders at a resolution of 1440x1440 on an AMD RX6800-XT on Linux using the mesa RADV Vulkan driver.
Figure 5.3 shows the minimum and maximum field function evaluation count parameterized by the render distance on the x-axis for each scenario. It is to note that we abort root finding[4] after 8192 steps of the respective ray tracing algorithm, to prevent the graphics card from locking up.
Observe that the segment-tracing uses more field evaluations in the numerical case than sphere tracing most of the time. It needs six field evaluations to create the directional derivative, two times per step (needed for start and end of the segment), as well as an initial field evaluation. This comes down to 13 field evaluations per segment-tracing step in the numerical case, compared to one for sphere-tracing.
The analytical, AD based case takes three evaluations. Two gradient-field evaluations (for the start and end of the segment) as well as an initial field evaluation. That makes segment tracing more efficient than sphere tracing in all three scenarios.
Special attention should be given to the *gracing angle* case. Sphere tracing maintains the (expected) linear growth after the 150-unit mark[5]. In contrast, segment-tracing (in the analytical case) only grows till approximately 120-units. After that point it maintains about 1000 evaluations per frame maximum. The reason is that the gracing-angle case can reliability be overstepped: Whether a ray terminates, or not, is reliability found after those evaluations, including the never-terminating, parallel-to-the-surface case.
Interestingly, the minimum field evaluations per frame are similar for all three algorithms.

[4] Meaning the process of finding a *close to zero* value on the ray.
[5] This is when all *ever* intersection ray have terminated, and only the *never terminating* rays are still in flight. Therefore, the evaluation count linearly grows by the constant *iterations-per-unit* amount.



(a) Mixed (b) Bound (c) Gracing angle

Figure 5.2.: Segment tracing
The three rendering scenarios with field evaluation heat maps (darker is better). Top to bottom: Sideways scheme of the rendered scene, Rendering, Sphere-Tracing, Segment-Tracing Numeric, Segment-Tracing Analytic

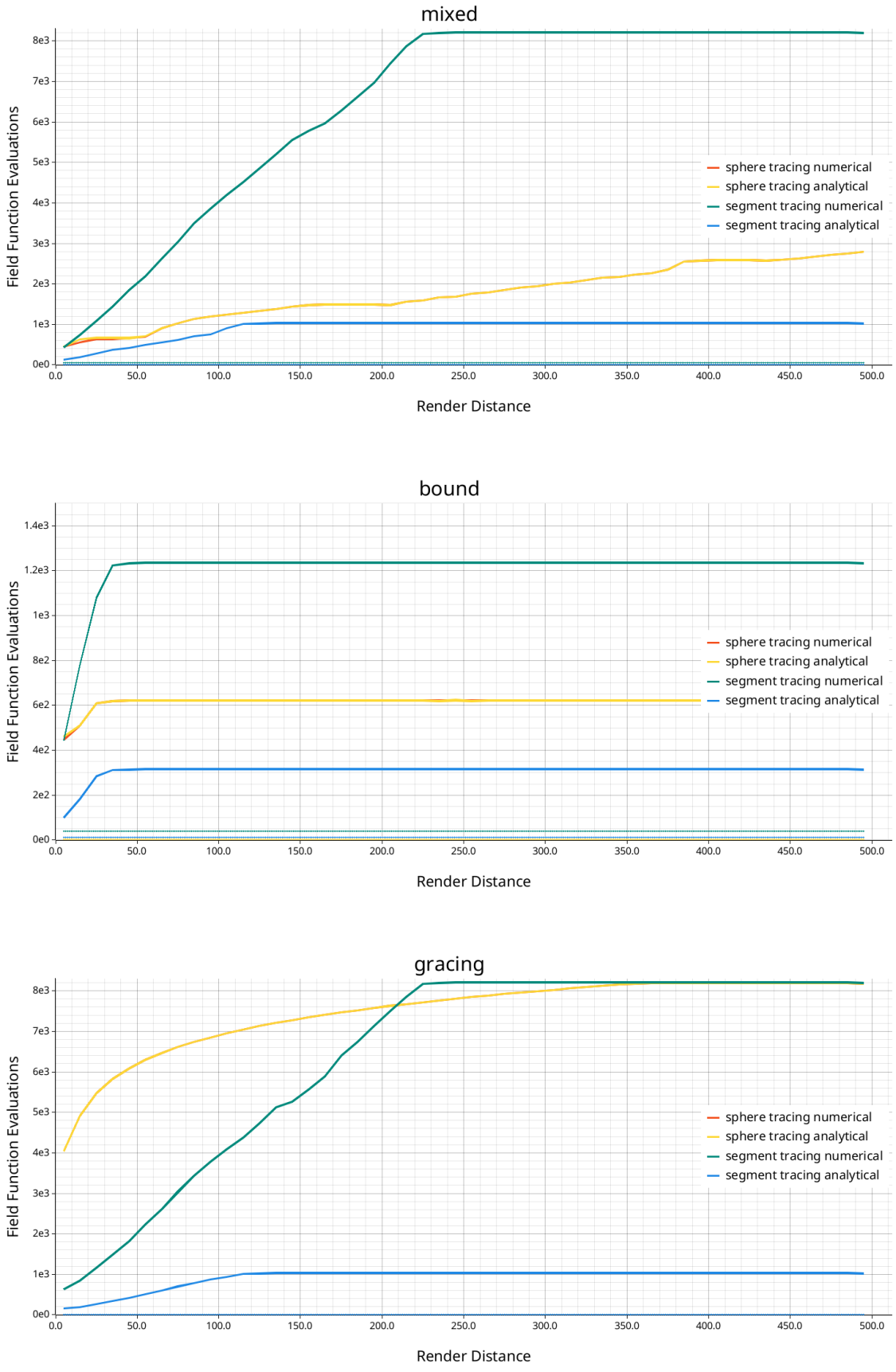


Figure 5.3.: Rendering scenarios: Evaluations at rendering distance

5.4.2. Normal vector calculation

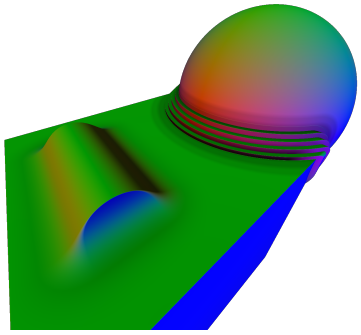
The normal vector of a surface plays a crucial role for shading surfaces in computer graphics, even for simple [8] shading models.
In SDF the normal vector at any given point in the field is equal to the normalized gradient, at that point. This allows us to formulate a *normal-vector-field* in Vola in fig. 5.4a. Applying the normal vector as the color of an object, one can observe the smooth interpolation over the surface in fig. 5.4b.
What makes this approach interesting, though, is the fact, that such a normal vector does not have to reside on a surface[6]. It is defined in the whole parameter space of the SDF. We could define density fields (by redefining the distance function's interpretation) as a special case of the SDF. Elaborate volume rendering algorithms, like [95], exploit that fact. The make use of that information to implement shaded volume rendering, without having to resort to expensive discretization of the volume data.

```
operation NormalToColor();
impl NormalToColor<sub> for Color3d(at){
    let grad = diff(eval sub.Sdf3d(at), at);

    abs(grad / [length(grad); 3])
}

define myugf(time: s){
    NormalToColor(){
    UnionStairs(0.2, 5.0 + sin(time)){
        Sphere(1.0)
    }{
        SmoothUnion(0.2){
            Trans3d([1.5, 1.0, 0.0]){
                Box([1.0; 3])
            }
        }{
            Trans3d([2.0, -0.0, 0.0]){
                Cylinder(0.2, 0.5)
            }
        }
    }}
}
```

(a) Coloring based on normal vector



(b) NormalToColor applied to object

Figure 5.4.: Normal vector coloring

5.4.3. Time derivative based animation emphasis

The ability to differentiate any function offers a tool to simplify artistic processes. Movement curves can be differentiated to enhance visual dynamics, such as emphasizing motion in a cartoon-like style. The first derivative of a curve can be used to orientation an object along the artist defined curve, while the second derivative is used to retrieve acceleration information, which can amplify the movement of an object by scaling it. The latter is also known in animation as *squash-and-stretch* [67].
This technique is already employed when utilizing splines as a means of describing paths. The derivative calculation for such paths is well explored. The AD implementation enhances that workflow by not constraining the artist to splines. It is offering that freedom for any animation path in space.

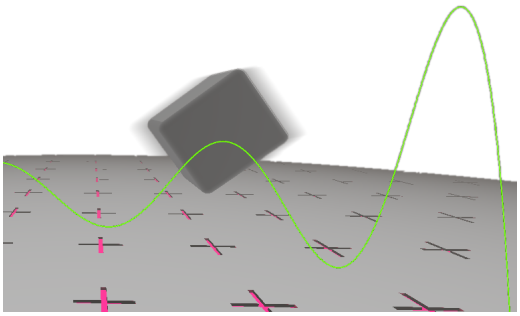
[6] Contrary to surface based model representations like polygons or non-uniform rational B-spline (NURBS).

In fig. 5.5 we use a Sinc function to represent a cartoon inspired *bouncing* motion. The `ObjectAlong`-operator takes care of animating the sub object along that curve, and squashing/stretching it according to the function's acceleration characteristics. Be aware that `ObjectAlong` does only retrieve the final position of the object, not its curve function. The artist is free to supply any curve, as long as it stays in 3D space.

```
define animated_cube(time: s){
    let anim_phase = (time % 5.0) * 2.0;
    let a = 3.141 * (2.0 * anim_phase - 1.0);
    let sinc = sin(a) / a;

    ObjectAlong(
        [anim_phase * 3.0, sinc * 5.0, 0.0],
        time
    ){
        Round(0.2){Box([1.0;3])}
    }
}
```

(a) Using the *ObjectAlong* operator



(b) Movement of the object

Figure 5.5.: Animation with time derivatives

5.4.4. Edge-sharpness based coloring

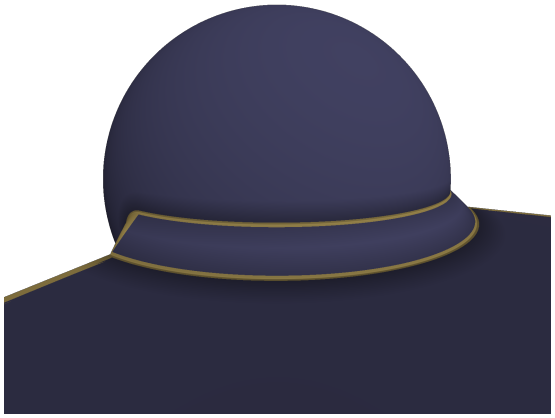
Sharpness of an object is useful in an artistic context. It can be used to drive effects like edge wear, shading, or simply finding edges on an object.
The first derivative of a signed distance function with respect to the evaluation location is the *change* of the distance field at the point (`ddist` in fig. 5.6a).
The derivative of this value is the *change of the change of the distance field*, or in other words, the curvature of the field. A higher value signals a greater change, also known as a sharper change. What constitutes an edge, becomes a threshold on that measurement.
Applied to our operator, we build the derivative of `ddist`. The result is a $3x3$ matrix. We fold this matrix into the *sharpness* measurement by taking the length of each column and then the maximum element of each length. The resulting `sharpness` value is the highest change in the distance field on each dimensional axis.
In this case, we simply switch the colors `cola` and `colb` based on the mentioned threshold value. Artistically, you could also interpolate instead, select a coloring function etc.

```
operation Sharpness(
    threshold: s,
    color_a: vec3,
    color_b: vec3
);
impl Sharpness<sub> for Color3d(at){
    let ddist = diff(eval sub.Sdf3d(at), at);
    let second = diff(ddist, at);
    let maxes = [
        length(second.x),
        length(second.y),
        length(second.z)
    ];
    let sharpness = max_element(maxes);

    let is_sharp = if sharpness > threshold{
        1.0
    }else{
        0.0
    };

    mix(color_a, color_b, [is_sharp; 3])
}
```

(a) Vola code for sharpness based coloring

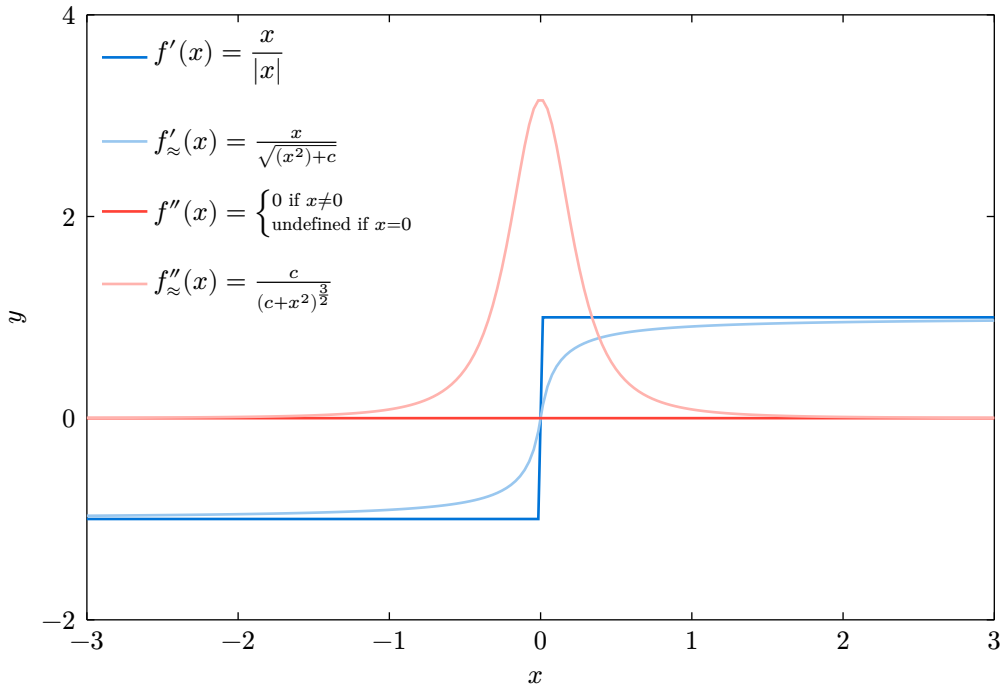


(b) Sharpness operator applied to a chamfer/bevel-union of a sphere and a box.

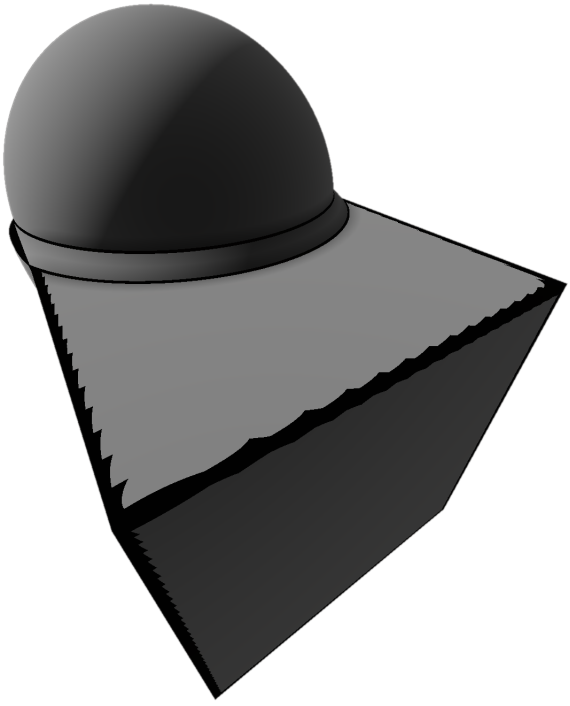
Figure 5.6.: Sharpness based coloring operator

5.5. Shortcomings

While working with higher order derivatives, a problem with the approximation of non-differential functions (cf. section 3.1.5) becomes apparent. An approximation that produces a good first derivative approximation can lead to bigger errors in higher derivatives.
In our case, we approximate $|x|$ with $\sqrt{x^2 + c}$. If we plot the first and second derivative of both functions in fig. 5.7a, we can see that the first derivative is a reasonable approximation, but the second derivative does not approximate the original function's second derivative anymore. This creates artifacts as seen in fig. 5.7b for the sharpness-based coloring shown in section 5.4.4. Notice the banding artifacts along the edges of the cube.
There are two solutions to the issue. One could choose the approximation more carefully to account for better approximations in higher derivatives. If the derivative count foreseeable and in a low range, this might be the fastest way of accounting for such errors. This approach does not scale to arbitrary derivative calculations. The better, more advanced procedure would be able to keep track of the original expression, and apply a fitting approximation depending on the derivative. In our example case, the derivative producer (described in section 4.3) would emit f' for the first derivative, but annotate the origin of the emitted expression as *being the first derivative of* $|x|$. Whenever an expression is differentiated once more, the producer finds that annotation, and can therefore apply a *special case* for the expression f''.
This implies a much greater bookkeeping overhead compared to the currently employed canonicalization approach, but is rewarded with better higher order derivatives.



(a) First and second derivative of $f(x) = |x|$.



(b) Artifacts on cube's edges

Figure 5.7.: Second order derivative artifacts.

6. Conclusion

Our application research suggests, that having access to derivatives enables a wide range of real-world use cases by providing the mathematical tools necessary for optimization, simulation, and modeling in various domains. We surveyed the current landscape of AD tools, and conducted a comparison among bigger, active implementations across multiple domains and abstraction levels. Special interest was taken in their specific design decisions and how common challenges, most notable non-differential code and memory-usage, are overcome. Using this information, we developed an implementation tailored to our domain, overcoming those challenges by leveraging assumptions we can make in our field. We found that our domain-specific approach allowed us to take shortcuts unavailable in more general frameworks, resulting in a comparatively small implementation. Using the compiler level for our implementation offered additional advantages, as it provided access to existing optimization strategies and control-flow and data-flow information available in the IR.

Our work resulted in a forward-mode differentiation implementation that is surprisingly robust against incorrect handling of derivative. We permit approximations that enhance its practical applicability when precision is not needed. However, these approximations proved inadequate, when scaling to higher-order derivatives, as detailed in section 5.5. A more comprehensive system, that uses auxiliary information provided by the compiler, could address these problems. It could leverage the additional information to emit better fitting approximations via special casing rules. Such a system might scalable better to higher derivatives but also introduces challenges in maintainability, as the complexity of the AD implementations increases with the number of supported operations.

Our exploration also demonstrated applications of derivatives within the context of SDFs, showcasing their potential in simple scenarios. Further applications outlined in section 2.3 suggest more sophisticated applications that remain open to further investigation. This work highlights both the potential of domain-specific AD implementations and the ongoing challenges and opportunities in extending their capabilities.

Referring back to our initial motivation we could show, that differentiating compilers are an efficient way of combining AD and DF research, while also preserving the mathematical properties and precision of such fields until evaluation. In the future such a framework can proof useful in other, not yet explored applications in the field.

6.1. Future work

While working on, and with the AD implementation, several ideas and aspects came to light, that we did not consider before, or did not have the time to implement.

6.1.1. Backward mode

We do not currently have a backward-mode implementation. Backward mode introduces additional complexity, to both, the activity analysis and derivative generation. Unlike forward-mode differentiation, which is straightforward for many scenarios, reverse-mode requires careful consideration of computational dependencies to ensure efficiency. Especially, reuse of existing values becomes important not just for a fast compilation, but also short runtime of the generated code (cf. section 2.4.4).
We also currently lack a well-defined heuristic for determining when to apply forward or backward mode. Most tools leave this decision to the user, relying on domain-specific knowledge or trial-and-error to choose between forward and reverse modes. Developing a robust heuristic or adaptive mechanism to automate this decision could significantly enhance usability for the non-expert user of the DSL, and the AD feature.
Nevertheless, backward mode would likely be crucial for any application involving machine learning or gradient descent. Its ability to handle large-scale problems efficiently makes it a foundational requirement for modern computational frameworks in these domains.

6.1.2. Better approximation framework

AD in compilers can benefit from maintaining more context about the nodes in the computational graph. By tracking richer information about the relationships of nodes (cf. section 5.5), the compiler can make more informed decisions about derivative computations. This enables better handling of special cases, a strategy already used for some operations (section 4.3), and which could be expanded significantly with enhanced context-awareness.
Special-casing allows the compiler to optimize patterns, such as simplifying certain derivatives (based on the enhanced context information) or recognizing higher order derivatives. With a more informed framework, these optimizations could extend to more complex cases, improving performance and flexibility. However, increasing reliance on special cases introduces maintenance challenges, as the approach's strength depends on the number of cases it supports.

6.1.3. DSL Rewriting

Rewriting systems offer a powerful way for expressing transformations on program representations. A domain-specific language tailored to rewriting, inspired by systems like Cranelift's ISLE [26], MLIR's PDLL [74], and Egg/Egglog [112, 116], could enhance the flexibility of derivative creation. This way we would reformulate the derivative creation as a term-rewriting problem. This DSL would enable the specification of derivative rules in a declarative manner. With conditions attached to rewrite rules, the system could describe derivative transformations blindly while relying on an optimizer to select the most efficient paths. Paired with *enhanced context* idea from section 5.5 such a system would reduce the maintenance burden (compared to handwritten transformations), and potentially increase the generated derivative's quality.

6.1.4. Differential types

The current type-system handles scalars, vectors, and matrices well, it lacks native support for higher-level types like complex numbers, quaternions, and rotors though. These types have well-defined derivatives and are a good fit for accurately representing operations such as rotations. If we try to represent quaternions as four-component vectors, differentiating at the moment will yield incorrect results. In order to correctly handle such cases, and improving the fitness of the language itself, we would have to introduce such concepts as first-class types to the language and its compiler.
We support matrices, and therefore rotation matrices, representing rotations at all is therefore already possible. Since the DSL is focused on representing objects in space, it is reasonable to introduce such specialized types. The same idea also applies to other mathematical constructs with defined derivatives.

6.1.5. Interval arithmetic

Similar projects in the SDF domain (e.g. Libfive [58], MPR [60] and Fidget [81]) enhance their capabilities by adding interval arithmetic to their evaluation options. It is adjacent to standard evaluation and derivative creation, since it provides yet another perspective on interpreting a given function. The standard evaluation returns the result of a function. AD returns the change of that function. Interval evaluation would return the upper and lower bound of that function's results in a given interval. This is useful whenever an understanding of a function's behavior over a region is needed, rather than at a discrete point.
That capability opens up even more analysis options for the represented DF. It can aid in polygonization [108] and image generation through ray-tracing [65].
The combined use of AD and interval arithmetic allows the analysis of a distance-functions behavior in an interval, which is the basis for the linear forward-inclusion-functions [4] mentioned in section 2.3.4.

Bibliography

[1] Bytecode Alliance. "Wasmtime: A standalone runtime for WebAssembly". In: (2024). URL: https://github.com/bytecodealliance/wasmtime.

[2] Joel A E Andersson et al. "CasADi – A software framework for nonlinear optimization and optimal control". In: *Mathematical Programming Computation* 11.1 (2019), pp. 1–36. DOI: 10.1007/s12532-018-0139-4.

[3] Enzyme Rust Authors. In: (2024). URL: https://web.archive.org/web/20241113102728/https://enzyme.mit.edu/index.fcgi/rust/print.html.

[4] Melike Aydinlilar and Cédric Zanni. "Forward inclusion functions for ray-tracing implicit surfaces". In: *Computers and Graphics* 114 (June 2023), pp. 190–200. DOI: 10.1016/j.cag.2023.05.026. URL: https://inria.hal.science/hal-04129922.

[5] Csaba Bálint and Gábor Valasek. "Accelerating Sphere Tracing". In: *Eurographics*. 2018. URL: https://api.semanticscholar.org/CorpusID:51958599.

[6] Atilim Gunes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. "DiffSharp: Automatic Differentiation Library". In: *CoRR* abs/1511.07727 (2015). arXiv: 1511.07727. URL: http://arxiv.org/abs/1511.07727.

[7] Atilim Gunes Baydin et al. "Automatic Differentiation in Machine Learning: a Survey". In: *Journal of Machine Learning Research* 18.153 (2018), pp. 1–43. URL: http://jmlr.org/papers/v18/17-468.html.

[8] Gary Bishop and David M. Weimer. "Fast Phong shading". In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 103–106. ISSN: 0097-8930. DOI: 10.1145/15886.15897. URL: https://doi.org/10.1145/15886.15897.

[9] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[10] Nick Brown et al. "xDSL: A common compiler ecosystem for domain specific languages". English. In: Supercomputing 2023, SC23 ; Conference date: 12-11-2023 Through 17-11-2023. Nov. 2022. URL: https://sc23.supercomputing.org/.

[11] Dan Cascaval et al. "Differentiable 3D CAD Programs for Bidirectional Editing". In: *CoRR* abs/2110.01182 (2021). arXiv: 2110.01182. URL: https://arxiv.org/abs/2110.01182.

[12] Christopher Choy, JunYoung Gwak, and Silvio Savarese. "4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 3075–3084.

[13] Sebastian Christodoulou and Uwe Naumann. *Differentiable Programming: Efficient Smoothing of Control-Flow-Induced Discontinuities*. 2023. arXiv: 2305.06692 [cs.PL]. URL: https://arxiv.org/abs/2305.06692.

[14] William Clifford. "Preliminary Sketch of Biquaternions". In: *Proceedings of the London Mathematical Society* s1-4.1 (1871), pp. 381–395. DOI: https://doi.org/10.1112/plms/s1-4.1.381. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s1-4.1.381. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s1-4.1.381.

[15] Thomas F. Coleman and Arun Verma. "The Efficient Computation of Sparse Jacobian Matrices Using Automatic Differentiation". In: *SIAM Journal on Scientific Computing* 19.4 (1998), pp. 1210–1233. DOI: 10.1137/S1064827595295349. eprint: https://doi.org/10.1137/S1064827595295349. URL: https://doi.org/10.1137/S1064827595295349.

[16] Calvin Rose & Janet contributors. "Janet Language". In: (2024). URL: https://janet-lang.org/.

[17] Blake Courter. "Unit Gradient Fields: SDFs, UGFs, and their friends". In: (2023). URL: https://www.blakecourter.com/2023/05/18/field-notation.html.

[18] Blake Courter. "Unit Gradient Fields: What do we mean by "offset"?" In: (2023). URL: https://www.blakecourter.com/2023/05/05/what-is-offset.html.

[19] John Dannenhoffer and Robert Haimes. "Design Sensitivity Calculations Directly on CAD-based Geometry". In: *53rd AIAA Aerospace Sciences Meeting*. DOI: 10.2514/6.2015-1370. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2015-1370. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2015-1370.

[20] Benjamin Dauvergne and Laurent Hascoët. "The Data-Flow Equations of Checkpointing in Reverse Automatic Differentiation". In: *Computational Science – ICCS 2006*. Ed. by Vassil N. Alexandrov et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 566–573. ISBN: 978-3-540-34386-8.

[21] Aesara Developers. "Aesara". In: (2024). URL: https://github.com/aesara-devs/aesara.

[22] PyMC Developers. "PyTensor". In: (2024). URL: https://github.com/pymc-devs/pytensor.

[23] The Rust Project Developers. "The Rust Language Reference". In: (2024). URL: https://doc.rust-lang.org/reference/types/trait-object.html.

[24] Tao Du et al. "InverseCSG: automatic conversion of 3D models to CSG trees". In: *ACM Trans. Graph.* 37.6 (Dec. 2018). ISSN: 0730-0301. DOI: 10.1145/3272127.3275006. URL: https://doi.org/10.1145/3272127.3275006.

[25] Conal Elliott. "Beautiful differentiation". In: *International Conference on Functional Programming (ICFP)*. 2009. URL: http://conal.net/papers/beautiful-differentiation.

[26] Chris Fallin. "Cranelift's Instruction Selector DSL, ISLE: Term-Rewriting Made Practical". In: (Jan. 2023). URL: https://web.archive.org/web/20241005052455/https://cfallin.org/blog/2023/01/20/cranelift-isle/.

[27] Christèle Faure and Uwe Naumann. "Minimizing the Tape Size". In: *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Ed. by George Corliss et al. New York, NY: Springer New York, 2002, pp. 293–298. ISBN: 978-1-4613-0075-5. DOI: 10.1007/978-1-4613-0075-5_34. URL: https://doi.org/10.1007/978-1-4613-0075-5_34.

[28] Pierre-Alain Fayolle and Markus Friedrich. "A Survey of Methods for Converting Unstructured Data to CSG Models". In: *Comput. Aided Des.* 168 (2023), p. 103655. URL: https://api.semanticscholar.org/CorpusID:258436689.

[29] Jeffrey A. Fike and Juan J. Alonso. "Automatic Differentiation Through the Use of Hyper-Dual Numbers for Second Derivatives". In: *Recent Advances in Algorithmic Differentiation*. Ed. by Shaun Forth et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 163–173. ISBN: 978-3-642-30023-3.

[30] Shaun A. Forth. "An efficient overloaded implementation of forward mode automatic differentiation in MATLAB". In: *ACM Trans. Math. Softw.* 32.2 (June 2006), pp. 195–222. ISSN: 0098-3500. DOI: 10.1145/1141885.1141888. URL: https://doi.org/10.1145/1141885.1141888.

[31] Markus Friedrich et al. "CSG Tree Extraction from 3D Point Clouds and Meshes Using a Hybrid Approach". In: *Computer Vision, Imaging and Computer Graphics Theory and Applications*. Ed. by Kadi Bouatouch et al. Cham: Springer International Publishing, 2022, pp. 53–79. ISBN: 978-3-030-94893-1.

[32] Eric Galin et al. "Segment Tracing Using Local Lipschitz Bounds". In: *Computer Graphics Forum* 39.2 (2020), pp. 545–554. DOI: https://doi.org/10.1111/cgf.13951. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13951. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13951.

[33] Assefaw H. Gebremedhin and Andrea Walther. "An introduction to algorithmic differentiation". In: *WIREs Data Mining and Knowledge Discovery* 10.1 (2020), e1334. DOI: https://doi.org/10.1002/widm.1334. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1334. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1334.

[34] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. "What Color Is Your Jacobian? Graph Coloring for Computing Derivatives". In: *SIAM Rev.* 47 (2005), pp. 629–705. URL: https://api.semanticscholar.org/CorpusID:8849423.

[35] Kyriakos C. Giannakoglou and Dimitrios I. Papadimitriou. "Adjoint Methods for Shape Optimization". In: *Optimization and Computational Fluid Dynamics*. Ed. by Dominique Thévenin and Gábor Janiga. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 79–108. ISBN: 978-3-540-72153-6. DOI: 10.1007/978-3-540-72153-6_4. URL: https://doi.org/10.1007/978-3-540-72153-6_4.

[36] Andreas Griewank. "Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation". In: *Optimization Methods and Software* 1.1 (1992), pp. 35–54. DOI: 10.1080/10556789208805505. eprint: https://doi.org/10.1080/10556789208805505. URL: https://doi.org/10.1080/10556789208805505.

[37] Andreas Griewank. "Who Invented the Reverse Mode of Differentiation". In: 2012. URL: https://api.semanticscholar.org/CorpusID:15568746.

[38] Andreas Griewank and Andrea Walther. "Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation". In: *ACM Trans. Math. Softw.* 26.1 (Mar. 2000), p. 19. ISSN: 0098-3500. DOI: 10.1145/347837.347846. URL: https://doi.org/10.1145/347837.347846.

[39] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Second. USA: Society for Industrial and Applied Mathematics, 2008. ISBN: 0898716594.

[40] Hao-Xiang Guo et al. "Implicit Conversion of Manifold B-Rep Solids by Neural Halfspace Representation". In: *ACM Trans. Graph.* 41.6 (Nov. 2022). ISSN: 0730-0301. DOI: 10.1145/3550454.3555502. URL: https://doi.org/10.1145/3550454.3555502.

[41] Milad Hakimi and Arrvindh Shriraman. “TapeFlow: Streaming Gradient Tapes in Automatic Differentiation”. In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2024, pp. 81–92. DOI: 10.1109/CGO57630.2024.10444805.

[42] Ralf Hannemann et al. “Discrete first- and second-order adjoints and automatic differentiation for the sensitivity analysis of dynamic models”. In: *Procedia Computer Science* 1.1 (2010). ICCS 2010, pp. 297–305. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2010.04.033. URL: https://www.sciencedirect.com/science/article/pii/S1877050910000347.

[43] John C. Hart. “Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces”. In: *The Visual Computer* 12.10 (1996), pp. 527–545. ISSN: 1432-2315. DOI: 10.1007/s003710050084. URL: https://doi.org/10.1007/s003710050084.

[44] Laurent Hascoet and Valérie Pascual. “The Tapenade automatic differentiation tool: Principles, model, and specification”. In: *ACM Trans. Math. Softw.* 39.3 (May 2013). ISSN: 0098-3500. DOI: 10.1145/2450153.2450158. URL: https://doi.org/10.1145/2450153.2450158.

[45] Ian Henry. “Bauble.Studio”. In: (). URL: https://github.com/ianthehenry/bauble.studio.

[46] Robin J. Hogan. “Fast Reverse-Mode Automatic Differentiation using Expression Templates in C++”. In: *ACM Trans. Math. Softw.* 40.4 (July 2014). ISSN: 0098-3500. DOI: 10.1145/2560359. URL: https://doi.org/10.1145/2560359.

[47] J.H. Hubbard and B.B. Hubbard. *Vector Calculus, Linear Algebra, and Differential Forms: A Unified Approach*. Appendix A4. Prentice Hall, 2002. ISBN: 9780130414083. URL: https://books.google.de/books?id=bW92QgAACAAJ.

[48] Jan Hückelheim et al. “Automatic Differentiation for Adjoint Stencil Loops”. In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP ’19. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337906. URL: https://doi.org/10.1145/3337821.3337906.

[49] nTopology Inc. “nTop”. In: (2024). URL: https://www.ntop.com/.

[50] nTopology Inc. “nTop 5 is here: Expanding the implicit ecosystem, boosting performance and precision”. In: (2024). URL: https://www.ntop.com/resources/product-updates/ntop-5/.

[51] Michael Innes. “Don’t Unroll Adjoint: Differentiating SSA-Form Programs”. In: *CoRR* abs/1810.07951 (2018). arXiv: 1810.07951. URL: http://arxiv.org/abs/1810.07951.

[52] Masao Iri and Koichi Kubota. “Automatic differentiation: introduction, history and rounding error estimationAutomatic Differentiation: Introduction, History and Rounding Error Estimation”. In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos M. Pardalos. Boston, MA: Springer US, 2001, pp. 97–102. ISBN: 978-0-306-48332-5. DOI: 10.1007/0-306-48332-7_19. URL: https://doi.org/10.1007/0-306-48332-7_19.

[53] *Cache-Aware and Roofline-Ideal Automatic Differentiation*. Vol. Day 1 Tue, October 26, 2021. SPE Reservoir Simulation Conference. Oct. 2021, D011S012R004. DOI: 10.2118/203933-MS. eprint: https://onepetro.org/spersc/proceedings-pdf/21RSC/1-21RSC/D011S012R004/2508238/spe-203933-ms.pdf. URL: https://doi.org/10.2118/203933-MS.

[54] LunarG John Kessenich. “SPIR-V: A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels”. In: (2015). URL: https://registry.khronos.org/SPIR-V/papers/WhitePaper.html.

[55] Randi Rost John Kessenich Dave Baldwin. "The OpenGL® Shading Language". In: (June 2014). URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.40.pdf.

[56] M.W. Jones, J.A. Baerentzen, and M. Sramek. "3D distance fields: a survey of techniques and applications". In: *IEEE Transactions on Visualization and Computer Graphics* 12.4 (2006), pp. 581–599. DOI: 10.1109/TVCG.2006.56.

[57] Tim Kaler et al. "PARAD: A Work-Efficient Parallel Algorithm for Reverse-Mode Automatic Differentiation". In: *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pp. 144–158. DOI: 10.1137/1.9781611976489.11. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611976489.11. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976489.11.

[58] Matt Keeter. "libfive". In: (2024). URL: https://libfive.com/about.

[59] Matt Keeter. "Lineage of CBA CAD Tools". In: (2017). URL: https://www.mattkeeter.com/blog/2017-01-09-lineage/.

[60] Matthew J. Keeter. "Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 39.4 (July 2020).

[61] Benjamin Keinert et al. "Enhanced Sphere Tracing". In: *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. Ed. by Andrea Giachetti. The Eurographics Association, 2014. ISBN: 978-3-905674-72-9. DOI: /10.2312/stag.20141233.

[62] Kamil A. Khan and Paul I. Barton. "A vector forward mode of automatic differentiation for generalized derivative evaluation". In: *Optimization Methods and Software* 30.6 (2015), pp. 1185–1212. DOI: 10.1080/10556788.2015.1025400. eprint: https://doi.org/10.1080/10556788.2015.1025400. URL: https://doi.org/10.1080/10556788.2015.1025400.

[63] Marius Kintel. "OpenSCAD: The Programmers Solid 3D CAD Modeller". In: (2024). URL: https://openscad.org/about.html#underlying-technology.

[64] Elisabeth Kluth. "Raumkünstler". In: (2023). URL: https://github.com/elisabeth96/Raumkuenstler.

[65] Aaron Knoll et al. "Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic". In: *Computer Graphics Forum* 28 (2009). URL: https://api.semanticscholar.org/CorpusID:696144.

[66] Johann Korndörfer et al. *HG_SDF A glsl library for building signed distance functions*. https://mercury.sexy/hg_sdf. Accessed: 2021-07-28. 2015.

[67] Ji-yong Kwon and In-Kwon Lee. "The Squash-and-Stretch Stylization for Character Motions". In: *IEEE Transactions on Visualization and Computer Graphics* 18.3 (2012), pp. 488–500. DOI: 10.1109/TVCG.2011.48.

[68] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based Python JIT compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[69] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[70] Soeren Laue, Matthias Mitterreiter, and Joachim Giesen. "Computing Higher Order Derivatives of Matrix and Tensor Expressions". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/0a1bf96b7165e962e90cb14648c9462d-Paper.pdf.

[71] Allan M. M. Leal. *autodiff, a modern, fast and expressive C++ library for automatic differentiation*. https://autodiff.github.io. 2018. URL: https://autodiff.github.io.

[72] Seppo Linnainmaa. *Algoritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden*. fin. 2020. URL: URN:NBN:fi:hulib-202006173019;http://hdl.handle.net/10138/316565.

[73] Hsueh-Ti Derek Liu et al. "A Unified Differentiable Boolean Operator with Fuzzy Logic". In: *International Conference on Computer Graphics and Interactive Techniques*. 2024. URL: https://api.semanticscholar.org/CorpusID:271182710.

[74] LLVM. "PDLL - PDL Language". In: (2024). URL: https://mlir.llvm.org/docs/PDLL/#why-build-a-new-language-instead-of-improving-tablegen-drr.

[75] Julia Longtin. "ImplicitCAD". In: (2024). URL: https://implicitcad.org/.

[76] Honglin Luo, Xianfu Wang, and Brett Lukens. "Variational Analysis on the Signed Distance Functions". In: *Journal of Optimization Theory and Applications* 180.3 (2019), pp. 751–774. ISSN: 1573-2878. DOI: 10.1007/s10957-018-1414-2. URL: https://doi.org/10.1007/s10957-018-1414-2.

[77] Dan MacKinlay. "Automatic differentiation". In: (2023). URL: https://danmackinlay.name/notebook/autodiff.html.

[78] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. "Autograd: Effortless gradients in numpy". In: *ICML 2015 AutoML workshop*. Vol. 238. 5. 2015.

[79] Charles C. Margossian. "A review of automatic differentiation and its efficient implementation". In: *WIREs Data Mining and Knowledge Discovery* 9.4 (Mar. 2019). ISSN: 1942-4795. DOI: 10.1002/widm.1305. URL: http://dx.doi.org/10.1002/WIDM.1305.

[80] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[81] Bruce Mitchener Matt Keeter. "Fidget". In: (2024). URL: https://github.com/mkeeter/fidget.

[82] Tendsin Mende. "Enzyme Bug Report". In: (2024). URL: https://web.archive.org/web/20241107124616/https://groups.google.com/g/enzyme-dev/c/wd3fu5aIcJE.

[83] Tendsin Mende. "Vola Test Renderer". In: (2024). URL: https://gitlab.com/tendsinmende/vola-sdf-renderer.

[84] Tendsin Mende. "Vola-Enzyme differential testing". In: (2024). URL: https://gitlab.com/tendsinmende/vola-tests.

[85] Tendsin Mende. "Vola: Volume Language". In: (2024). URL: https://gitlab.com/tendsinmende/vola.

[86] Tendsin Mende. "Vola: Volume Language, Egg based expression rewriting". In: (2024). URL: https://gitlab.com/tendsinmende/vola/-/tree/feature-egg-expr-rewrite.

[87] Bart van Merrienboer, Dan Moldovan, and Alexander Wiltschko. "Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/748d6b6ed8e13f857ceaa6cfbdca14b8-Paper.pdf.

[88] Raphael Mosaner et al. "Machine-Learning-Based Self-Optimizing Compiler Heuristics". In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. MPLR'22. Brussels, Belgium: Association for Computing Machinery, 2022, p. 98. ISBN: 9781450396967. DOI: 10.1145/3546918.3546921. URL: https://doi.org/10.1145/3546918.3546921.

[89] William Moses and Valentin Churavy. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 12472–12485. URL: https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf.

[90] William S. Moses et al. "Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476165. URL: https://doi.org/10.1145/3458817.3476165.

[91] Uwe Naumann. "Call Tree Reversal is NP-Complete". In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 13–22. ISBN: 978-3-540-68942-3.

[92] Uwe Naumann and Jan Riehme. "A differentiation-enabled Fortran 95 compiler". In: *ACM Trans. Math. Softw.* 31.4 (Dec. 2005), p. 458. ISSN: 0098-3500. DOI: 10.1145/1114268.1114270. URL: https://doi.org/10.1145/1114268.1114270.

[93] Alex Fischman Oliver Flatt Anjali Pal. "Eggcc". In: (). URL: https://github.com/egraphs-good/eggcc.

[94] Mai Jacob Peng and Christophe Dubach. "LAGrad: Statically Optimized Differentiable Programming in MLIR". In: *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. CC 2023. Montréal, QC, Canada: Association for Computing Machinery, 2023, p. 228. ISBN: 9798400700880. DOI: 10.1145/3578360.3580259. URL: https://doi.org/10.1145/3578360.3580259.

[95] M. Piochowiak, T. Rapp, and C. Dachsbacher. "Stochastic Volume Rendering of Multi-Phase SPH Data". In: *Computer Graphics Forum* 40.1 (2021), pp. 97–109. DOI: https://doi.org/10.1111/cgf.14121. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14121. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14121.

[96] Nico Reissmann et al. "RVSDG: An Intermediate Representation for Optimizing Compilers". In: *ACM Trans. Embed. Comput. Syst.* 19.6 (Dec. 2020). ISSN: 1539-9087. DOI: 10.1145/3391902. URL: https://doi.org/10.1145/3391902.

[97] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. "Forward-Mode Automatic Differentiation in Julia". In: *CoRR* abs/1607.07892 (2016). arXiv: 1607.07892. URL: http://arxiv.org/abs/1607.07892.

[98] Sam. "incorrect derivative of function that returns struct". In: (2024). URL: https://web.archive.org/web/20241127141146/https://github.com/EnzymeAD/Enzyme/issues/1894.

[99] Paul D Sampson. "Fitting conic sections to "very scattered" data: An iterative refinement of the bookstein algorithm". In: *Computer Graphics and Image Processing* 18.1 (1982), pp. 97–108. ISSN: 0146-664X. DOI: https://doi.org/10.1016/0146-664X(82)90101-0. URL: https://www.sciencedirect.com/science/article/pii/0146664X82901010.

[100] Gopal Sharma et al. "CSGNet: Neural Shape Parser for Constructive Solid Geometry". In: *CoRR* abs/1712.08290 (2017). arXiv: 1712.08290. URL: http://arxiv.org/abs/1712.08290.

[101] Jamey Sharp. "optir: "optimizing intermediate representation"". In: (2024). URL: https://github.com/jameysharp/optir/.

[102] Jag Mohan Singh and P. J. Narayanan. "Real-Time Ray Tracing of Implicit Surfaces on the GPU". In: *IEEE Transactions on Visualization and Computer Graphics* 16.2 (2010), pp. 261–272. DOI: 10.1109/TVCG.2009.41.

[103] Jeffrey Mark Siskind and Barak A. Pearlmutter. "Divide-and-conquer checkpointing for arbitrary programs with no user annotation". In: *Optimization Methods and Software* 33.4-6 (2018), pp. 1288–1330. DOI: 10.1080/10556788.2018.1459621. eprint: https://doi.org/10.1080/10556788.2018.1459621. URL: https://doi.org/10.1080/10556788.2018.1459621.

[104] Ole Stauning. "Obtaining 2nd Order Derivates Using Mixed Forward- and Backward Automatic Differentiation Strategy for use in Interval Optimization". English. In: *Obtaining 2nd Order Derivates Using Mixed Forward- and Backward Automatic Differentiation Strategy for use in Interval Optimization*. INTERVAL '96 ; Conference date: 01-01-1996. 1997.

[105] Auto Differentiation Dev Team et al. *auto-differentiation/xad: v1.6.0*. Version v1.6.0. July 2024. DOI: 10.5281/zenodo.12764574. URL: https://doi.org/10.5281/zenodo.12764574.

[106] The Theano Development Team et al. *Theano: A Python framework for fast computation of mathematical expressions*. 2016. arXiv: 1605.02688 [cs.SC]. URL: https://arxiv.org/abs/1605.02688.

[107] Mircea Trofin et al. "MLGO: a Machine Learning Guided Compiler Optimizations Framework". In: *CoRR* abs/2101.04808 (2021). arXiv: 2101.04808. URL: https://arxiv.org/abs/2101.04808.

[108] Gokul Varadhan et al. "Reliable implicit surface polygonization using visibility mapping". In: *Eurographics Symposium on Geometry Processing*. 2006. URL: https://api.semanticscholar.org/CorpusID:14198399.

[109] V. Vassilev et al. "Clad – Automatic Differentiation Using Clang and LLVM". In: vol. 608. 1. IOP Publishing, May 2015, p. 012055. DOI: 10.1088/1742-6596/608/1/012055. URL: https://iopscience.iop.org/article/10.1088/1742-6596/608/1/012055/pdf.

[110] Delio Vicini, Sébastien Speierer, and Wenzel Jakob. "Differentiable Signed Distance Function Rendering". In: *Transactions on Graphics (Proceedings of SIGGRAPH)* 41.4 (July 2022), 125:1–125:18. DOI: 10.1145/3528223.3530139.

[111] Andrea Walther, Andreas Griewank, and Olaf Vogel. "ADOL-C: Automatic Differentiation Using Operator Overloading in C++". In: *PAMM* 2.1 (2003), pp. 41–44. DOI: https://doi.org/10.1002/pamm.200310011. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/pamm.200310011. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/pamm.200310011.

[112] Max Willsey et al. "egg: Fast and Extensible Equality Saturation". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

[113] Fenggen Yu et al. "CAPRI-Net: Learning Compact CAD Shapes with Adaptive Primitive Assembly". In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022, pp. 11758–11768. DOI: 10.1109/CVPR52688.2022.01147.

[114] Wenbin Yu and Maxwell Blair. "DNAD, a simple tool for automatic differentiation of Fortran codes using dual numbers". In: *Computer Physics Communications* 184.5 (2013), pp. 1446–1452. DOI: 10.1016/j.cpc.2012.12.025. URL: http://www.sciencedirect.com/science/article/pii/S0010465513000027.

[115] Jingyang Zhang, Yao Yao, and Long Quan. "Learning Signed Distance Field for Multi-view Surface Reconstruction". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), pp. 6505–6514. URL: https://api.semanticscholar.org/CorpusID:237266838.

[116] Yihong Zhang et al. "Better Together: Unifying Datalog and Equality Saturation". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591239. URL: https://doi.org/10.1145/3591239.

Acronyms

AD automatic differentiation. 35, 37, 38, 41–43

CSG constructive solid geometry. 39, 42

DF distance function. 42

GLSL OpenGL shading language. 42

HLSL high level shading language. 42

RVSDG Regionalized Value State Dependency Graph. 36

SDF signed distance function. 40, 42, 43

UGF unit gradient field. 40

WASM web assembly. 40, 41

List of Figures

List of Tables

A. Appendix

Notes on table 5.1 For each cell: left is the Enzyme and right the Vola measurement. Note that Enzyme compiles to a Linux x86 dynlib, while Vola compiles to a WASM module which explains the difference in codesize. Also note that the runtime of the generated code is diminishing small. We instead opt to running each module 1000 times, and provide the average timing.

Name	Description
ADOL-C [111]	C++ based library, developed since at least 2003. Some research papers extend ADOL-C with custom solutions.
Aesara [21]	Tensor library for the Python eco-system. Based on Theano.
AutoDiff [71]	Lightweight C++17 library.
Autograd [78]	Python and Numpy library. Used in Pytorch, JAX and other higher-level ML tools.
Casadi [2]	C++ library from the mathematical computing community. In develop ment since at least 2019. Used in computer algebra systems with custom CasADi syntax.
CLAD [109]	Library for C++ code, implemented as Clang compiler plugin. In development since at least 2015.
DiffSharp [6]	F# library that follows PyTorch's naming convention. In development since at least 2014.
Enzyme [90]	LLVM-IR transformation based AD tool. Implemented for multiple languages that use LLVM as a compiler backend, like C/C++, Swift, Julia, Rust, Fortran and more.
ForwardDiff [97]	Light forward-mode library for Julia.
JuliaDiff	Collection of several AD related libraries and tools in the Julia community. The packages are heavily intertwined, which is why we treat them as one.
Minkowski Engine [12]	Nvidia library for sparse tensor AD. Slow but steady development since 2021, first mentioned in 2019.
PyTensor [22]	Python based tensor library with AD capabilities. Exposes the underlying computational graph at runtime. Fork of Aesara.
TensorFlow [80]	ML framework with Python and C++ API.
Zygote [51]	Julia based AD library. Used in the Flux programming framework. Acts as a compiler plugin similar to Casadi.

Table A.1.: List of selected AD tools

Forward Trace	Reverse Adjoint Trace
$v_0 = 1$	$\overline{v}_0 = \overline{v}_3 \frac{\partial v_3}{\partial v_0} = 0.134 * 2v_0 = 0.267$
$v_1 = 2$	$\overline{v}_1 = \overline{v}_4 \frac{\partial v_4}{\partial v_1} = 0.134 * 2v_1 = 0.535$
$v_2 = 3$	$\overline{v}_2 = \overline{v}_5 \frac{\partial v_5}{\partial v_2} = 0.134 * 2v_2 = 0.802$
$v_3 = v_0^2 = 1$	$\overline{v}_3 = \overline{v}_6 \frac{\partial v_6}{\partial v_3} = 0.134 * 1 = 0.134$
$v_4 = v_1^2 = 4$	$\overline{v}_4 = \overline{v}_6 \frac{\partial v_6}{\partial v_4} = 0.134 * 1 = 0.134$
$v_5 = v_2^2 = 9$	$\overline{v}_5 = \overline{v}_7 \frac{\partial v_7}{\partial v_5} = 0.134 * 1 = 0.134$
$v_6 = v_3 + v_4 = 5$	$\overline{v}_6 = \overline{v}_7 \frac{\partial v_7}{\partial v_6} = 0.134 * 1 = 0.134$
$v_7 = v_6 + v_5 = 14$	$\overline{v}_7 = \overline{v}_8 \frac{\partial v_8}{\partial v_7} = 1 * \frac{1}{2\sqrt{v7}} = 0.134$
$v_8 = \sqrt{v_7} = 3.741$	$\overline{v}_8 = \overline{v}_{10} \frac{\partial v_{10}}{\partial v_8} = 1 * 1 = 1$
$v_9 = 1$	$\overline{v}_9 = \overline{v}_{10} \frac{\partial v_{10}}{\partial v_9} = 1 * (-1) = (-1)$
$v_{10} = v_8 - v_9 = 2.741$	$\overline{v}_{10} = \overline{y} = \frac{\partial y}{\partial y} = 1$

Table A.2.: Forward execution of f_{sphere}, followed by reverse accumulation, that produces the derivative for each input.

	Compile time		AD compile		Runtime		Codesize	
	Enzyme	Vola	Enzyme	Vola	Enzyme	Vola	Enyme	Vola
Sphere	5.50s	207ms	2.50s	1.7ms	53μs	60μs	370.6kb	8.4kb
	5.58s	262ms	2.54s	2.9ms	-	-	370.6kb	8.4kb
	5.51s	225ms	2.51s	2.3ms	-	-	370.6kb	8.4kb
Union-Hard	5.62s	222ms	2.54s	1.8ms	57μs	60μs	371.1kb	67.6kb
	5.51s	236ms	2.43s	2.8ms	-	-	371.1kb	67.6kb
	5.61s	221ms	2.56s	2.0ms	-	-	371.1kb	67.6kb
Union-Soft	5.65s	225ms	2.54s	2.8ms	58μs	58μs	371.2kb	70.6kb
	5.61s	224ms	2.56s	3.9ms	-	-	371.2kb	70.5kb
	5.67s	219ms	2.53s	2.9ms	-	-	371.2kb	70.5kb
Blob	5.63s	225ms	2.61s	4.0ms	60μs	62μs	371.8kb	79.9kb
	5.59s	229ms	2.65s	4.3ms	-	-	371.8kb	79.9kb
	5.5s	230ms	2.7s	4.0ms	-	-	371.8kb	79.9kb

Table A.3.: Raw benchmarking data.

```
module stdlib::prelude;

operation SetColor(col: vec3);
impl SetColor<sub> for Color3d(at) {
    col
}

define common(at: vec3, offset: vec3, time: s) {
    Subtract(){
        Intersect(){
            SetColor([1.0, 0.0, 0.0]){
                Box([0.5; 3])
            }
        }{
            SetColor([0.0, 0.0, 1.0]){
                Sphere(0.7)
            }
        }
    }{
        SetColor([0.0, 1.0, 0.0]){
            Union(){
                Union(){
                    Cylinder(0.35, 1.0)
                }{
                    Rot3dAxis([1.0, 0.0, 0.0], 3.141 / 2.0){
                        Cylinder(0.35,1.0)
                    }
                }
            }{
                Rot3dAxis([0.0, 1.0, 0.0], 3.141 / 2.0){
                    Cylinder(0.35, 1.0)
                }
            }
        }
    }
}

export evalsdf(at: vec3, offset: vec3, time: s){
    csg myfield = common(at, offset, time);
    eval myfield.Sdf3d(at)
}
```

Figure A.1.: Full Vola source code for the model in fig. 1.3.

```rust
#![feature(autodiff)]
use glam::Vec3;
///Traits as "concept".
pub trait Sdf3d{
    fn eval_sdf3d(&self, at: Vec3) -> f32;
}

//Note "Box" is taken, use cuboid
struct Cuboid(Vec3);
struct Sphere(f32);
struct Union{
    left: Box<dyn Sdf3d>,
    right: Box<dyn Sdf3d>
}
struct Offset{
    offset: Vec3,
    sub: Box<dyn Sdf3d>
}

impl Sdf3d for Cuboid{
    fn eval_sdf3d(&self, at: Vec3) -> f32 {
        let q = at.abs() - self.0;
        q.max(Vec3::ZERO).length() + q.max_element().min(0.0)
    }
}
impl Sdf3d for Sphere{
    fn eval_sdf3d(&self, at: Vec3) -> f32 {
        at.length() - self.0
    }
}
impl Sdf3d for Union{
    fn eval_sdf3d(&self, at: Vec3) -> f32 {
        let a = self.left.eval_sdf3d(at);
        let b = self.right.eval_sdf3d(at);
        a.min(b)
    }
}
impl Sdf3d for Offset{
    fn eval_sdf3d(&self, at: Vec3) -> f32 {
        self.sub.eval_sdf3d(at - self.offset)
    }
}

#[autodiff(autodiff_sdf, Forward, Dual, Dual)]
fn sdf(at: Vec3, res: &mut f32) {
    let csg = Union{
        left: Box::new(Offset{
            offset: Vec3::X,
            sub: Box::new(Cuboid(Vec3::ONE))
        }),
        right: Box::new(Sphere(1.0))
    };
    *res = csg.eval_sdf3d(at);
}
```

Figure A.2.: Whole CSG-tree based Rust implementation