Journal of Scientific Computing manuscript No.
(will be inserted by the editor)

Operational calculus on programming spaces and generalized tensor networks

Žiga Sajovic · Martin Vuk

Abstract In this paper, we develop the theory of analytic virtual machines, that implement analytic programming spaces and operators acting upon them.

A programming space is a subspace of the function space of maps on a vector space serving as a virtual memory space. We construct an operator of differentiation on programming spaces by extending the virtual memory to a tensor product of the virtual memory space with tensor algebra of its dual. The extended virtual memory serves by itself as an algebra of programs, giving the expansion of the original program as an infinite tensor series at program's input values.

We present a theory of operators on programming spaces, that enables analysis of programs and computations on the operator level, which favors general implementation. Theory enables approximation and transformations of programs to a more appropriate function basis'. We also present several examples of how the theory can be used in computer science.

We generalize neural networks by constructing general tensor networks, that naturally exist in virtual memory. Transformations of programs to these trainable networks are derived, providing a meaningful way of network initialization. Theory opens new doors in program analysis, while fully retaining algorithmic control flow. We develop a general procedure which takes a program that tests an object for a property and constructs a program that imposes that property upon any object. We use it to generalize state of the art methods for analyzing neural networks to general programs and tensor networks.

Keywords programming spaces · operational calculus · differentiable programs · neural networks · tensor calculus · program analysis

Mathematics Subject Classification (2010) 47L99 · 47-04 · 46N99 · 26B40 · 26B12

Acknowledgements First author extends his gratitude to Jure Kališnik for proofreading the proofs, and Borut Robič for general guidance through the academia.

University of Ljubljana, Faculty of Computer and Information Science,
Večna pot 113, 1000 Ljubljana, Slovenia
E-mail: ziga.sajovic@gmail.com

1 Introduction

> *The enchanting charms of this sublime science reveal themselves in all their beauty only to those who have the courage to go deeply into it.*
>
> — Carl F. Gauss

Programming holds the power of algorithmic control flow and freedom of expression, whose abstinence severely limits descriptiveness of closed form methods of *pen and paper* mathematics, thus firmly cementing programming as the language of modern problem solving. Yet, a vibrant tradition of mathematics has existed since the dawn of mind, that remains, with the exception of combinatorics, largely untapped by computer science.

Just as the discrete nature of physical reality is studied through analytic means, so can the nature of digital phenomena. Studying these procedures as objects undergoing change in some virtual space, has partially been attempted in some fields, such as Hamiltonian Monte Carlo methods of Bayesian predictors, that Girolami and Calderhead [1] studied as manifolds to great success, using unfortunately impractical methods of hard-coding derivatives of distributions. This of course stifles the freedom of algorithmic expression programming is embraced for.

The way evaluation of algorithmic expressions differs from evaluation of symbolic expressions of standard analysis, lies at the very core of this dissonance. The disconnect was partially mitigated by methods of automatic differentiation [2], utilized today in machine learning, engineering, simulations, etc. Yet under the lack of a proper formalism the model collapses [3] when one tries to generalize to such notions as a differentiable program p_1 operating on (differentiable) derivatives of another program p_2 (where only coding of p_2 is required within p_1), while retaining the famed expressive freedom. Models allowing for nested differentiation [4], still fail in providing algorithms with an algebra enabling study and formulation of programs through analytic equations. Thus, existing models [5] [6] remain nothing more than efficient means to calculating derivatives, void of any meaningful algebraic insight, lacking the true power the vast field of analysis is endowed with.

The aim of this paper is bridging the gap between programming and analysis left behind by previous models. By generalizing them, we show them to be specific views of the same great elephant. Employing tensor algebra, we construct a virtual memory whose internal structure reflects and carries evaluation of algorithmic expressions. In Section 5 we provide an exact definition and construction of an analytic virtual machine, capable of implementing infinitely-differentiable programming spaces and operators acting upon them, supporting actions on multiple memory locations at a time. These programming spaces are shown to be a subalgebra, giving rise to symbolic manipulations of programs, while attaining construction by algorithmic control flow. Through it operators expanding a program into an infinite tensor series are derived in Section 5.2. The operator of program composition is constructed in Section 5.2.1, generalizing both forward [5] and reverse [6] mode of automatic differentiation to arbitrary order, under a single invariant operator in the theory. Through projections and embeddings in this space, the problem of nested differentiation is resolved in Section 5.2.2, where a program p_1 acts on differentiable derivatives of p_2, requiring only coding of p_2 within p_1.

Theory grants us the ability to choose programs' complexity through approximations inhabiting the virtual space. It being modeled by tensor algebra consisting of

multi-linear maps is tailor made for efficient implementation by GPU parallelism [7]. In Section 5.3, employing the developed algebra, we derive functional transformations of programs in an arbitrary function basis. As different hardware is optimized for running of different sets of functions, this proves useful with adapting code to fit specific hardware. We provide instructions on methods of usage in Section 5.6 and outline the process of how these transformations are to be interchangeably applied in practice.

Analytic virtual machines fully integrate control structures as shown in Section 5.5, thus retaining algorithmic control flow and the expressive power it possesses. We generalize neural networks by constructing general tensor networks in Section 6 and reveal their connection with programs in Section 6.1. In Section 6.1.1 we derive transformations of arbitrary programs to general tensor networks, providing a meaningful way of network initialization. These tensor networks are freely composed with general programs, as revealed in Section 6.1.2, allowing algorithmic constructions of layers performing specific task, like memory management [8][9]. All such constructs are trainable (as is their training process itself) and adhere to the operational calculus presented in this paper. This is demonstrated in Section 6.2, enabling analytic study through the theory.

Theory offers new insights into programs, as we demonstrate how to probe their inner structure in Section 7, revealing what actions properties of the domain most react to in Section 7.1. This enables us to alter data by imposing some property it lacks upon it. We generalize state of the art methods for analyzing neural networks [10] to general programs in Section 7.1.1 and provide a framework for analysis of machine learning architectures and other virtual constructs, as actions on the virtual space.

2 Computer programs as maps on a vector space

We will model computer programs as maps on a vector space. If we only focus on the real valued variables (of type `float` or `double`), the state of the virtual memory can be seen as a high dimensional vector[1]. A set of all the possible states of the program's memory, can be modeled by a finite dimensional real vector space $\mathcal{V} \equiv \mathbb{R}^n$. We will call $\mathcal{V}$ the *memory space of the program.* The effect of a computer program on its memory space $\mathcal{V}$, can be described by a map

$$P : \mathcal{V} \to \mathcal{V}. \tag{1}$$

A programing space is a space of maps $\mathcal{V} \to \mathcal{V}$ that can be implemented as a program in specific programming language.

Definition 21 (Euclidian virtual machine) *The tuple $(\mathcal{V}, \mathcal{F})$ is an Euclidian virtual machine, where*

- *$\mathcal{V}$ is a finite dimensional vector space over a complete field K, serving as a memory*[2]
- *$\mathcal{F} < \mathcal{V}^{\mathcal{V}}$ is a subspace of the space of maps $\mathcal{V} \to \mathcal{V}$, called* programming space*, serving as actions on the memory.*

[1] we assume the variables of interest to be of type `float` for simplicity. Theoretically any field can be used instead of $\mathbb{R}$.

[2] In most applications the field K will be $\mathbb{R}$

3 Differentiable programs

To define differentiable programs, let us first recall some definitions from multivariate calculus.

Definition 31 (Derivative) *Let V, U be Banach spaces. The map $P : V \to U$ is differentiable at the point $x \in V$, if there exists a linear bounded operator $TP_x : V \to U$ such that*

$$\lim_{h \to 0} \frac{\|P(x+h) - P(x) - TP_x(h)\|}{\|h\|} = 0. \tag{2}$$

The map TP_x is called the Fréchet derivative *of the map P at the point $\mathbf{x}$.*

For maps $\mathbb{R}^n \to \mathbb{R}^m$ Fréchet derivative can be expressed by multiplication of vector h by the Jacobi matrix $\mathrm{J}P$ of the partial derivatives of the components of the map P

$$T_x P(h) = \mathrm{J}P(x) \cdot h.$$

We assume for the remainder of this section that the map $P : V \to U$ is differentiable for all $\mathbf{x} \in V$. The derivative defines a map from V to linear bounded maps from V to U. We further assume U and V are finite dimensional. Then the space of linear maps from V to U is isomorphic to tensor product $U \otimes V^*$, where the isomorphism is given by the tensor contraction, sending a simple tensor $\mathbf{u} \otimes f \in U \otimes V^*$ to a linear map

$$\mathbf{u} \otimes f : \mathbf{x} \mapsto f(\mathbf{x}) \cdot \mathbf{u}. \tag{3}$$

The derivative defines a map

$$\partial P : V \to U \otimes V^* \tag{4}$$

$$\partial P : \mathbf{x} \mapsto T_{\mathbf{x}} P. \tag{5}$$

One can consider the differentiability of the derivative itself ∂P by looking at it as a map (4). This leads to the definition of the higher derivatives.

Definition 32 (higher derivatives) *Let $P : V \to U$ be a map from vector space V to vector space U. The derivative $\partial^k P$ of order k of the map P is the map*

$$\partial^k P : V \to U \otimes (V^*)^{\otimes k} \tag{6}$$

$$\partial^k P : \mathbf{x} \mapsto T_{\mathbf{x}} \left(\partial^{k-1} P \right) \tag{7}$$

Remark 31 *For the sake of clarity, we assumed in the definition above, that the map P as well as all its derivatives are differentiable at all points $\mathbf{x}$. If this is not the case definitions above can be done locally, which would introduce mostly technical difficulties.*

Let $\mathbf{e}_1, \ldots, \mathbf{e}_n$ be a basis of U and $x_1, \ldots x_m$ the basis of V^*. Denote by $P_i = x_i \circ P$ the $i-th$ component of the map P according to the basis $\{\mathbf{e}_i\}$ of U. Then $\partial^k P$ can be defined in terms of directional(partial) derivatives by the formula

$$\partial^k P = \sum_{\forall_{i,\alpha}} \frac{\partial^k P_i}{\partial x_{\alpha_1} \ldots \partial x_{\alpha_k}} \mathbf{e}_i \otimes dx_{\alpha_1} \otimes \ldots \otimes dx_{\alpha_k}. \tag{8}$$

3.1 Differentiable programs

We want to be able to represent the derivatives of a computer program in an Euclidean virtual machine again as a program in the same euclidean virtual machine. We define three subspaces of the virtual memory space $\mathcal{V}$, that describe how different parts of the memory influence the final result of the program.

Denote by $e_1, \dots e_n$ a standard basis of the memory space $\mathcal{V}$ and by $x_1, \dots x_n$ the dual basis of $\mathcal{V}^*$. The functions x_i are coordinate functions on $\mathcal{V}$ and correspond to individual locations(variables) in the program memory.

Definition 33 *For each program P in the programming space $\mathcal{F} < \mathcal{V}^{\mathcal{V}}$, we define the* input *or* parameter space $I_P < \mathcal{V}$ *and the* output space $O_P < \mathcal{V}$ *to be the minimal vector sub-spaces spanned by the standard basis vectors, such that the map P_e, defined by the following commutative diagram*

$$\begin{array}{ccc} \mathcal{V} & \xrightarrow{P} & \mathcal{V} \\ {\scriptstyle \mathbf{i}\mapsto\mathbf{i}+\mathbf{f}}\uparrow & & \downarrow{\scriptstyle \mathrm{pr}_{O_P}} \\ I_P & \xrightarrow{P_e} & O_P \end{array} \tag{9}$$

does not depend of the choice of the element $\mathbf{f} \in F_P = (I_P + O_P)^{\perp}$.

The space $F_P = (I_P + O_P)^{\perp}$ is called free space *of the program P.*

The variables x_i corresponding to standard basis vectors spanning the parameter, output and free space are called *paramters* or *input variables*, *output variables* and *free variables* correspondingly. Free variables are those that are left intact by the program and have no influence on the final result other than their value itself. The output of the program depends only on the values of the input variables and consists of variables that have changed during the program. Input parameters and output values might overlap.

The map P_e is called the *effective map* of the program P and describes the actual effect of the program P on the memory ignoring the free memory.

The derivative of the effective map is of interest, when we speak about differentiability of computer programs.

Definition 34 (Automatically differentiable programs) *A program $P : \mathcal{V} \to \mathcal{V}$ is* automatically differentiable *if there exist an embedding of the space $O_P \otimes I_P^*$ into the free space F_P, and a program $\partial P : \mathcal{V} \to \mathcal{V}$, such that its effective map is the map*

$$P_e \oplus \partial P_e : I_P \to O_P \oplus (O_P \otimes I^*). \tag{10}$$

A program $P : \mathcal{V} \to \mathcal{V}$ is automatically differentiable of order k *if there exist a program $\tau_k P : \mathcal{V} \to \mathcal{V}$, such that its effective map is the map*

$$P_e \oplus \partial P_e \oplus \dots \partial^k P_e : I_P \to O_P \oplus \left(O_P \otimes I^*\right) \oplus \dots \left(O_P \otimes \left(I_p^*\right)^{k\otimes}\right). \tag{11}$$

If a program $P : \mathcal{V} \to \mathcal{V}$ is automatically differentiable then it is also differentiable as a map $\mathcal{V} \to \mathcal{V}$. However only the derivative of program's effective map can be implemented as a program, since the memory space is limited to $\mathcal{V}$. To be able to differentiate a program to the k-th order, we have to calculate and save all the derivatives of the orders k and less.

4 Differentiable programming spaces

4.1 Virtual memory space for differentiable programs

Motivated by the Definition 34, we define virtual memory for differentiable programs as a sequence of vector spaces with the recursive formula

$$\mathcal{V}_0 = \mathcal{V} \tag{12}$$

$$\mathcal{V}_k = \mathcal{V}_{k-1} + \left(\mathcal{V}_{k-1} \otimes \mathcal{V}^*\right). \tag{13}$$

Note that the sum is not direct, since some of the subspaces of $\mathcal{V}_{k-1}$ and $\mathcal{V}_{k-1} \otimes \mathcal{V}^*$ are naturally isomorphic and will be identified[3].

The space

$$\mathcal{V}_k = \mathcal{V} \otimes \left(K \oplus \mathcal{V}^* \oplus (\mathcal{V}^* \otimes \mathcal{V}^*) \oplus \ldots (\mathcal{V}^*)^{\otimes k}\right) = \mathcal{V} \otimes T_k(\mathcal{V}^*) \tag{14}$$

satisfies the recursive formula (12), where the space $T_k(\mathcal{V}^*)$ is a subspace of *tensor algebra* $T(\mathcal{V}^*)$, consisting of linear combinations of tensors of rank less or equal k. This construction enables us to define all the derivatives as maps with the same domain and codomain $\mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$. Putting memory considerations aside, we propose an universal model of the memory for differentiable programs.

Definition 41 *Let $(\mathcal{V}, \mathcal{F})$ be an Euclidean virtual machine and let*

$$\mathcal{V}_\infty = \mathcal{V} \otimes T(\mathcal{V}^*) = \mathcal{V} \oplus (\mathcal{V} \otimes \mathcal{V}^*) \oplus \ldots, \tag{15}$$

where $T(\mathcal{V}^*)$ is the tensor algebra of the dual space $\mathcal{V}^*$. We call $\mathcal{V}_\infty$ the differentiable virtual memory *of a virtual computing machine $(\mathcal{V}, \mathcal{F})$.*

The term virtual memory is used as it is only possible to embed certain subspaces of $\mathcal{V}_\infty$ into memory space $\mathcal{V}$, making it similar to virtual memory as a memory management technique.

We can extend each program $P : \mathcal{V} \to \mathcal{V}$ to the map on universal memory space $\mathcal{V}_\infty$ by setting the first component in the direct sum (15) to P, and all other components to zero. Similarly derivatives $\partial^k P$ can be also seen as maps from $\mathcal{V}$ to $\mathcal{V}_\infty$ by setting k-th component in the direct sum (15) to $\partial^k P$ and all others to zero.

4.2 Differentiable programming spaces

Let us define the following function spaces:

$$\mathcal{F}_n = \{f : \mathcal{V} \to \mathcal{V} \otimes T_n(\mathcal{V}^*)\} \tag{16}$$

All of these function spaces are can be seen as sub spaces of $\mathcal{F}_\infty = \{f : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)\}$, since $\mathcal{V}$ is naturally embedded into $\mathcal{V} \otimes T(\mathcal{V}^*)$. The Fréchet derivative defines an operator on the space of smooth maps from $\mathcal{F}_\infty$[4]. We denote this operator ∂. The

[3] The spaces $\mathcal{V} \otimes (\mathcal{V}^*)^{\otimes(j+1)}$ and $\mathcal{V} \otimes (\mathcal{V}^*)^{\otimes j} \otimes \mathcal{V}^*$ are naturally isomorphic and will be identified in the sum.

[4] the operator ∂ may be defined partially for other maps as well, but we will handle this case later.

image of any map $P : \mathcal{V} \to \mathcal{V}$ by operator ∂ is its first derivative, while the higher order derivatives are just powers of operator ∂ applied to P. Thus ∂^k is a mapping between function spaces (16)

$$\partial^k : \mathcal{F}^n \to \mathcal{F}^{n+k}. \tag{17}$$

Definition 42 *A* differentiable programming space $\mathcal{P}_0$ *is any subspace of* $\mathcal{F}_0$ *such that*

$$\partial\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{18}$$

When all elements of $\mathcal{P}_0$ *are analytic, we call* $\mathcal{P}_0$ *an* analytic programming space.

Theorem 41 *Any differentiable programming space* $\mathcal{P}_0$ *is an infinitely differentiable programming space, meaning that*

$$\partial^k\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{19}$$

for any $k \in \mathbb{N}$.

Proof By induction on order k. For $k = 1$ the claim holds by Definition 42. Assume $\forall_{P\in\mathcal{P}_0}, \partial^n\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*)$. Denote by $P^i_{\alpha,k}$ the component of the k-th derivative for a multiindex α denoting the component of $T(\mathcal{V}^*)$ and an index i denoting the component of $\mathcal{V}$.

$$\partial^{n+1}P^i_{\alpha,k} = \partial(\partial^n P^i_\alpha)_k \wedge (\partial^n P^i_\alpha) \in \mathcal{P}_0 \implies \partial(\partial^n P^i_\alpha)_k \in \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{20}$$

$$\Longrightarrow$$

$$\partial^{n+1}\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*)$$

Thus by induction, the claim (19) holds for all $k \in \mathbb{N}$.

Definition 43 *Let* $\mathcal{P}_0$ *be a differentiable programming space. The space* $\mathcal{P}_n < \mathcal{F}_n$ *spanned by* $\mathcal{D}^n\mathcal{P}_0$ *over* K*, where* $\mathcal{D}^n = \{\partial^k; \quad 0 \le k \le n\}$*, is called a differentiable programming space of order* n*. If all elements of* $\mathcal{P}_0$ *are analytic, than so are the elements of* $\mathcal{P}_n$*.*

Corollary 1 *A differentiable programming space of order* n*,* $\mathcal{P}_n : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$*, can be embedded into the tensor product of the function space* $\mathcal{P}_0 : \mathcal{V} \to \mathcal{V}$ *and the subspace* $T_n(\mathcal{V}^*)$ *of tensor algebra of the dual of the virtual space* $\mathcal{V}$*.*

By taking the limit as $n \to \infty$, we will consider

$$\mathcal{P}_\infty < \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*), \tag{21}$$

where $\mathcal{T}(\mathcal{V}^*) = \prod_{k=0}^{\infty}(\mathcal{V}^*)^{\otimes k}$ is *the tensor series algebra*, the algebra of the infinite formal tensor series, which is a completion of the tensor algebra $T(\mathcal{V}^*)$ in suitable topology.

5 Operational calculus on programming spaces

By Corollary 1 we may represent calculation of derivatives of the map $P : \mathcal{V} \to \mathcal{V}$, with only one mapping τ. We define the operator τ_n as a direct sum of operators

$$\tau_n = 1 + \partial + \partial^2 + \ldots + \partial^n \tag{22}$$

The image $\tau_k P(\mathbf{x})$ is a multitensor of order k, which is a direct sum of the maps value and all derivatives of order $n \leq k$, all evaluated at the point $\mathbf{x}$:

$$\tau_k P(\mathbf{x}) = P(\mathbf{x}) + \partial_{\mathbf{x}} P(\mathbf{x}) + \partial^2_{\mathbf{x}} P(\mathbf{x}) + \ldots + \partial^k_{\mathbf{x}} P(\mathbf{x}). \tag{23}$$

The operator τ_n satisfies the recursive relation.

$$\tau_{k+1} = 1 + \partial \tau_k, \tag{24}$$

that can be used to recursively construct programming spaces of arbitrary order.

Claim 51 *Only explicit knowledge of $\tau : \mathcal{P}_0 \to \mathcal{P}_1$ is required for the construction of $\mathcal{P}_n$ from $\mathcal{P}_1$.*

Proof The construction is achieved by following the argument (20) of the proof of Theorem 41, allowing simple implementation, as dictated by (24).

Remark 51 *Maps $\mathcal{V} \otimes T(\mathcal{V}^*) \to \mathcal{V} \otimes T(\mathcal{V}^*)$ are constructible using tensor algebra operations and compositions of programs in $\mathcal{P}_n$.*

Definition 51 (Algebra product) *For any bilinear map $\cdot : \mathcal{V} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ we can define a bilinear product $\cdot$ on $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ by the following rule on the simple tensors:*

$$\begin{aligned} (\mathbf{v} \otimes f_1 \otimes \ldots f_k) \cdot (\mathbf{u} \otimes g_1 \otimes \ldots) &= f_k(\mathbf{u}) \mathbf{v} \otimes f_1 \otimes \ldots f_{k-1} \otimes g_1 \otimes \ldots \\ \mathbf{v} \cdot (\mathbf{u} \otimes g_1 \otimes \ldots g_l) &= (\mathbf{v} \cdot \mathbf{u}) \otimes g_1 \otimes \ldots g_l \end{aligned} \tag{25}$$

Theorem 51 (Programming algebra) *An infinitely-differentiable programming space $\mathcal{P}_\infty$ is a function algebra, with the product defined by* (25) *for any bilinear map $\cdot : \mathcal{V} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$.*

5.1 Analytic virtual machine

We propose an abstract computational model, a virtual machine capable of constructing and implementing the derived theory. Such a machine provides a framework for analytic study of algorithmic procedures through algebraic means.

Claim 52 *The tuple $(\mathcal{V}, \mathcal{P}_0)$ and the belonging* tensor series algebra *are sufficient conditions for the existence and construction of* infinitely differentiable programing spaces *$\mathcal{P}_\infty$* (43)*, which are* function algebras *by Theorem* 51*, through linear combinations of elements of $\mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*)$*

Definition 52 (Analytic virtual machine) *The tuple $M = \langle \mathcal{V}, \mathcal{P}_0 \rangle$ is an analytic, infinitely differentiable virtual machine, where*

- *$\mathcal{V}$ is a finite dimensional vector space*
- *$\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ is the virtual memory space*

– *$\mathcal{P}_0$ is an analytic programming space over $\mathcal{V}$*

When $\mathcal{P}_0$ is a differentiable programming space, this defines an infinitely differentiable virtual machine.

Algebraic manipulations these machines enable are employed throughout the paper, revealing analytic insights in Section 7. All analytic virtual machines fully integrate usage of control structures, as it is demonstrated and proven in Section 5.5, thus retaining algorithmic control flow and the expressive power it possesses.

5.1.1 Implementation

An illustrative example of the implementation of the theory is available on github [11]. Implementation closely follows theorems and derivations of this paper, intended as an educational guide.

Virtual memory $\mathcal{V}$ is constructed and expanded to $\mathcal{V} \otimes T(\mathcal{V}^*)$, serving by itself as an algebra of programs. We provide a construction of an analytic programming space $C{+}{+} \subset \mathcal{P}_0 : \mathcal{V} \to \mathcal{V}$ and expand it to $\mathcal{D}^n C{+}{+} \subset \mathcal{P}_n : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$ using the developed operational calculus. A paper [12] explaining the process of implementation accompanies the source-code.

5.2 Tensor series expansion

Expansion into a series offers valuable insights into programs through methods of analysis, as explored in the coming sections.

There exists a space spanned by the set $\mathcal{D}^n = \{\partial^k; \quad 0 \leq k \leq n\}$ over a field K. Thus, the expression

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{(h\partial)^n}{n!} \tag{26}$$

is well defined. In coordinates, the operator $e^{h\partial}$ can be written as a series over all multi-indices α

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall_{i,\alpha}} \frac{\partial^n}{\partial x_{\alpha_1} \dots \partial x_{\alpha_n}} \mathbf{e}_i \otimes dx_{\alpha_1} \otimes \dots \otimes dx_{\alpha_n}. \tag{27}$$

The operator $e^{h\partial}$ is a mapping between function spaces (16)

$$e^{h\partial} : \mathcal{P} \to \mathcal{P}_\infty. \tag{28}$$

It also defines a map

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*), \tag{29}$$

by taking the image of the map $e^{h\partial}(P)$ at a certain point $\mathbf{v} \in \mathcal{V}$. Through (29) we may construct a map from the space of programs, to the space of polynomials. Note that the space of multivariate polynomials $\mathcal{V} \to K$ is isomorphic to symmetric algebra $S(\mathcal{V}^*)$, which is in turn a quotient of tensor algebra $T(\mathcal{V}^*)$. To any element of $\mathcal{V} \otimes T(\mathcal{V}^*)$ one can attach corresponding element of $\mathcal{V} \otimes S(\mathcal{V}^{i*})$ namely a polynomial map $\mathcal{V} \to \mathcal{V}$. Thus, similarly to (21), we consider the completion of the symmetric algebra $S(\mathcal{V}^*)$

as the *formal power series* $\mathcal{S}(\mathcal{V}^*)$, which is in turn isomorphic to a quotient of *tensor series algebra* $\mathcal{T}(\mathcal{V}^*)$, arriving at

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^{i*}) \tag{30}$$

For any element $\mathbf{v}_0 \in \mathcal{V}$, the expression $e^{h\partial}(\cdot, \mathbf{v}_0)$ is a map $\mathcal{P} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^*)$, mapping a program to a formal power series. We can express the correspondence between multi-tensors in $\mathcal{V} \otimes T(\mathcal{V}^*)$ and polynomial maps $\mathcal{V} \to \mathcal{V}$ given by multiple contractions for all possible indices. For a simple tensor $\mathbf{u} \otimes f_1 \otimes \ldots \otimes f_n \in \mathcal{V} \otimes (\mathcal{V}^*)^{\otimes n}$ the contraction by $\mathbf{v} \in \mathcal{V}$ is given by applying co-vector f_n to $\mathbf{v}$

$$\mathbf{u} \otimes f_1 \otimes \ldots \otimes f_n \cdot \mathbf{v} = f_n(\mathbf{v})\mathbf{u} \otimes f_1 \otimes \ldots f_{n-1}. \tag{31}$$

Applying contraction multiple times with the same vector $\mathbf{v}$ one can assign to any simple tensor a monomial map

$$\mathbf{u} \otimes f_1 \otimes \ldots \otimes f_n : \mathbf{v} \mapsto f_n(\mathbf{v}) \cdots f_1(\mathbf{v})\mathbf{u} \tag{32}$$

and by linearity to any finite rank multi-tensor in $\mathcal{V} \otimes T(\mathcal{V}^*)$ a polynomial map.

Theorem 52 *For a program $P \in \mathcal{P}$ the expansion into an infinite tensor series at the point $\mathbf{v}_0 \in \mathcal{V}$ is expressed by multiple contractions*

$$\begin{aligned} P(\mathbf{v}_0 + h\mathbf{v}) = \left((e^{h\partial}P)(\mathbf{v}_0)\right) \cdot \mathbf{v} &= \sum_{n=0}^{\infty} \frac{h^n}{n!} \partial^n P(\mathbf{v}_0) \cdot (\mathbf{v}^{\otimes n}) \\ &= \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall_{i,\alpha}} \frac{\partial^n P_i}{\partial x_{\alpha_1} \ldots \partial x_{\alpha_n}} \mathbf{e}_i \cdot dx_{\alpha_1}(\mathbf{v}) \cdot \ldots \cdot dx_{\alpha_n}(\mathbf{v}). \end{aligned} \tag{33}$$

Proof We will show that $\frac{d^n}{dh^n}(\text{LHS})|_{h=0} = \frac{d^n}{dh^n}(\text{RHS})|_{h=0}$. Then LHS and RHS as functions of h have coinciding Taylor series and are therefore equal.

$\Longrightarrow$

$$\frac{d^n}{dh^n} P(\mathbf{v}_0 + h\mathbf{v})\Big|_{h=0} = \partial^n P(\mathbf{v}_0)(\mathbf{v})$$

$\Longleftarrow$

$$\frac{d^n}{dh^n} \left((e^{h\partial})(P)(\mathbf{v}_0)\right)(\mathbf{v})\Big|_{h=0} = \left((\partial^n e^{h\partial})(P)(\mathbf{v}_0)\right)(\mathbf{v})\Big|_{h=0}$$

$$\wedge$$

$$\partial^n e^{h\partial}\Big|_{h=0} = \sum_{i=0}^{\infty} \frac{h^i \partial^{i+n}}{i!}\Big|_{h=0} = \partial^n$$

$$\Longrightarrow$$

$$\left(\partial^n(P)(\mathbf{v}_0)\right)(\mathbf{v}^{\otimes n})$$

Remark 52 *The operator $e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ evaluated at $h = 1$ is a broad generalization of the shift operator [13], as the theory it exists in offers more than a mere shift, as will become apparent in the coming sections. For a specific $v_0 \in \mathcal{V}$ it is hereon denoted by*

$$e^{\partial}|_{v_0} : \mathcal{P} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*) \tag{34}$$

When the choice of $v_0 \in \mathcal{V}$ is arbitrary, we omit it from expressions for brevity.

The image of the contraction is an element of the original virtual space $\mathcal{V}^i$. Independence of the operator (29) from a coordinate system, translates to independence in execution. Thus the expression (33) is invariant to the point in execution of a program, a fact we explore later on.

It is trivial to show that the operator $e^{h\partial}$ is an automorphism of the programming algebra $\mathcal{P}_\infty$,

$$e^{h\partial}(p_1 \cdot p_2) = e^{h\partial}(p_1) \cdot e^{h\partial}(p_2) \tag{35}$$

where $\cdot$ stands for any bilinear map.

Corollary 1 through (21) implies $e^{h\partial}(\mathcal{P}_0) \subset \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*)$ which enables efficient implementation.

5.2.1 Operator of program composition

In this section we generalize both forward [5] and reverse [6] mode of automatic differentiation to arbitrary order, under a single invariant operator in the theory. We demonstrate how calculations can be performed on the operator level, easing general implementation. This condenses complex notions to simple expressions allowing meaningful manipulations before being applied to a particular programming space.

Theorem 53 *Composition of maps $\mathcal{P}$ is expressed as*

$$e^{h\partial}(f \circ g) = \exp(\partial_f e^{h\partial_g})(g, f) \tag{36}$$

where $\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \times \mathcal{P} \to \mathcal{P}_\infty$ is an operator on pairs of maps (g, f), where ∂_g is applied to g, and ∂_f to f.

Proof We will show that $\frac{d^n}{dh^n}(\text{LHS})|_{h=0} = \frac{d^n}{dh^n}(\text{RHS})|_{h=0}$. Then LHS and RHS as functions of h have coinciding Taylor series and are therefore equal.

$\Longrightarrow$

$$\lim_{\|h\|\to 0} (\frac{d}{dh})^n e^{\partial}(f \circ g) = \lim_{\|h\|\to 0} \partial^n e^{h\partial}(f \circ g)$$
$$\Longrightarrow$$
$$\partial^n (f \circ g) \tag{37}$$

$\Longleftarrow$

$$\exp(\partial_f e^{h\partial_g}) = \exp\left(\partial_f \sum_{i=0}^{\infty} \frac{(h\partial_g)^i}{i!}\right) = \prod_{i=1}^{\infty} e^{\partial_f \frac{(h\partial_g)^i}{i!}}\left(e^{\partial_f}\right)$$
$$\Longrightarrow$$
$$\exp(\partial_f e^{h\partial_g})(g, f) = \sum_{\forall_n} h^n \sum_{\lambda(n)} \prod_{k\cdot l \in \lambda} \left(\frac{\partial_f \partial_g^l(g)}{l!}\right)^k \frac{1}{k!}\left(\left(e^{\partial_f}\right) f\right)$$

where $\lambda(n)$ stands for the partitions of n. Thus

$$\lim_{\|h\|\to 0} (\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k\cdot l \in \lambda} \left(\frac{\partial_f \partial_g^l(g)}{l!}\right)^k \frac{1}{k!}\left(\left(e^{\partial_f}\right) f\right) \tag{38}$$

taking into consideration the fact that $e^{\partial_f}(f)$ evaluated at a point $v \in \mathcal{V}$ is the same as evaluating f at v, the expression (38) equals (37) by Faà di Bruno's formula.

$$\lim_{\|h\|\to 0} (\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k\cdot l \in \lambda} \left(\frac{\partial_f \partial_g^l(g(v))}{l!}\right)^k \frac{1}{k!}\Big(f(g(v))\Big) \tag{39}$$

The Theorem 53 enables an invariant implementation of the operator of program composition in $\mathcal{P}_n$, expressed as a tensor series through (36) and (38).

By fixing one mapping in

$$\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \times \mathcal{P} \to \mathcal{P}_\infty, \tag{40}$$

the operator $\exp(\partial_f e^{h\partial_g})(g)$ performs a pullback of an arbitrary map through g.

$$\exp(\partial_f e^{h\partial_g})(g) : \mathcal{P} \to \mathcal{P}_\infty(g) \tag{41}$$

Claim 53 *When projecting onto the space spanned by* $\{1, \partial\}$*, the action of the operator* $\exp(\partial_f e^{h\partial_g})(g)$ *of* (41) *is equivalent to forward [5] mode automatic differentiation. Both forward [5] and reverse [6] mode (generalized to arbitrary order) are obtainable using the same operator* $\exp(\partial_f e^{h\partial_g})$ *of* (40)*, by fixing the appropriate one of the two maps. This generalizes both concepts under a single operator in the theory. The operator* (36) *can be generalized for the notion of a pullback to arbitrary operators.*

Thus, through (36) and all its' descendants (exponents), the operator (41) grants invariance to the point in execution of a program, which proves useful as invariants are at the center of proving algorithms' correctness. This is analogous to the principle of general covariance [14][See section 7.1] in general relativity, the invariance of the form of physical laws under arbitrary differentiable coordinate transformations. This principle and what it offers to programming is explored in Section 5.6.

With this we turn towards easing such calculations, towards completing them on the level of operators. The derivative $\frac{d}{dh}$ of (41) is

$$\frac{d}{dh} \exp(\partial_f e^{h\partial_g})(g) = \partial_f(\partial_g g) e^{h\partial_g} \exp(\partial_f e^{h\partial_g})(g) \tag{42}$$

We note an important distinction to the operator $e^{h\partial_g}$, the derivative of which is

$$\frac{d}{dh} e^{h\partial_g} = \partial_g e^{h\partial_g} \tag{43}$$

We may now compute derivatives (of arbitrary order) of the pullback operator. As an example we compute the second derivative.

$$\left(\frac{d}{dh}\right)^2 \exp\left(\partial_f e^{h\partial_g}\right)(g) = \frac{d}{dh}\left(\partial_f(\partial_g g) e^{h\partial_g} \exp\left(\partial_f e^{h\partial_g}\right)(g)\right)$$

which is by equations (42) and (43), using algebra and correct applications

$$\left(\partial_f(\partial_g^2 g)\right) e^{h\partial_g} \exp(\partial_f e^{h\partial_g})(g) + (\partial_f^2(\partial_g g)^2) e^{2h\partial_g} \exp(\partial_f e^{h\partial_g})(g) \tag{44}$$

The operator is always shifted to the evaluating point (29) $v \in \mathcal{V}$, thus, only the behavior in the limit as $h \to 0$ is of importance. Taking this limit in the expression (44) we obtain the operator

$$\left(\partial_f(\partial_g^2 g) + \partial_f^2(\partial_g g)^2\right) \exp(\partial_f) : \mathcal{P} \to \partial^2 \mathcal{P}(g) \tag{45}$$

Thus, without imposing any additional rules, we computed the operator of the second derivative of composition with g, directly on the level of operators. The result of course matches the equation (38) for $n = 2$.

As it is evident from the example, calculations using operators are far simpler, than direct manipulations of functional series, as it was done in the proof of Theorem 53. This of course enables a simpler implementation, that functions over arbitrary programming (function) spaces. In the space that is spanned by $\{\partial^n \mathcal{P}_0\}$ over K, derivatives of compositions may be expressed solely through the operators, using only the product rule (35), the derivative of the composition operator (42) and the derivative of the general shift operator (43). Thus, explicit knowledge of rules for differentiating compositions is unnecessary, as it is contained in the structure of the operator $exp(\partial_f e^{h\partial_g})$ itself, which is differentiated using standard rules, as in the above example.

$$\partial^n(f \circ g) = \left(\frac{d}{dh}\right)^n \exp\left(\partial_f e^{h\partial_g}\right)(g, f)\Big|_{h=0} \tag{46}$$

Corollary 2 *The operator $e^{h\partial}$ commutes with composition over $\mathcal{P}$*

$$e^{h\partial}(p_2 \circ p_1) = e^{h\partial}(p_2) \circ e^{h\partial}(p_1) \tag{47}$$

Proof Follows from (30) and Theorem 53.

Remark 53 *With explicit evaluations in Corollary 2*

$$e^{h\partial}|_{v_0}(p_n \circ \cdots \circ p_0) = e^{h\partial}|_{v_n}(p_n) \circ \cdots \circ e^{h\partial}|_{v_0}(p_0) \tag{48}$$

the wise choice of evaluation points is $v_i = p_{i-1}(v_{i-1}) \in \mathcal{V}$.

5.2.2 Order reduction for nested applications

It is useful to be able to use the k-th derivative of a program $P \in \mathcal{P}$ as part of a different differentiable program P_1. As such, we must be able to treat the derivative itself as a differentiable program $P'^k \in \mathcal{P}$, while only coding the original program P.

Theorem 54 *There exists a reduction of order map $\phi : \mathcal{P}_n \to \mathcal{P}_{n-1}$, such that the following diagram commutes*

$$\begin{array}{ccc} \mathcal{P}_n & \xrightarrow{\phi} & \mathcal{P}_{n-1} \\ \downarrow{\scriptstyle \partial} & & \downarrow{\scriptstyle \partial} \\ \mathcal{P}_{n+1} & \xrightarrow{\phi} & \mathcal{P}_n \end{array} \tag{49}$$

satisfying

$$\forall_{P_1 \in \mathcal{P}_0} \exists_{P_2 \in \mathcal{P}_0} \left(\phi^k \circ e_n^{\partial}(P_1) = e_{n-k}^{\partial}(P_2)\right) \tag{50}$$

for each $n \geq 1$, where e_n^{∂} is the projection of the operator e^{∂} onto the set $\{\partial^n\}$.

Corollary 3 *By Theorem 54, n-differentiable k-th derivatives of a program $P \in \mathcal{P}_0$ can be extracted by*

$${}^n P^{k\prime} = \phi^k \circ e_{n+k}^{\partial}(P) \in \mathcal{P}_n \tag{51}$$

Thus, we gained the ability of writing a differentiable program acting on derivatives of another program, stressed as crucial (but lacking in most models) by other authors [4]. Usage of the reduction of order map and other constructs of this Section are demonstrated in Section 7, as we analyze procedures as systems inducing change upon objects in virtual space.

5.3 Functional transformations of programs

Let's suppose a hardware H, optimized for the set of functions $F = \{f_i : \mathcal{V} \to \mathcal{V}\}$. The set F is specified by the manufacturer.

With technological advances, switching the hardware is common, which can lead to a decline in performance. Thus, we would like to employ transformations of a program $P \in \mathcal{P}$ in basis F. It is common to settle for a suboptimal algorithm, that is efficient on hardware H. Sub-optimality of the algorithm depends on the set F, whether it spans P, or not. A classic example of a transformation, is the Fourier transform in the basis $\{\sin(nx), \cos(mx)\}$, that spans $\mathcal{P}$ (which as stated is not true for an arbitrary set F).

Using the developed tools, the problem is solvable using linear algebra. Let e_n^∂ denote the projection of the operator e^∂, onto the first n basis vectors $\{\partial^i\}$. We can, by Theorem 52, construct a map (30) from the space of programs, to the space of polynomials, with unknowns in $\mathcal{V}^k$, using the operator e^∂. Let $\mathcal{X} = \{p_i\}$ denote a basis of the space of polynomials $\mathcal{V} \to \mathcal{V}$ [5]. Than, we can interpret $e_n^\partial(P \in \mathcal{P})$ as a vector of linear combinations of $\mathcal{X}$, which is assumed hereon.

Thus, we define the tensor of basis transformation $F \to \mathcal{X}$

$$T_{\mathcal{X}F} = p_1 \otimes e_n^\partial(f_1)^* + p_2 \otimes e_n^\partial(f_2)^* + \ldots + p_n \otimes e_n^\partial(f_n)^* \tag{52}$$

and the tensor of basis transformation $\mathcal{X} \to F$ is

$$T_{F\mathcal{X}} = T_{\mathcal{X}F}^{-1} \tag{53}$$

Using the tensor (53), we can easily perform basis transformations $\mathcal{X} \to F$. For a specific set F (and consequentially a hardware H, upon which the set F is conditioned), the tensor (53) only has to be computed once, and can then be used for transforming arbitrary programs (while using the same operator e_n^∂). Thus, the coordinates of program $P \in \mathcal{P}$ in basis F are

$$P_F = T_{F\mathcal{X}} \cdot e^\partial(P) \tag{54}$$

The expression (54) represents coordinates of program P in basis F. Thus, the program is expressible as a linear combination of f_i, with components P_F as coefficients.

$$P = \sum_{i=0}^{n} P_{Fi} f_i \tag{55}$$

If F does not span $\mathcal{P}$, or we used the projection of the operator $e_{n<N}^\partial$, the expression P_F still represents the best possible approximation of the original program, on components $\{\partial^n\}$, in basis F.

It makes sense to, before computing the tensor (53), expand the set F, by mutual (nested) compositions, and gain mappings, that can not be expressed as linear combinations (but are still optimized for hardware H), and so increasing the power of the method.

[5] one choice would be the monomial basis, consisting of elements $\mathbf{e}_i \otimes \prod_{\alpha,\forall_j} x_{\alpha_j}$, where $\mathbf{e}_i$ span $\mathcal{V}$, x_i span $\mathcal{V}^*$ and α multi-index

5.4 Special case of functions $\mathcal{V} \to K$

We describe a special case when $\mathcal{P}_0 = \mathcal{V} \otimes \mathcal{P}_{-1}$, where $\mathcal{P}_{-1} < K^{\mathcal{V}}$ is a subspace of the space of functions $\mathcal{V} \to K$. This is useful, if the set of functions F only contains functions $\mathcal{V} \to K$. It is very common, that basic operations in a programming language change one single real valued variable at a time. In that case, the value of changed variable is described by the function $\tilde{f} : \mathcal{V} \to K$, while the location, where the value is saved is given by a standard basis vector $\mathbf{e}_i$. The map $f : \mathcal{V} \to \mathcal{V}$ is then given as a tensor product $f = \mathbf{e}_i \otimes \tilde{f}$. We can start the construction of the differentiable programming spaces by defining differentiable programming space of functions $\mathcal{V} \to K$ instead of maps $\mathcal{V} \to \mathcal{V}$ as in definition 42. Analog to the the Theorem 41 and Corollary 1 it is easy to verify, that

$$\partial^k \mathcal{P}_{-1} < \mathcal{P}_{-1} \otimes T_k(\mathcal{V}^*). \tag{56}$$

Since tensoring with elements of $\mathcal{V}$ commutes with differentiation operator ∂

$$\partial^k \mathcal{P}_0 < \mathcal{V} \otimes \partial^k \mathcal{P}_{-1} \tag{57}$$

and analytic virtual machine can be defined in terms of functions $\mathcal{P}_{-1}$, enabling more efficient implementation of the operators ∂ and e^{∂}. The functional transformation becomes much more efficient, since the set of functions F can be generated by the functions of the form $f = \mathbf{e}_i \otimes \tilde{f}_j$, where $F_{-1} = \{\tilde{f}_j : \mathcal{V} \to K\}$.

Theorem 55 *Suppose that $\mathcal{P}_0 = \mathcal{V} \otimes \mathcal{P}_{-1}$ where $\mathcal{P}_{-1}$ is a subspace of functions $\mathcal{V} \to K$. If $F = \{\mathbf{e}_i \otimes \tilde{f}_j\}$ and $\mathcal{X} = \{\mathbf{e}_l \otimes \tilde{p_k}\}$ be the basis of the space of polynomial maps $\mathcal{V} \to \mathcal{V}$, with $\mathcal{X}_{-1} = \{p_k\}$ being the basis of polynomial functions $\mathcal{V} \to K$. Then the matrix $T_{\mathcal{X}F}$ is block diagonal with the same block along the diagonal*

$$T_{\mathcal{X}_{-1}F_{-1}} = \sum_{k,j} p_k \otimes e_n^{\partial}(\tilde{f}_j). \tag{58}$$

Corollary 4 *When the tensor $T_{\mathcal{X}F}$ (53) of basis transformation $F \to \mathcal{X}$ is block diagonal as by Theorem 55, the tensor $T_{F\mathcal{X}} = T^{-1}_{\mathcal{X}F}$ (52) of basis transformation $\mathcal{X} \to F$ is found by simply inverting each block* (58).

Note however, that this special case can not model basic operations that change several memory locations at once, while the general model presented in this paper can. Also note that the main goal of this work is to develop methods for analysis of computer programs, making the programming spaces of maps $\mathcal{V} \to \mathcal{V}$ much more appropriate than the programming spaces of functions.

5.5 Control structures

Until now, we restricted ourselves to operations, that change the memories' content. Along side assignment statements, we know control statements (ex. statements `if`, `for`, `while`, ...). Control statements don't directly influence values of variables, but change the execution tree of the program. This is why we interpret control structures as a piecewise-definition of a map (as a spline).

Each control structure divides the space of parameters into different domains, in which the execution of the program is always the same. The entire program divides the

space of all possible parameters to a finite set of domains $\{\Omega_i; \quad i = 1, \dots k\}$, where the programs' execution is always the same. As such, a program may in general be piecewise-defined. For $\mathbf{x} \in \mathcal{V}$

$$P(\mathbf{x}) = \begin{cases} P_{n_1 1} \circ P_{(n_1-1)1} \circ \dots P_{11}(\mathbf{x}); & \mathbf{x} \in \Omega_1 \\ P_{n_2 2} \circ P_{(n_2-1)2} \circ \dots P_{12}(\mathbf{x}); & \mathbf{x} \in \Omega_2 \\ \vdots & \vdots \\ P_{n_k k} \circ P_{(n_k-1)k} \circ \dots P_{1k}(\mathbf{x}); & \mathbf{x} \in \Omega_k \end{cases} \tag{59}$$

The operator e^{∂} (at some point) of a program P, is of course dependent on initial parameters $\mathbf{x}$, and can also be expressed piecewise inside domains Ω_i

$$e^{\partial}P(\mathbf{x}) = \begin{cases} e^{\partial}P_{n_1 1} \circ e^{\partial}P_{(n_1-1)1} \circ \dots \circ e^{\partial}P_{11}(\mathbf{x}); & \mathbf{x} \in \mathrm{int}(\Omega_1) \\ e^{\partial}P_{n_2 2} \circ e^{\partial}P_{(n_2-1)2} \circ \dots \circ e^{\partial}P_{12}(\mathbf{x}); & \mathbf{x} \in \mathrm{int}(\Omega_2) \\ \vdots & \vdots \\ e^{\partial}P_{n_k k} \circ e^{\partial}P_{(n_k-1)k} \circ \dots \circ e^{\partial}P_{1k}(\mathbf{x}); & \mathbf{x} \in \mathrm{int}(\Omega_k) \end{cases} \tag{60}$$

Theorem 56 *Each program $P \in \mathcal{P}$ containing control structures is infinitely-differentiable on the domain $\Omega = \bigcup_{\forall_i} \mathrm{int}(\Omega_i)$.*

Proof Interior of each domain Ω_i is open. As the entire domain $\Omega = \bigcup_{\forall_i} \mathrm{int}(\Omega_i)$ is a union of open sets, it is therefore open itself. Thus, all evaluations are computed on some open set, effectively removing boundaries, where problems might have otherwise occurred. Theorem follows directly from the proof of Theorem 41 through argument (20).

5.6 Transformations in practice

Branching of programs into domains (59) is done through conditional statements. Each conditional causes a branching in programs' execution tree.

Claim 54 *Cardinality of the set of domains $\Omega = \{\Omega_i\}$ equals $|\{\Omega_i\}| = 2^k$, where k is the number of conditionals contained within the program.*

Remark 54 *Iterators, that do not change exit conditions within its' body, do not cause branching.*

It follows from Claim 54, that the complexity of naive implementations of methods presented in Sections 5.2 and 5.3 to piecewise-defined maps are exponential in the number of branching points (if we were to treat each domain Ω_i by itself). But, with correct application of the theorems developed in this paper, we may drastically reduce this down to linear. This is the subject of study in this section.

Theorem 57 *A program $P \in \mathcal{P}$ can be equivalently represented with at most $2n + 1$ applications of the operator e^{∂}, on $2n + 1$ analytic programs.*

Proof Source code of a program $P \in \mathcal{P}$ can be represented by a directed graph, as shown in Figure 1. Each branching causes a split in the execution tree, that after completion returns to the splitting point. By Theorem 53, each of these branches can be viewed as a program p_i, for which it holds

$$e^{\partial}(p_n \circ p_{n-1} \circ \cdots \circ p_1) = e^{\partial}(p_n) \circ e^{\partial}(p_{n-1}) \circ \cdots \circ e^{\partial}(p_1)$$

by Theorem 53.

Thus, the source code contains $2n$ differentiable branches, from its' first branching on, not counting the branch leading up to it, upon which the application of the operator e^{∂} is needed. Total of $2n+1$. By Theorem 41, each of these branches is analytic.

Claim 55 *Images of the operator e^{∂} and $T_{F\mathcal{X}}$ are elements of the original space $\mathcal{P}$, which may be composed. Thus, for $P = p_3 \circ p_2 \circ p_1$, the following makes sense*

$$P = \left(p_3 \circ e^{\partial}(p_2) \circ T_{F\mathcal{X}} e^{\partial}(p_1)\right) \in \mathcal{P} \tag{61}$$

The same holds true for all permutations of applications of operators e^{∂}, $T_{F\mathcal{X}}$ and id, as visible in Figure 1.

Remark 55 *In practice, we always use projections of the operator e^{∂} to some finite subset $\mathcal{D}^n$, resulting in e^{∂}_n. Therefore, we must take note that the following relation holds*

$$e^{\partial}_m(P_2) \circ e^{\partial}_n(P_1) = e^{\partial}_k(P_2 \circ P_1) \iff 0 \leq k \leq \min(m,n) \tag{62}$$

when composing two images of the applied operator, projected to different subspaces.

The transformation tensor $T_{F\mathcal{X}}$ is needed to be computed only once and can then on be applied to any program running on said hardware. The same holds true for each branch p_i, which can, by Theorem 2, be freely composed amongst each other.

Claim 56 *Images of the operator $e^{\partial}(P \in \mathcal{P}_0)$ are elements of $\mathcal{V} \otimes T(\mathcal{V}^*)$ by* (29)*, consisting of multilinear maps. As such, their evaluation and composition $(e^{\partial}(P_1) \circ e^{\partial}(P_2))$ is tailor made for efficient implementation by methods of parallelism [7], with computable complexities.*

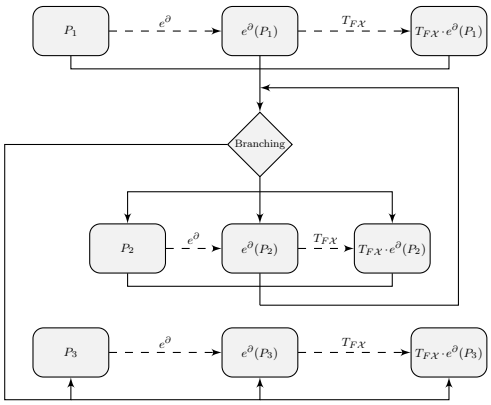

Fig. 1: Transformation diagram

Transformations by (54) need to be computed only once. Complexity of the composition of images of transformations $T_{F\mathcal{X}} \cdot e^{\partial}(p_2) \circ T_{F\mathcal{X}} \cdot e^{\partial}(p_1)$ depends on complexities of $f_i \in F$.

6 Generalized tensor networks

Let W be an element of the virtual memory $\mathcal{V} \otimes T_n(\mathcal{V}^*)$ in an analytic machine of Definition 52. We define the application of $v \in \mathcal{V}$ by

$$W(v) = \sum_{i=0}^{n} (w_i \in \mathcal{V} \otimes \mathcal{V}^{*\otimes i})(v^{\otimes i}) \in \mathcal{V} \tag{63}$$

where each contraction $(w_i \in \mathcal{V} \otimes \mathcal{V}^{*\otimes i})(v^{\otimes i})$ is defined by the programming algebra of Theorem 51.

Definition 61 *A general tensor network $\mathcal{N}$ is defined by the equation*

$$L_{i+1} = \Phi_i \circ W_i(L_i) \tag{64}$$

for each layer, where

- $L_i \in \mathcal{V}$ *is the input*
- $L_{i+1} \in \mathcal{V}$ *is the output*
- $W_i \in \bigoplus_{k=0}^{n} \mathcal{V} \otimes \mathcal{V}^{*k\otimes}$ *are the weights (and the bias)*
- $\Phi_i \in \mathcal{P}_0$ *is the activation function*

Thus, a general tensor network with n layers is

$$\mathcal{N} = L_n \tag{65}$$

Anything that operates under these specifications is a general tensor network.

Claim 61 *The common neural network defined by the equation*

$$L_{i+1} = \Phi_i(b_i + W_i(L_i))$$

is a tensor network with $W = b_i + W_i \in \mathcal{V} \oplus \mathcal{V} \otimes \mathcal{V}^*$.

Generalizations of recurrent [15], convolutional [16] and highway [17] neural networks, and features such as long short term memory [8], are easily generated by this model, as they are all elements of a differential programming space $\mathcal{P}$.

Remark 61 *Theory allows simple extensions to the concept of general tensor networks, like a convolutional filter of a convolutional layer itself being a small general tensor network instead of a simple filter. Such feats are easily accomplished, as the developed operational calculus applies to any layer $L \in \mathcal{P}$.*

This offers a broad generalization of a concept of a neural network, with a rich theory of operational calculus developed in this paper at its disposal for the study and training of such objects, as we demonstrate in Section 6.2.

Claim 62 *A general tensor network $\mathcal{N}$ can be represented and implemented by a deeper common neural network. This equivalence means, that a general tensor network with fewer layers can provide the same results as a deeper common neural network, while mitigating the vanishing gradient problem [18] that occurs in training due to the depth of the network coupled with machine precision.*

By Claim 62, existing architectures like Theano [19], TensorFlow [20] and others could be easily adapted to handle general tensor networks.

6.1 Programs as general tensor networks

Let $P = P_n \circ \cdots \circ P_0 \in \mathcal{P}_0$ be the procedure of interest, with $P_i \in \mathcal{P}_0$ being the source code between two branching points, like shown in Figure 1 of Section 5.6. By Theorem 52 we have

$$P(v_0 + v \in \mathcal{V}) = e^{\partial}|_{v_0} P(v \in \mathcal{V}) \tag{66}$$

and through Corollary 2

$$P(v_0 + v \in \mathcal{V}) = e^{\partial}|_{v_n} P_n \circ \cdots \circ e^{\partial}|_{v_0} P_0(v \in \mathcal{V}) \tag{67}$$

which is the transformation hereon denoted by $e^{\partial} P$.

Claim 63 *The image of the application of the operator e^{∂} to a program $P \in \mathcal{P}_0$ as in* (67)*, is a general tensor network, with the activation function Φ_i being the identity map, at each layer.*

6.1.1 Transformations of programs to general tensor networks

The transformed program equals the original program by Theorem 52 and Corollary 2. But in practice, we are always working with a finite virtual memory $\mathcal{V} \otimes T_n(\mathcal{V}^*)$ and the equality becomes an approximation. Thus we treat the transformation of the original program, as the initialization of the weights (and the bias) of a general tensor network to be trained.

Theorem 61 *Let $\Phi = \{\Phi_k \in \mathcal{P}_0 : \mathcal{V} \to \mathcal{V}; \quad 0 \leq k \leq n\}$ be the set of activation functions. Then the transformation*

$$\tilde{P} : \mathcal{V} \to \mathcal{V} \tag{68}$$

$$\tilde{P} = \Phi_n \circ e_N^{\partial}|_{v_n} P_n \circ \cdots \circ \Phi_0 \circ e_N^{\partial}|_{v_0} P_0 \tag{69}$$

is the transformation of a program to a general tensor network, with $e^{\partial} P_i$ being applied at a specific point $v_i = P_{i-1}(v_{i-1}) \in \mathcal{V}$.

Proof

$$\begin{gathered} e_N^{\partial}|_{v_i} P_i = W_i \in V \otimes T_N(V^*) \implies \tilde{P} = \Phi_n \circ W_n \circ \cdots \circ \Phi_0 \circ W_0 \\ \wedge \\ L_{i+1} = \Phi_i \circ W_i \circ \cdots \circ \Phi_0 \circ W_0 \implies L_{i+1} = \Phi_i \circ W_i(L_i) \\ \Longrightarrow \\ \tilde{P} = L_{n+1} \end{gathered}$$

Corollary 5 *Using the operator e_1^{∂} in the transformation of Theorem 61, we arrive at a transformation from a program to a common neural network.*

This may prove useful in future developments of the field, as currently neural networks give best results in many fields. When the original program is a sub-optimal algorithm, providing an approximate solution, as it is more often than not in practice, the derived tensor network may provide better results than the original program after training. The original program serves as a great initialization point of the general tensor network, which may be viewed as a family of algorithms to be optimized.

Remark 62 *All coefficients $W_i \in \mathcal{V} \otimes T(\mathcal{V}^*)$ are multi-linear maps allowing efficient implementation through GPU parallelism as by Claim 56. Thus, general tensor networks may employ it, just as the common neural networks they generalize do.*

6.1.2 Compositions of tensor networks with general programs

Methods of practical use presented in Section 5.6 apply to general tensor networks.

Claim 64 *By Claim 55, an arbitrary $p \in \mathcal{P} : \mathcal{V} \to \mathcal{V}$ can be composed with general tensor networks $\mathcal{N}_i : \mathcal{V} \to \mathcal{V}$*

$$P = \mathcal{N}_2 \circ p \circ \mathcal{N}_1 \in \mathcal{P} \tag{70}$$

The expression $P \in \mathcal{P}$ (70) is an element of a differentiable programming space, and can thus be treated by the operational calculus presented in this paper.

Remark 63 *Any layer $L_i \in \mathcal{P}_0$ can be composed with an arbitrary element of the differentiable programming space by Claim 64. This allows algorithmic coding of trainable memory managers, generalizing concepts such as long short term memory [8] and easing the implementation of networks capable of reading from and writing to an external memory [9], by freeing semantics of their design process.*

6.2 Training of general tensor networks

All transformed programs $\tilde{P} : \mathcal{V} \to \mathcal{V}$ are elements of a differentiable programming space $\mathcal{P}_0$. As such, the operational calculus derived in this paper, and the operators it offers, can freely be applied to them.

By Corollary 2 we have

$$e_n^{\partial}(L_{i+1}) = e_n^{\partial}(\Phi_i) \circ e_n^{\partial}(W_i \circ L_i) \in \mathcal{P}_n \tag{71}$$

Thus by Corollary 3, the n-differentiable k-th derivatives can be extracted by

$${}^{n}L_{i+1}^{k\prime} = \phi^k \circ e_{n+k}^{\partial}(L_{i+1}) \in \mathcal{P}_n \tag{72}$$

from (71) through Corollary 3, where ϕ is the reduction of order map of Theorem 54. Derivatives of specific order are extracted through projections on components of P_n, and can be used in any of the well established training methods in the industry.

Claim 65 *Using the operator $\exp(\partial_f e^{h\partial_g})$ of Theorem 53, both forward [5] and reverse [6] mode automatic differentiation (generalized to arbitrary order) can be implemented on general tensor networks, as by Claim 53.*

Remark 64 *The operational calculus developed in this paper can be applied to the training process $T \in \mathcal{P}_0$ of a general tensor network itself, as it is but an element of a differentiable programming space $\mathcal{P}$. Thus, hyper-parameters of the training process [21] can be studied, analyzed and trained [22], as to be adapted to the particulars of the problem being solved.*

By Claim 64, any training methods enabled by the operational calculus presented in this paper apply to all compositions of general tensor networks $\mathcal{N}_i$ with arbitrary programs $P \in \mathcal{P}$. This allows seamless trainable transitions between code formulations, naturally extending the Transformation diagram of Figure 1.

7 Analysis

The theory presented in this paper grants new insights through methods of analysis. It reveals a procedure to be a system exerting change on objects inhabiting virtual space. These revelations are the object of study of this section, as we demonstrate how to intertwine algorithmic control flow, operational calculus and algebra, towards a common goal.

7.1 Study of properties

We will denote the fact, that some object v has the property X, by $v \in X$. Suppose we have $v \notin X$, and desire a procedure $P \in \mathcal{P}_0$, that in some way modifies v to have the property X, changing the element v as little as possible:

$$P \in \mathcal{P}_0 : v \notin X \to P(v) \in X \tag{73}$$

Usually such procedures are difficult to construct and may not even explicitly exist. An easier task is to construct a procedure $T \in \mathcal{P}_0$, whose output allows deduction of whether v has the property X or not.

We propose an algorithm

$$A : T \in \mathcal{P} \to P \in \mathcal{P} \tag{74}$$

transforming a procedure T testing for a property X_j, to a procedure $P \in \mathcal{P}_0 : v \notin X_j \to P(v) \in X_j$ (73), imposing that property onto any object in the domain $\Omega \subset \mathcal{V}$.

We employ the developed operational calculus in such endeavors, as we probe procedures' inner structure and explore how it interacts with the elements of the domain.

7.1.1 Activity profiles and property measures

To be able to determine which conceptual steps are the most important for the result of the testing procedure T, we have to somehow measure the activity at that step. A simple example of the measure of activity $\mathcal{A}$ could simply be the norm of the appropriate derivative, as it measures the rate of change. A measure easily replaceable by a more sophisticated one, as made available by the theory.

Let

$$T = T_n \circ T_{n-1} \circ \cdots \circ T_1$$

for simplicity, but it could just as easily be piecewise-differentiable as in (59) (by a specific T_i containing a control structure, as shown in Section 5.5). Here $T_i = \varepsilon_k \circ \ldots \varepsilon_1$, is a conceptual step in the procedure.

Definition 71 (Activity profile) *The function $\mathcal{A}_i : \Omega \to [0,1]$ is called a* measure of activity *in the conceptual step T_i, where*

$$\mathcal{A}_i \circ e^{\partial_{T_i}} T(v)$$

is the activity level *of T_i, for a given element of the domain $v \in \Omega$ taken as the input of T, with only the variable parameters of the sub-procedure T_i being considered as such.*

The vector

$$\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n) : \Omega \to [0,1]^n$$

represents the activity profile of the procedure T.

Definition 72 (Property measure) *A function*

$$M_X \in \mathcal{P}_0 : [0,1]^n \to [0,1]$$

is called the property measure, measuring the amount of property X in an object $v \in \Omega$, if there exists $c \in (0,1)$ such that

$$v \in X \iff M_x \circ \mathcal{A}(v) \geq c.$$

Algorithm 1 Construct property measure

```
procedure Construct property measure
    for each v_k ∈ Ω do
        for each T_i do
            extract a_k^i = A_i ∘ e^{∂_{T_i}} T(v_k)
        end for
    end for
    initialize set I
    for each X_j do
        generate property measure M_{X_j} from a_k^i
        add M_{X_j} to I
    end for
    return I
end procedure
```

Once the activity profiles and property measures are generated, starting with an element, without the property X_j, we can use any established optimization technique, to optimize and increase the measure M_{X_j}. When the increase of the measure M_{X_j} is sufficient, this results in an new object possessing the property X_j.

Theorem 71 *With an appropriate choice of the activity profile $\mathcal{A}$, there exist an algorithm*

$$A : T \in \mathcal{P} \to P \in \mathcal{P} \tag{75}$$

transforming a procedure $T \in \mathcal{P}$, testing for a property X_j, to a procedure P, such that it increases the property measure of any object in the domain.

Corollary 6 *By Theorem 71, existence of a procedure testing an object for validity is sufficient for constructing a procedure transforming a non-valid object to a valid one*

$$P \in \mathcal{P}_0 : v \notin X_j \to P(v) \in X_j$$

under the assumption that the increase of the property measure is sufficient.

When T serves as a simulation, with v modeling a real-life object of study, the procedure opens new doors in analyzing real-life phenomena. Through it, we may observe how v evolves through iterations and study stages of change, which serves as a useful insight when designing procedures causing change in real-life phenomena.

Algorithm 2 Appoint property X_j to $v \in \Omega$

procedure APPOINT PROPERTY X_j TO $v \in \Omega$
  initialize path γ with v
  for each step **do**
    for each $T_i \in T_{X_j}$ **do**
      extract $T_i' = \phi \circ e_2^{\partial_v}(T_i \circ \cdots \circ T_1) \in \mathcal{P}_1$
      compute the energy $E_i = e_1^{\partial_v}(M_{X_j}) \circ T_i' \in \mathcal{P}_1$
      extract the derivative $\partial_v E_i = proj_{\{\partial\}}(E_i)$
      add $\partial_v E_i$ to $\partial_v E$
    end for
    update v by $step(\partial_v E, v)$
    insert v to γ
  end for
  return γ
end procedure

7.1.2 Example

The derived procedures P, are a broad generalization of special cases such as Google's Deep Dream project [10], where the program $T \in \mathcal{P}_0$ represents a neural network, with the resulting $v \in X_j$ being an altered image.

Remark 71 *Approach is trivially generalized to general tensor networks.*

We take the measure of activity to be the norm of the derivative with respect to the variable parameters in sub-procedure T_i.

$$\|\partial_{T_i} T\| = \left\| proj_{\{\partial\}} \left(e_1^{\partial_{T_i}}(T) \right) \right\|$$

Then for each property X_j we select the set of sub-procedures

$$T_{X_j} = \{T_i \in T_{X_j} \iff \|\partial_{T_i} T\| \geq c\}$$

that have the highest measure of activity at the elements of X_j. The property measure for the property X_j is then simply the sum of squares of norms of the derivatives of selected sub-procedures $T_1^i = T_i \circ \cdots \circ T_1$. This completes Algorithm 1.

By Corollary 3, n-differentiable k-th derivatives ${}^n T_i^{k\prime} \in \mathcal{P}_n$ with respect to the input $v \in \Omega$ are computed by

$${}^n T_i^{k\prime} = \phi^k \circ e_{n+k}^{\partial_v}(T_1^i) \in \mathcal{P}_n \tag{76}$$

where ϕ is the reduction of order map of Theorem 54. Thus, using $\|\cdot\| \in \mathcal{P}_1$ as the norm map, the property measure is

$$M_{X_j} = \sum_{T_i \in T_{X_j}} \left\| T_i' \right\|^2 \in \mathcal{P}_1 \tag{77}$$

assuming it only needs to be once differentiable. Optimization of the property measure completes Algorithm 2.

When $T \in \mathcal{P}_0$ represents a neural network, T_i stands for a specific layer in the network, with neurons being its variable parameters, than the procedure achieves what Google's Deep Dream Project [10] does. However our procedure may be applied to any program $T \in \mathcal{P}_0$. As neural networks are just programs contained in the programming space $\mathcal{P}$, our view gazes over a broader landscape of programs it can be utilized on.

8 Conclusions

Existence of a program is embedded in a virtual reality, forming a system of objects undergoing change in a virtual space. Just as the reality inhabited by us is being studied by science, revealing principles and laws, so can the virtual reality inhabited by programs. Yet here lies a tougher task, as the laws of the system are simultaneously observed and constructed; the universe is bug-free, up to philosophic precision, while our programs are not. This reinforces the need for a language capable of not only capturing, but also constructing digital phenomena, a feat demonstrated by the theory presented in this paper.

Like Feynman [23] and Heaviside [24] for physics before us, we developed operational calculus on programming spaces, allowing analytic conclusions through algebraic means, easing implementation. We derived the operator of program composition, generalizing both forward [5] and reverse [6] mode of automatic differentiation to arbitrary order, under a single operator in the theory. Both the use of algebra and operational calculus were demonstrated upon the operator, as calculations and manipulations were performed on the operator level, before the operator is applied to a particular program. This language condenses complex notions into simple expressions, enabling formulation of meaningful algebraic equations to be solved. Doing so, we derived functional transformations of programs in arbitrary function basis', a useful tool when adapting code to the specifics of a given hardware, among other. All such formulations are invariant not only to the choice of a programming space, but also to the point in execution of a program, introducing the principle of general covariance [14] to programming. Offerings of this principle were exploited, as we designed methods on how transformations are to be interchangeably applied in practice in Section 5.6. These methods allow for seamless transitions between transformed forms and original code throughout the program. All analytic machines attain expressive freedom of algorithmic control flow, as they fully integrate control structures. Employing them, we can construct analytic procedures, viewing algorithms in a new light. One can start incorporating variable parameters into algorithm design, revealing the true nature of hyper-parameters often used in practice.

In Section 6, we generalized neural networks by constructing general tensor networks and showed them to be differentiable and trainable using operators constructed in Section 5. We derived transformations of arbitrary programs to general tensor networks and provided a meaningful way of initializing networks. Tensor networks can be composed with any program in a differential programming space, allowing algorithmic constructions of layers performing specific task, like memory management [8][9]. All such constructs are trainable and adhere to the operational calculus presented in this paper. This may prove useful in future developments of the field, as neural networks currently give best results to many problems. To this aim, we developed tools for analyzing procedures and probe their inner structure. State of the art methods for analyzing neural networks were generalized to general programs in the theory. We demonstrated how the developed operational calculus is employed in practice towards such a cause, and how through it countless methods from analysis can be trivially transfered to the subject, limited only by the imagination and inventiveness of the employer of the theory.

References

1. M. Girolami, B. Calderhead, Riemann manifold Langevin and Hamiltonian Monte Carlo methods, Journal of the Royal Statistical Society.
2. G. Corliss, Automatic differentiation of algorithms: from simulation to optimization, Vol. 1, Springer Science & Business Media, 2002.
3. B. A. Pearlmutter, et al., Putting the Automatic Back into AD: Part I, What's Wrong (CVS: 1.1), ECE Technical Reports.
4. B. A. Pearlmutter, et al., Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast (CVS: 1.1), ECE Technical Reports.
5. K. A. Khan, P. I. Barton, A vector forward mode of automatic differentiation for generalized derivative evaluation, Optimization Methods and Software 30 (6) (2015) 1185–1212. doi:10.1080/10556788.2015.1025400.
6. R. J. Hogan, Fast reverse-mode automatic differentiation using expression templates in c++, ACM Trans. Math. Softw. 40 (4) (2014) 26:1–26:16. doi:10.1145/2560359.
   URL http://doi.acm.org/10.1145/2560359
7. A. A. et al., High-performance tensor contractions for {GPUs}, Procedia Computer Science 80 (2016) 108 – 118, international Conference on Computational Science 2016, {ICCS} 2016, 6-8 June 2016, San Diego, California, {USA}. doi:http://dx.doi.org/10.1016/j.procs.2016.05.302.
8. S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (8) (1997) 1735–1780.
9. A. Graves, et al., Hybrid computing using a neural network with dynamic external memory, Nature advance online publication, article.
10. A. Mordvintsev, et al., Inceptionism: Going deeper into neural networks (2015).
    URL https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html
11. Žiga Sajovic, dcpp (2016).
    URL https://github.com/zigasajovic/dCpp
12. Žiga Sajovic, Implemetation of operational calculus on programing spaces with applications (2016).
    URL https://zigasajovic.github.io/dCpp/paper/dCpp.pdf
13. N. Wiener, The operational calculus (1926).
    URL http://gdz.sub.uni-goettingen.de/en/dms/loader/img/?PID=GDZPPN002270846&physid=PHYS_0561
14. O'Hanian, H. C., Ruffini, Remo, Gravitation and Spacetime (2nd ed.), W. W. Norton, 1994.
15. R. Socher, et al., Parsing natural scenes and natural language with recursive neural networks (2011).
16. A. Krizhevsky, et al., Imagenet classification with deep convolutional neural networks, in: Advances in neural information processing systems, 2012, pp. 1097–1105.
17. R. K. Srivastava, et al., Training very deep networks, in: Advances in neural information processing systems, 2015, pp. 2377–2385.
18. R. Pascanu, T. Mikolov, Y. Bengio, On the difficulty of training recurrent neural networks., ICML (3) 28 (2013) 1310–1318.
19. Theano Development Team, Theano: A Python framework for fast computation of mathematical expressions, arXiv e-prints abs/1605.02688.
    URL http://arxiv.org/abs/1605.02688
20. M. Abadi, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org (2015).
    URL http://tensorflow.org/
21. Y. Bengio, Gradient-based optimization of hyperparameters, Neural computation 12 (8) (2000) 1889–1900.
22. C. Thornton, et al., Auto-weka: Combined selection and hyperparameter optimization of classification algorithms, in: Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2013, pp. 847–855.
23. R. P. Feynman, An operator calculus having applications in quantum electrodynamics, Phys. Rev. 84 (1951) 108–128. doi:10.1103/PhysRev.84.108.
24. J. R. Carson, The heaviside operational calculus, Bell System Technical Journal 1 (2) (1922) 43–55. doi:10.1002/j.1538-7305.1922.tb00388.x.
25. Žiga Sajovic, M. Vuk, Operational calculus on programming spaces and generalized tensor networks (2016). arXiv:1610.07690.