Implementation and Employment of an Analytic virtual machine

Žiga Sajovic*

October 31, 2016

Abstract

We provide an illustrative implementation of an analytic, infinitely-differentiable virtual machine, implementing infinitely-differentiable programming spaces and operators acting upon them, as constructed in the paper *Operational calculus on programming spaces and generalized tensor networks*[1]. Implementation closely follows theorems and derivations of the paper, intended as an educational guide.

Analytic virtual machines allow nested transformations, with operational calculus opening new doors in program analysis. We outline the process of employing such a machine to several causes, seamlessly interweaving operational calculus and algorithmic control flow.

Nothing at all takes place in the universe in which some rule of maximum or minimum does not appear.

— Leonhard Euler

*ziga.sajovic@xlab.si

Contents

1 Introduction **3**

2 Virtual memory **3**
2.1 Initialization 4
2.2 Algebra over a field 4
2.2.1 Vector space over a field K 5
2.2.2 Algebra over a field K 6

3 Analytic virtual machine **7**
3.1 Operators 8
3.2 Differentiable programming space 9
3.2.1 Example 9
3.2.2 Order reduction for nested applications 11
3.3 External libraries 12

4 Employment **13**
4.1 General engineering 13
4.1.1 Training as a differentiable program 14
4.2 Analysis 14
4.2.1 Study of properties 14
4.2.2 Discovering redundancy 14

1 Introduction

We provide an illustrative implementation of the virtual memory $\mathcal{V}$ and its expansion to $\mathcal{V} \otimes T(\mathcal{V}^*)$, serving by itself as an algebra of programs along with an infinitely differentiable programming space [1, Theorem 5.1]

$$\mathcal{D}^n C{+}{+} < \mathcal{P}_n : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*) \tag{1}$$

acting on it.

We provide a construction of operators

$$\mathcal{D}^n = \{\partial^k; \quad 0 \leq k \leq n\} \tag{2}$$

allowing derivation of the operator of tensor series expansion [1, Theorem 5.4]

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*), \tag{3}$$

and projecting it onto the unit n-cube, deriving the operator increasing the order of a differentiable programming space [1, Corollary 5.2]

$$\tau_n : \mathcal{P}_k \to \mathcal{P}_{n+k} \tag{4}$$

Analytic virtual machines allow nested transformations, with operational calculus opening new doors in program analysis. We outline the process of employing such a machine to several causes, from engineering to discovering model redundancy, seamlessly interweaving operational calculus and algorithmic control flow. Source code can be found on GitHub [2].

2 Virtual memory

We model an element of the virtual memory $v \in \mathcal{V}_n$ [1, Claim 4.1]

$$\mathcal{V}_n = \mathcal{V}_{n-1} \oplus (V_{n-1} \otimes \mathcal{V}^*) \tag{5}$$

$$\mathcal{V}_n = \mathcal{V} \oplus \mathcal{V} \otimes \mathcal{V}^* \oplus \cdots \oplus \mathcal{V} \otimes \mathcal{V}^{*n\otimes} \implies \mathcal{V}_n = \mathcal{V} \otimes T(\mathcal{V}^*) \tag{6}$$

with the class *var*.

```
template<class V>
class var
{
    public:
        int order;
        V id;
        std::shared_ptr<std::map<var*,var> >* dTau;

        var();
        var(V id);
        var(const var& other);
        ~var();
        void init(int order);
        var d(var* dVar);
        //declerations of algebraic operations
        //declerations of order logic
};
```

The expanded virtual memory $\mathcal{V}_n$ is the tensor product of the virtual memory $\mathcal{V}$ with the tensor algebra of its dual. Tensor products of the virtual memory $\mathcal{V}$ with its dual are modeled using maps, denoted by *dTau*. Naming reflects how the algebra is constructed, mimicking the operator τ_n (4). The address *var** stands for the component of $v \in \mathcal{V}_{n-1}$ on which the tensor product with the component of $v^* \in \mathcal{V}^*$ was computed to generate $v \in \mathcal{V}_n$ in equation (5). This depth is contained in the *int order*.

2.1 Initialization

A constant element v_0 of the virtual memory is an element of $\mathcal{V}_0 = \mathcal{V} < \mathcal{V}_n$. We initialize an element to be n-differentiable, by mapping

$$init : \mathcal{V} \times \mathbb{N} \to \mathcal{V}_n \tag{7}$$

$$v_0.init(n) = v_n \in \mathcal{V}_n \tag{8}$$

The image v_n is an element of $\mathcal{V}_1 = \mathcal{V} \otimes \mathcal{V}^*$ with natural inclusion in $\mathcal{V}_n$.

$$v_n = (v_0 \in \mathcal{V}) + (\delta^i_j \in \mathcal{V} \otimes \mathcal{V}^{*\otimes}) + \sum_{i=2}^{n} (0 \in \mathcal{V} \otimes \mathcal{V}^{*i\otimes}) \tag{9}$$

where δ^i_j is the identity.

2.2 Algebra over a field

Algebra over a field is a vector space equipped with a bilinear product. Thus, an algebra is an algebraic structure, which consists of a set, together with operations of multiplication, addition, and scalar multiplication by elements of the underlying field. [3, p. 3]

Algebra is constructed by mimicking application of a direct sum of the operators τ_n (4) mapping $\mathcal{P}_0 \oplus \mathcal{P}_0 \to \mathcal{P}_n$ to the maps of scalar multiplication, addition and a bilinear product. This is reflected in the structure of the class *var* modeling the elements of the virtual space $v \in \mathcal{V}_n$.

Theorem 2.1. *An instance of the class var is an element of the virtual memory* $\mathcal{V}_n$.

$$var \in \mathcal{V}_n \tag{10}$$

Proof.

$$id \in \mathcal{V} \wedge dTau \in \mathcal{V}_{n-1} \tag{11}$$

$$\wedge$$

$$var = id \oplus dTau \implies var \in \mathcal{V}_n \tag{12}$$

□

Compositions are modeled by mimicking the operator of program composition [1, Theorem 5.6]

$$\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \to \mathcal{P}_\infty \tag{13}$$

generalizing both forward and reverse mode automatic differentiation [1, Claim 5.3]. By fixing one of the mappings in (13), we derive projections of the pullback operator

$$\exp(\partial_f e^{h\partial_g})(g) : \mathcal{P} \to \mathcal{P}_\infty(g) \tag{14}$$

to the unit hyper-cube and applying it to the resulting direct sum.

2.2.1 Vector space over a field K

We begin our construction of an algebra, by constructing a vector space $\mathcal{V}_n$ over a field K. A vector space is collection of objects that can be added together and multiplied with elements of the underlying field K, .

We begin with scalar multiplication.

```
template<class K>
var var::operator*(K n)const{
    var out;
    out.id=this->id*n;
    for_each_copy(...,mul_make_pair<pair<var*,var> >, n);
    return out;
}

template<class K>
var var::operator/(K n)const {...};
```

Scalar multiplication and its convenient inverse employ the function *for_ each_ copy*, applying the provided operation

```
template<class V, class K>
V mul_make_pair(V v, K n) {
    return std::make_pair(v.first, v.second * n);
}
```

to each one of the components of *this* and storing the result in $out.dTau$.

Vector addition by component is implemented by

```
var var::operator+(const var& v)const{
    var out;
    out.id=this->id+v.id;
    merge_apply(..., sum_pairs<pair<$var, var> >);
    return out;
}
```

Vector addition by component employs the function *merge_ apply*, applying the provided function *sum_ pairs*

```
template<class V>
T sum_pairs(V v1, V v2) {
    return std::make_pair(v1.first, v1.second + v2.second);
}
```

to corresponding components, storing the result in $out.dTau$, in $\mathcal{O}(n\log(n))$.

Theorem 2.2. *Class var models a vector space over a field K.*

Proof. By implementations of addition by components and multiplication with a scalar $k \in K$, the axioms of the vector space are satisfied. □

2.2.2 Algebra over a field K

With the vector space constructed, we turn towards its elevation to an algebra. To construct an algebra over a field K, we equip the vector space $\mathcal{V}_n$ with a bilinear product by components.

```
var var::operator*(const var& v)const{
    var out;
    out.id=this->real*v.id;
    out.order=this->order<v.order?this->order:v.order;
    if(out.order>0){
        map<int,double> tmp1;
        map<int,double> tmp2;
        for_each_copy(...,mul_make_pair<pair<var*,var> >,
          v.reduce());
        for_each_copy(...,mul_make_pair<pair<var*,var> >,
          this->reduce());
        merge_apply(..., sum_pairs<pair<var*,var> >);
    }
    return out;
}
```

The employed functions have been explained at previously usage. The *reduce* function makes a shallow copy of *this*, while reducing the order of the returned copy. This bilinear product contains Leibniz rule within its structure, as the projection of the operator $\exp(\partial_f e^{h\partial_g})(g) : \mathcal{P} \to \mathcal{P}_\infty(g)$ (14) to the unit n-cube was applied to the algebra.

Theorem 2.3. *Class var models an algebra over a field K.*

Proof. By the implementation of a bilinear product by components the axioms of an algebra over a field are satisfied. □

For ease of expression we implement exponentiation, naturally existing in the algebra

```
var var::operator^(double n) const{
    var out;
    out.id=std::pow(this->real,n);
    out.order=this->order<v.order?this->order:v.order;
    if(n>0&&out.order>0){
        for_each_copy(...,mul_make_pair<pair<var*,var> >,
          (this->reduce()^(n-1))*=n);
    }
    return out;
}
```

employing the function *for_each_copy*, applying the provided operation to each component.

We may now trivially implement operators existing in the algebra.

```
var var::operator-(const var& v)const{
    return *this+(-1)*v;
}

var var::operator/(const var& v)const{
    return *this*(v^(-1));
}
```

Similarly, the following operators can be assumed to be generated by the existing algebra, implementation of which is omitted here for brevity.

```
var operator*(double n)const;
var operator+(double n)const;
var operator-(double n)const;
var operator/(double n)const;
var operator^(double n) const;
var& operator=(const var& v);
var& operator+=(const var& v);
var& operator-=(const var& v);
var& operator*=(const var& v);
var& operator/=(const var& v);
var& operator*=(double n);
var& operator/=(double n);
var& operator+=(double n);
var& operator-=(double n);
var operator*(double n, const var& v);
var operator+(double n, const var& v);
var operator-(double n, const var& v);
var operator/(double n, const var& v);
var operator^(double n, const var& v);
```

Order logic is trivially implemented, by mapping

$$\mathcal{V}.id \times \mathcal{V}.id \to \{0, 1\} \tag{15}$$

```
bool operator==(const var& v)const;
bool operator!=(const var& v)const;
bool operator<(const var& v)const;
bool operator<=(const var& v)const;
bool operator>(const var& v)const;
bool operator>=(const var& v)const;
```

3 Analytic virtual machine

Definition 3.1 (Analytic virtual machine)**.** *The tuple $M = \langle \mathcal{V}, \mathcal{P}_0 \rangle$ is an analytic, infinitely differentiable virtual machine.*

- *$\mathcal{V}$ is a finite dimensional vector space*
- *$\mathcal{V} \otimes T(\mathcal{V}^*)$ is the virtual memory space, serving as alphabet symbols*
- *$\mathcal{P}_0$ is an analytic programming space over $\mathcal{V}$*

When $\mathcal{P}_0$ is a differentiable programming space, this defines an infinitely differentiable virtual machine. [1, Defintion 5.4]

3.1 Operators

The operator of tensor series expansion [1, Theorem 5.4]

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*), \tag{16}$$

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{(h\partial)^n}{n!} \tag{17}$$

is evaluated at $h = 1$ and projected onto the unit N-cube, arriving at

$$\tau_N = 1 + \partial + \partial^2 + \ldots + \partial^N, \tag{18}$$

which is used to implement $\mathcal{D}^n C{+}{+} \subset \mathcal{P}_n$, employing the recursive relation for increasing the order of $C{+}{+}$ [1, Corollary 5.2].

$$\tau_{k+1} = 1 + \partial\tau_k, \tag{19}$$

$$\tau_{k+1} C{+}{+} = C{+}{+} + \partial\tau_k C{+}{+}. \tag{20}$$

The *double id* stands for the identity operator 1 in the expression and the *map<var*, var> dTau* for the operator $\partial\tau_k$.

Compositions of these operators is achieved by mimicking the generalized pullback operator [1, Claim 5.3]

$$\exp(\partial_f e^{h\partial_g})(g) : \mathcal{P} \to \mathcal{P}_\infty(g), \tag{21}$$

projected onto the unit N-cube.

Remark 3.1. *Implementation in this paper is intended to be simply understood and is written as such. But, with the existing algebra and operational calculus, one could easily implement the operator* $\exp(\partial_f e^{h\partial_g})(g)$ *by any of the efficient techniques available, as it is given by a generating function.*

```
template<class dTau, class K>
class tau
{
public:
    tau();
    tau(K mapping, dTau primitive);
    ~tau();
    var operator()(const var&v);
private:
    dTau primitive;
    K mapping;
};
```

```
var tau::operator()(const var&v){
    var out;
    out.id=mapping(v.id);
    for_each_copy(...,mul_make_pair<std::pair<var*,var> >,
      primitive(v.reduce()));
    return out;
}
```

3.2 Differentiable programming space

With the algebra over $\mathcal{V}_n$ implemented, we turn to the construction of a differentiable programming space

$$C{+}{+} : \mathcal{V} \to \mathcal{V} \tag{22}$$

Definition 3.2. *A* differentiable programming space $\mathcal{P}_0$ *is any subspace of* $\mathcal{F}_0$ *such that*

$$\partial\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(V^*) \tag{23}$$

When all elements of $\mathcal{P}_0$ *are analytic, we denote* $\mathcal{P}_0$ *as an* analytic programming space. *[1, Definition 5.1]*

Theorem 3.1. *Any differentiable programming space* $\mathcal{P}_0$ *is an infinitely differentiable programming space, such that*

$$\partial^k\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(V^*) \tag{24}$$

for any $k \in \mathbb{N}$. *[1, Theorem 5.1]*

Thus, in order to have a differential programming space, we must provide closure under the differential operator [1, Corollary 5.1], for the function space $C{+}{+}$, expanded by the tensor product with the tensor algebra of the dual of the virtual memory $\mathcal{V}$.

$$\mathcal{D}^n C{+}{+} < \mathcal{P}_n \iff \mathcal{D}^n C{+}{+} \subset C{+}{+} \otimes T(\mathcal{V}^*) \tag{25}$$

Claim 3.1. *Any library implementing functions acting on variables of type* $double$ *or* $float$*, could be trivially included into* $C{+}{+} < \mathcal{P}_0$*, simply by replacing all variables with the class* var*. Moreover, any implementations using the implemented algebra in its construction of functions, are contained in* $\mathcal{P}_0$*.*

3.2.1 Example

As an illustrative example, we provide a simple implementation of a differentiable programming space *dCpp* through the use of the operator *tau*. Assume the existence of functions

$$\sin\_double : double \to double \tag{26}$$

$$\cos\_double : double \to double \tag{27}$$

$$e\_double : double \to double \tag{28}$$

$$\ln\_double : double \to double \tag{29}$$

filling the set spanning $dCpp < C{+}{+}$. Note, that these functions are usually implemented using operations existing in the algebra on $\mathcal{V}_n$ constructed in Section 2.2. Thus by employing it in their construction (coding), they would have been elements of a differentiable programming space, as by Claim 3.1. Here we demonstrate how to explicitly construct them as maps.

Previous declarations of needed functions are assumed.

```
namespace dCpp{
    var sin(const var& v);
    tau cos;
    tau e;
    tau ln;
    var cos_primitive(const var& v);
    var ln_primitive(const var& v);
    var e_primitive(const var& v);
}
```

We construct the map *sin* explicitly for educational purposes

```
var dC::sin(const var& v){
    var out;
    out.id=std::sin_double(v.id);
    out.order=v.order;
    if(v.order>0){
        for_each_copy(...,mul_make_pair<std::pair<var*,var> >,
                cos(v.reduce()));
    }
    return out;
}
```

Other maps are constructed through employment of the operator *tau*.

```
typedef var (*dTau)(var);
typedef double (*mapping)(double);

var dCpp::cos_primitive(cont $var v){
    return (-1)*sin(v);
}

dCpp::cos=tau<dTau,mapping>(cos_double,cos_primitive);

var dCpp::ln_primitive(cont $var v){
    return 1/v;
}

dCpp::ln=tau<dTau,mapping>(ln_double,ln_primitive);

dCpp::e_primitive(cont $var v){
    return e(v);
}

tau dCpp::e=tau<dTau,mapping>(e_double,e_primitive);
```

Theorem 3.2. *The programming space dCpp is a differentiable programming space satisfying*

$$dCpp < \mathcal{P}_0 \iff \mathcal{D}dCpp \subset dCpp \otimes T(\mathcal{V}^*) \tag{30}$$

Proof.

$$\forall_{\Phi_i \in dCpp} \exists_{\Phi_j \in dCpp} (\Phi_i.primitive = \Phi_j) \tag{31}$$

$$\Longrightarrow$$

$$\mathcal{D}dCpp \subset dCpp \otimes T(\mathcal{V}^*) \tag{32}$$

□

Corollary 3.1. *The programming space dCpp is an infinitely-differentiable programming space satisfying*

$$dCpp < \mathcal{P}_0 \iff \mathcal{D}^n dCpp \subset dCpp \otimes T(\mathcal{V}^*) \tag{33}$$

Proof. Follows directly from Theorem 3.2 by Theorem 3.1. □

Corollary 3.2. *The tuple*

$$M = \langle \mathcal{V}, dCpp \rangle \tag{34}$$

is an Analytic virtual machine.

3.2.2 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. [1, Section 5.2.2]

Theorem 3.3. *There exists a reduction of order map $\phi : \mathcal{P}_n \to \mathcal{P}_{n-1}$, such that the following diagram commutes*

$$\begin{array}{ccc} \mathcal{P}_n & \xrightarrow{\phi} & \mathcal{P}_{n-1} \\ \downarrow{\scriptstyle \partial} & & \downarrow{\scriptstyle \partial} \\ \mathcal{P}_{n+1} & \xrightarrow{\phi} & \mathcal{P}_n \end{array} \tag{35}$$

satisfying

$$\forall_{P_1 \in \mathcal{P}_0} \exists_{P_2 \in \mathcal{P}_0} \Big(\phi^k \circ \tau_n(P_1) = \tau_{n-k}(P_2) \Big) \tag{36}$$

for each $n \geq 1$.

Corollary 3.3. *By Theorem 3.3, n-differentiable k-th derivatives of a program $P \in \mathcal{P}_0$ can be extracted by*

$${}^n P^{k\prime} = \phi^k \circ \tau_{n+k}(P) \in \mathcal{P}_n \tag{37}$$

Remark 3.2. *Theorem 3.3 and Corollary 3.3 are derived by using projections onto the unit hyper-cube on [1, Theorem 5.7] and [1, Corollary 5.5].*

We construct a reduction of order map $d : \mathcal{P}_n \times \mathcal{V} \to \mathcal{P}_{n-1}$,

```
var var::d(var* dvar){
    return (*this->dTau.get())[dvar];
}
```

returning $n-1$ differentiable derivative of v with respect to v_i. Explicitly expressing $v \in \mathcal{V}_n$ in terms of $v.d(\&v_i)$

$$v = v.id + \sum_{\forall_i} v.d(\&v_i) \otimes dv_i \in \mathcal{V}_n \implies v.d(\&v_i) \in \mathcal{V}_{n-1} \tag{38}$$

reveals the nature of the reduction of order map.

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 4, as we analyze procedures as systems inducing change upon objects in virtual space.

3.3 External libraries

Any *C++* library written in the generic paradigm employing templates is fully compatible with the differentiable programming space *dCpp* acting on the virtual memory $\mathcal{V}$.

We illustrate on the example of Eigen [5]. We will code a perceptron with sigmoid activations, followed by softmax normalization, taking 28x28 image as an input and outputting a 10 class classifier. Existence of needed, but trivial and intuitively understood mappings is assumed.

```
template <typename Derived>
    void softmax(Eigen::MatrixBase<Derived>& matrix){
            //maps each element of the matrix by y=e^x;
            dCpp::map_by_element(matrix,&dCpp::e);
            //sums the elements of the matrix using Eigens function
            var tmp=matrix.sum();
            //divides each element by the sum
            for (size_t i=0, nRows=matrix.rows(),
                 nCols=matrix.cols(); i<nCols; ++i)
                 for (size_t j=0; j<nRows; ++j){
                     matrix(j,i)/=tmp;
                 }
}

int main(){
//order of derivatives needed
int order=...;
//    Matrix holding the inputs (imgSizeX1 vector)
const int imgSize=28*28;
const Eigen::Matrix<var,1,imgSize>input=Eigen::Matrix<var,1,
        imgSize>::Random(1,imgSize);
const int numOfOutOnFirstLevel=10;
//    matrix of weights on the first level
//    (imgSizeXnumOfOutOnFirstLevel)
Eigen::Matrix<var,imgSize,numOfOutOnFirstLevel>firstLayerVars=
        Eigen::Matrix<var,imgSize,numOfOutOnFirstLevel>::
        Random(imgSize,numOfOutOnFirstLevel);
//    initializing weights
dCpp::init(firstLayerVars, order);
//    mapping of the first layer --> resulting in 10x1 vector
Eigen::Matrix<var,numOfOutOnFirstLevel,1>firstLayerOutput=
        input*firstLayerVars;
//    apply sigmoid layer --> resulting in 10x1 vector
dCpp::map_by_element(firstLayerOutput,&dCpp::sigmoid);
//    apply sofmax layer --> resulting in 10x1 vector
softmax(firstLayerOutput);
//retrieve the computed derivatives
```

4 Employment

The algebra over programs [1, Theorem 5.3] an Analytic virtual machine enables and its operational calculus are employed towards achieving analytic conclusions through algebraic means, seamlessly interweaving operational calculus and algorithmic control flow.

4.1 General engineering

We demonstrate employment on the case of highway and railway design, where author has previously used the described approach. Demonstration is presented in an easily transfered general form.

Highway and railway designers use clothoid splines (planar G1 curves consisting of straight line segments, circular arcs, and clothoid segments) as center lines in route location [6]. These curves are usually computed algorithmically, using programs $P \in \mathcal{P}_0$ and are used in other programs to compute various quantities. We focus on this inclusion.

Assume existence of a program mapping parameters p_i and a point x_i to a point y_i on a clothoid.

$$cloth : p_i \times x_i \to y_i \tag{39}$$

Connecting two curves γ_i via a clothoid, matching derivatives at endpoints is a solved problem [6, Theorem 2]. In practice, algorithms solving it use derivatives of *cloth* in its body. We provide it in terms of our implementation for clarity and use in the coming section. For simplicity, all parameters of *cloth* are denoted by p_i, with *step* being the employed optimization step using k-th derivatives in its body.

Algorithm 1 Train

```
1: procedure TRAIN(n,k)
2:     initialize P_i = var(p_i.id) ∈ V_n
3:     for each step do
4:         re-initalize p_i = var(P_i.id) ∈ V_{n+k}
5:         extract derivatives ∂^k C_{p_i} = cloth(p_i).d^k(&p_i) ∈ P_n
6:         update parameters step(∂^k C_{p_i}, P_i) ∈ P_n
7:     end for
8:     return P_i ∈ V_n
9: end procedure
```

Remark 4.1. *Note that the derivatives $\partial^k C_{p_i}$ are n-differentiable as by Corollary 3.3, satisfying*

$$P_i \in \mathcal{V}_n \iff \partial^k C_{p_i} \in \mathcal{P}_n \tag{40}$$

This will hold meaning in the coming section. The p_i denote all parameters, including the end-points.

4.1.1 Training as a differentiable program

The output of Algorithm 1 are the required parameters P_i, with the algorithm itself being an element of $\mathcal{P}_n : \mathcal{V} \to \mathcal{V} \otimes T_n(\mathcal{V}^*)$, making it a n-differentiable procedure. This becomes useful, as usually the output of Algorithm 1 (connecting many sections) is only a part of an algorithm evaluating the resulting spline. Optimizing this evaluation is the main desire.

We assume the end-points to be allowed to vary over a limited domain, and are determined by a program mapping some subset of the parameters p_i (usually current end-points) of the curves forming the spline, to the set of adjusted end-points.

$$endP : \{p_i\} \to \{p_i\} \tag{41}$$

Simulation of the traversal of the resulting spline is used to accumulate its evaluation.

$$eval : \{p_i\} \to K \tag{42}$$

The desire is maximizing the evaluation, using *step* as an optimization step using n-th derivatives in its body.

Algorithm 2 Increase

```
1: procedure INCREASE(n,k)
2:     initialize p_i ∈ V_{n+k}
3:     for each step do
4:         update end-points endP(p_i)
5:         update p_i = Train(n + k, k) ∈ P_n (Algorithm 1)
6:         extract derivatives ∂^n C_{p_i} = eval(p_i).d^n(&p_i) ∈ P_0;
7:         update step(∂^n C_{p_i}, p_i) ∈ P_0
8:     end for
9: end procedure
```

Remark 4.2. *Note that the derivatives $\partial^n C_{p_i}$ are not differentiable, as they need not to be, as opposed to the derivatives used in Algorithm 1, as by Remark 4.1.*

Both algorithms can be generalized to other problems and were written as such. Mappings are trivially replaced, as are the parameters, due to the nature of their construction.

4.2 Analysis

4.2.1 Study of properties

4.2.2 Discovering redundancy

References

[1] Žiga Sajovic and Martin Vuk. *Operational calculus on programming spaces and generalized tensor networks*. 2016. arXiv: 1610.07690. URL: https://arxiv.org/abs/1610.07690.

[2] Žiga Sajovic. *dCpp*. 2016. URL: https://github.com/zigasajovic/dCpp.

[3] Michiel Hazewinkel et al. *Algebras, Rings and Modules*. 978-1-4020-2691-1, 2004. ISBN: 978-0130047632.

[4] Barak A. Pearlmutter and Jeffrey M Siskind. "Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast (CVS: 1.1)". In: *ECE Technical Reports*. (May 2008).

[5] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[6] Meek D. S. and Walton D. J. "Clothoid spline transition spirals". In: *Mathematics of computation* (1992). URL: http://www.ams.org/journals/mcom/1992-59-199/S0025-5718-1992-1134736-8/S0025-5718-1992-1134736-8.pdf.