$\mathcal{D}^n C_{++}$

Žiga Sajovic

October 19, 2016

Abstract

We provide an illustrative implementation of an analytic, infinitely-differentiable 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*. Implementation closely follows theorems and derivations of the paper, intended as an educational guide.

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

— Leonhard Euler

Contents

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 and an infinitely differentiable programming space

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

acting on it.

We provide a construction of operators

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

allowing for implementation of the operator

$$\tau_n = 1 + \partial + \partial^2 + \cdots + \partial^n \tag{3}$$

increasing the order of a differentiable programming space

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

All required theorems and proofs are provided by the paper *Operational calculus on programming spaces and generalized tensor networks*. Source code can be found on github [1].

2 Virtual memory

We model an element of the virtual memory $v \in \mathcal{V}_n$

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

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

$$\mathcal{V}_n = \mathcal{V} \otimes T(\mathcal{V}^*) \tag{7}$$

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);
        /*
        *declerations of algebraic operations
        */
};
```

The expanded virtual memory $\mathcal{V}_n$ is the tensor product of the virtual memory $\mathcal{V}$ with the tensor algebra of its dual. The address *var** stands for the component of $v \in \mathcal{V}_{n-1}$ the tensor product with the component of $v^* \in \mathcal{V}^*$ was computed on to generate $v \in \mathcal{V}_n$ in equation (7). This depth is contained in the *int order*.

Tensor products of the virtual memory $\mathcal{V}$ with its dual are modeled using maps, denoted by *dTau*. Naming reflects how the algebra will be constructed, mimicking the operator $\tau_n n$ (3).

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

$$init : \mathcal{V} \times \mathbb{N} \to \mathcal{V}_n \tag{8}$$

$$v_0.init(n) = v_n \in \mathcal{V}_n \tag{9}$$

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

$$v_n = (v_0 \in \mathcal{V}) + (\delta^i_j \in \mathcal{V} \otimes \mathcal{V}^{*\otimes}) + \sum_{i=2}^{n} (0 \in \mathcal{V} \otimes \mathcal{V}^{*i\otimes}) \tag{10}$$

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.

Algebra is constructed by mimicking a direct sum of operators τ_n (3) mapping $\mathcal{P}_0 \oplus \mathcal{P}_0 \to \mathcal{P}_n$. This is reflected in the structure of the class *var* modeling the elements of the virtual space $v \in \mathcal{V}_n$. Compositions are modeled by mimicking the projection of the operator $\exp(\partial_f e^{h\partial_g})$, generalizing both forward and reverse mode automatic differentiation, to the unit hyper-cube and applying it to the resulting direct sum.

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

$$var \in \mathcal{V}_n \tag{11}$$

Proof.

$$id \in \mathcal{V} \wedge dTau \in \mathcal{V}_{n-1} \tag{12}$$

$$\wedge$$

$$var = id \oplus dTau \implies var \in \mathcal{V}_n \tag{13}$$

□

2.2.1 Vector space over a field K

To firstly construct a vector space $\mathcal{V}_n$ over a field K, we implement scalar multiplication and addition.

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, the axioms of the vector space are satisfied. □

2.2.2 Algebra over a field K

With the vector space constructed, we turn towards constructing the 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;
        if(max(v.order,this->order)>0){
                map<int,double> tmp1;
                map<int,double> tmp2;
                for_each_copy(...,mul_make_pair<pair<var*,var> >,
                        v.reduceOrder());
                for_each_copy(...,mul_make_pair<pair<var*,var> >,
                        this->reduceOrder());
                merge_apply(..., sum_pairs<pair<var*,var> >);
        }
        return out;
}
```

The employed functions have been explained at previously usage. The *reduce.Order* function makes a shallow copy of *this*, while reducing the order of the returned copy. This bilinear product contains Leibniz rule within its structure.

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);
        if(this->order>0){
                for_each_copy(...,powTimes<pair<var*,var> >,
                        this->reduceOrder(),n);
        }
        return out;
}
```

employing the function *for_each_copy*, applying the provided operation

```
template<class T, class V, class K>
T powTimes(T v1, V v2, K n) {
  return std::make_pair(v1.first, n*(v2^(n-1))*v1.second);
}
```

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

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.

3.1 Operators

The operator

$$\tau_n = 1 + \partial + \partial^2 + \ldots + \partial^n \tag{14}$$

is used to implement $\mathcal{D}^n C{++} \subset \mathcal{P}_n$, employing the recursive relation

$$\tau_{k+1} = 1 + \partial \tau_k. \tag{15}$$

$$\tau_{k+1} C{++} = C{++} + \partial \tau_k C{++} \tag{16}$$

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

$$\exp(\partial_f e^{h\partial_g})(g) : \mathcal{P} \to \mathcal{P}_\infty(g) \tag{17}$$

projected onto the unit hyper-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, var dTau);
        ~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));
    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

$$C{++} : \mathcal{V} \to \mathcal{V} \tag{18}$$

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

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

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

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

$$\partial^k\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(V^*) \tag{20}$$

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

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

$$\mathcal{D}^n C{++} < \mathcal{P}_n \iff \mathcal{D}^n C{++} \subset C{++} \otimes T(\mathcal{V}^*) \tag{21}$$

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 through the use of the operator *tau*. Assume the existence of functions

$$\sin\_double : double \to double \tag{22}$$

$$\cos\_double : double \to double \tag{23}$$

$$e\_double : double \to double \tag{24}$$

$$\ln\_double : double \to double \tag{25}$$

filling the set spanning C++. Note, that these functions are usually implemented using operations existing in the algebra 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);
    if(v.order>0){
        for_each_copy(...,mul_make_pair<std::pair<var*,var> >,
                cos(v.reduceOrder()));
    }
    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.reduceOrder());
}

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

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

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

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

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

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

$$dCpp < \mathcal{P}_0 \iff \mathcal{D}dCpp \subset dCpp \otimes T(\mathcal{V}^*) \tag{26}$$

Proof.

$$\forall_{\phi_i \in dCpp} \exists_{\phi_j \in dCpp} (\phi_i.primitive = \phi_j) \tag{27}$$

$$\Longrightarrow$$

$$\mathcal{D}dCpp \subset dCpp \otimes T(\mathcal{V}^*) \tag{28}$$

□

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

$$dCpp < \mathcal{P}_0 \iff \mathcal{D}^n dCpp \subset dCpp \otimes T(\mathcal{V}^*) \tag{29}$$

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

3.3 External libraries

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

We illustrate on the example of Eigen [2]. 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 (ex. *init* initializes elements to the desired order).

```
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);
//    number of outputs of the layer
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
```

References

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

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