Variable-free representation of manifolds via transfinite blending with a functional language

Alberto Paoluzzi

Dipartimento di Informatica e Automazione, Università Roma Tre
Via della Vasca Navale 79, 00146 Roma, Italy

paoluzzi@dia.uniroma3.it

Abstract

In this paper a variable-free parametric representation of manifolds is discussed, using transfinite interpolation or approximation, i.e. function blending in some functional space. This is a powerful approach to generation of curves, surfaces and solids (and even higher dimensional manifolds) by blending lower dimensional vector-valued functions. Transfinite blending, e.g. used in Gordon-Coons patches, is well known to mathematicians and CAD people. It is presented here in a very simple conceptual and computational framework, which leads such a powerful modeling to be easily handled even by the non mathematically sophisticated user of graphics techniques. In particular, transfinite blending is discussed in this paper by making use of a very powerful and simple functional language for geometric design.

1 Introduction

Parametric curves and surfaces, as well as splines, are usually defined [2] as vector-valued functions generated from some vector space of polynomials or rationals (i.e. ratio of polynomials) over the field of real numbers. In this paper it is conversely presented an unified view of curves, surfaces, and multi-variate manifolds as vector-valued functions generated from the same vector spaces, but over the field of polynomial (or rational) functions itself. This choice implies that the *coefficients* of the linear combination which uniquely represents a curved mapping in a certain basis are not real numbers, as usually, but vector-valued functions.

This approach is a strong generalization, which contains the previous ones as very special cases. For example, the well-known approach of Hermite cubic interpolation of curves, where two extreme points and tangents are interpolated, can so be applied to surfaces, where two extreme curves of points are interpolated with assigned derivative curves, or even to volume interpolation of two

assigned surfaces with assigned normals. Such an approach is not new, and is quite frequently used in CAD applications, mainly to ship and airplane design, since from the times that Gordon-Coons patches were formulated [3, 5]. It is sometime called *function blending* [5, 8], or *transfinite interpolation* [6, 4]. Transfinite methods have been recently applied to interpolating implicitly defined sets of different cardinality [11].

Transfinite interpolation, that the author preferes to call *transfinite blending*, because it can also be used for approximation, is quite hard to handle by using standard imperative languages. In particular, it is quite difficult to be abstracted, and too often *ad hoc* code must be developed to handle the specific application class or case. A strong mathematical background is also needed to both implement and handle such kind of software. This fact discouraged the diffusion of such a powerful modeling technique outside the close neighbourhood of automobile, ship and airplane shell design.

The contribution of this paper is both in introducing a general algebraic setting which simplifies the description of transfinite blending by the use of functions without variables, and in embedding such an approach into a modern functional computing environment [10], where functions can be easily multiplied and added exactly as numbers. This results in an amazing descriptive power when dealing with parametric geometry. Several examples of this power are given in the paper. Consider, e.g., that multi-variate transfinite Bezier blending of any degree with both domain and range spaces of any dimension is implemented with few lines of quite readable source code.

Last but not least, in the paper we limited our exposition to Bézier and Hermite cases for sake of space. Actually, the same approach can be applied to any kind of parametric representation of geometry, including splines. Notice in particular that different kinds of curves surfaces and splines can be freely blended, so giving a considerable amount of design freedom.

2 Syntax

We use in our discussion the *design language* `PLaSM`, that is a geometric extension of the functional language `FL` [1]. We cannot recap the `PLaSM` syntax here; it is described in [10, 9].[1] In order to understand the examples, it may suffice to remember that the `FL` programming style is strongly based upon functional *composition*. In particular, the composition operator is denoted by "$\sim$", whereas function *application* is denoted by "`:`". Therefore, the `PLaSM` translation of the mathematical expression $(f \circ g)(x)$ is:

```
(f ~ g):x
```

Let us just remember that `< x1, ..., xn >`, where `x1`, ..., `xn` are arbitrary expressions, is a *sequence*, and that language *operators* (i.e., functions) are nor-

[1] The language IDE for *Windows*, *Linux* and *Mac OS X* platforms can be downloaded from `http://www.plasm.net/download/`.

mally *prefix* to their argument sequence. Also, notice that a higher-level function

$$f : A \to B \to C \equiv A \to (B \to C)$$

can be defined, using formal parameters, as

```
DEF f (a::isA)(b::isB) = body_expr
```

where `isA` and `isB` are predicates used to run-time test the set-membership of arguments. The function `f` is applied to actual arguments as `f:x:y` $\equiv$ `(f:x):y`, returning[2] a value `c` $\in C$. Also, notice that the value generated by evaluating the expression `f:x` is a *partial* function $f_{\mathtt{x}} : B \to C$.

3 Variable-free representation

We may define a curve $\boldsymbol{c} : \mathbb{R} \to \mathbb{E}^d$, when a Cartesian system $(\boldsymbol{o}, (\boldsymbol{e}_i))$ is given, as the point-valued mapping $\boldsymbol{c} = \boldsymbol{o} + \boldsymbol{\alpha}$, with $\boldsymbol{\alpha} = (\alpha_i)$, where $\alpha_i : \mathbb{R} \to \mathbb{R}$, for $1 \leq i \leq d$. Therefore, accordingly with the functional character of our language, we may denote a vector-valued function of one real parameter by using the variable-free notation:

$$\boldsymbol{\alpha} = (\alpha_i)^T$$

with $\alpha_i = \boldsymbol{\alpha} \cdot \boldsymbol{e}_i$, with $1 \leq i \leq d$. It should be clearly understood that each α_i is here a map $\mathbb{R} \to \mathbb{R}$ and that each $\boldsymbol{e}_i$ has the constant maps $\underline{0} : \mathbb{R} \to 0$ and $\underline{1} : \mathbb{R} \to 1$ as components.

The variable-free notation [7] discussed in this section, where functions are directly added and multiplied, exactly like numbers, is very useful for easily implementing curves and surfaces in our language, that allows for a very direct translation of such a notation.

Combinators Some combinators are used [7] to perform such variable-free calculus with functions. The model of computation supported by combinatory logic, as the study of combinators is known, is reduction or re-writing, where certain rules are used to re-write an expression over and over again, simplifying or reducing it each time, until to get the answer. The same type of reduction using combinators in used by *Mathematica* [12]. The combinators given below have a direct translation in `FL` and in its geometry-oriented extension `PLaSM`.

1. $\mathrm{id} : \mathbb{R} \to \mathbb{R} : x \mapsto x$ (identity)
2. $\underline{c} : \mathbb{R} \to \mathbb{R} : x \mapsto c$ (constant)
3. $\sigma : \{1, \ldots, d\} \times \mathbb{R}^d \to \mathbb{R} : (i, (x_1, \ldots, x_d)) \mapsto x_i$ (selection)

A computer scientist would probably prefer the following specification, just to point out that σ is often used as a *partial* function, i.e. a function which may be applied to a subset of its arguments:

[2] Since application is left-associative.

3. $\sigma : \{1, \ldots, d\} \to (\mathbb{R}^d \to \mathbb{R}); i \mapsto ((x_1, \ldots, x_d) \mapsto x_i)$ (selection)

Actually, the `FL` primitives `ID`, `K` and `SEL` used by the `PLaSM` language for the functions above, respectively, have no domain restrictions, and can be applied to arbitrary types of data objects.

Algebraic operations We also need to recall how to perform algebraic operations in the algebra of maps $\mathbb{R} \to \mathbb{R}$. In particular, for each map $\boldsymbol{\alpha}, \boldsymbol{\beta} : \mathbb{R} \to \mathbb{R}$ and each scalar $a \in \mathbb{R}$ we have

$$\boldsymbol{\alpha} + \boldsymbol{\beta} : u \mapsto \boldsymbol{\alpha}(u) + \boldsymbol{\beta}(u), \quad \boldsymbol{\alpha}\boldsymbol{\beta} : u \mapsto \boldsymbol{\alpha}(u)\boldsymbol{\beta}(u), \quad a\,\boldsymbol{\beta} : u \mapsto a\,\boldsymbol{\beta}(u).$$

Consequently, we have that

$$\boldsymbol{\alpha} - \boldsymbol{\beta} : u \mapsto \boldsymbol{\alpha}(u) - \boldsymbol{\beta}(u), \quad \boldsymbol{\alpha}/\boldsymbol{\beta} : u \mapsto \boldsymbol{\alpha}(u)/\boldsymbol{\beta}(u).$$

Coordinate representation Finally, remember that the coordinate functions of a curve $\boldsymbol{\alpha} = (\alpha_i)$ are maps $\mathbb{R} \to \mathbb{R}$. The variable-free vector notation stands for the linear combination of coordinate functions with the basis vectors of the target space:

$$(\alpha_1, \ldots, \alpha_d)^T : \ \mathbb{R} \to \mathbb{R}^d; \ u \mapsto \sum_{i=1}^{d} \alpha_i \boldsymbol{e}_i.$$

Example 3.1 (Circular arc)
Some different representations of a circle arc are given here. They have the same image in $\mathbb{E}^2$ but different coordinate representation in the space of functions $\mathbb{R} \to \mathbb{R}$. All such curves generate a circular arc of unit radius centered at the origin.

$$\boldsymbol{\alpha}(u) = \left(\cos\left(\frac{\pi}{2}u\right), \sin\left(\frac{\pi}{2}u\right)\right)^T \qquad u \in [0,1] \tag{1}$$

$$\boldsymbol{\beta}(u) = \left(\frac{1-u^2}{1+u^2}, \frac{2u}{1+u^2}\right)^T \qquad u \in [0,1] \tag{2}$$

$$\boldsymbol{\gamma}(u) = \left(u, \sqrt{1-u^2}\right)^T \qquad u \in [0,1] \tag{3}$$

Below we give a *variable-free representation* of the three maps on the $[0,1]$ interval shown in Example 3.1, that is exactly the representation we need for a `PLaSM` implementation of such maps, provided in Script 3.1:

$$\boldsymbol{\alpha} = \left(\cos \circ \left(\frac{\pi}{\underline{2}}\mathrm{id}\right), \ \sin \circ \left(\frac{\pi}{\underline{2}}\mathrm{id}\right)\right)^T \tag{4}$$

$$\boldsymbol{\beta} = \left(\frac{\underline{1} - \mathrm{id}^2}{\underline{1} + \mathrm{id}^2}, \ \frac{\underline{2}\,\mathrm{id}}{\underline{1} + \mathrm{id}^2}\right)^T \tag{5}$$

$$\boldsymbol{\gamma} = \left(\mathrm{id}, \ \mathrm{id}^{\frac{1}{2}} \circ (\underline{1} - \mathrm{id}^2)\right)^T \tag{6}$$

3.1 Implementation

The circle segment representations of Example 3.1 are directly used in the PLaSM implementation of curves in Script 3.1. To understand the implementation, notice that we generate a polyhedral complex by mapping the vector-valued function (either $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ or $\boldsymbol{\gamma}$) on a cell decomposition of the $[0, 1]$ domain.

According to the semantics of the MAP operator, the curve mapping is applied to all vertices of a simplicial decomposition of a polyhedral domain. But all vertices are represented as *sequences* of coordinates, say <*u*> for a curve, so that in order to act on u the mapping must necessarily *select* it from the sequence. Hence we might substitute each *id* function instance with the PLaSM denotation S1 for the $\sigma(1)$ selector function. Exactly the same result is obtained by using either $\boldsymbol{\alpha} \circ \sigma(1), \boldsymbol{\beta} \circ \sigma(1)$ or $\boldsymbol{\gamma} \circ \sigma(1)$, as done in the following code.

Script 3.1 (Circular arc maps)

```
DEF SQR = ID * ID;
DEF SQRT  = ID ** K:(1/2);

DEF alpha = < cos ~ (K:(PI/2) * ID), sin ~ (K:(PI/2) * ID) >;
DEF beta  = < (K:1 - SQR)/(K:1 + SQR), (K:2 * ID)/(K:1 + SQR) >;
DEF gamma = < ID, SQRT ~ (K:1 - SQR) >;

MAP:(CONS:alpha ~ S1):(interval:<0,1>:10);
MAP:(CONS:beta  ~ S1):(interval:<0,1>:10);
MAP:(CONS:gamma ~ S1):(interval:<0,1>:10);
```

Remarks Let us note that, e.g., alpha is a *sequence* of coordinate functions. Conversely, CONS:alpha is the correct implementation of the *vector-valued* function $\boldsymbol{\alpha}$, which only can be *composed* with other functions, say S1.

Notice also that SQR (square), given in Script 3.1, is the PLaSM implementation of the id^2 function and that the language explicitly requires the operator * to denote the product of functions.

Finally, we remark that SQRT (square root), which is actually primitive in PLaSM, can be also defined easily using standard algebraic rules, where ** is the predefined power operator.

Toolbox Some predicates and functions needed by the operators in this chapter are given in Script 3.2. In particular, the interval operator provides a simplicial decomposition with n elements of the real interval [a, b], whereas the interval2D operator returns a decomposition with n1 × n2 subintervals of the domain $[\mathtt{a1}, \mathtt{b1}] \times [\mathtt{a2}, \mathtt{b2}] \subset \mathbb{R}^2$.

Few other functions of general utility are used in the remainder. In particular, the predicates IsVect and IsPoint are used to test if a data object is a vector or a point, respectively. Some vector operations are also given above, where AA stands for *apply-to-all* a function to a sequence of arguments, producing the sequance of applications, and TRANS and DISTL respectively stand for

Script 3.2 (Toolbox)

```
DEF interval (a,b::IsReal)(n::IsIntPos) =
   (T:1:a ~ QUOTE ~ #:n):((b-a)/n);
DEF interval2D (a1,a2,b1,b2::IsReal)(n1,n2::IsIntPos) =
   interval:<a1,b1>:n1 * interval:<a2,b2>:n2;

DEF vectsum = AA:+ ~ TRANS;
DEF scalarvectprod = AA:* ~ DISTL;
```

transpose a sequence of sequences and for *distribute left* a value over a sequence, returning a sequence of pairs, with the value distributed as the left element of each pair.

Coordinate maps It should be remembered that curves, as $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ in the previous example, are vector-valued functions. In order to obtain their coordinate maps, say $\alpha_i : \mathbb{R} \to \mathbb{R}$, a composition with the appropriate selector function is needed:

$$\alpha_i = \sigma(i) \circ \boldsymbol{\alpha}$$

The conversion from a 3D vector-valued function `curve` := `CONS:alpha` to the sequence of its coordinate functions may be obtained in `PLaSM` as:

```
< S1 ~ curve, S2 ~ curve, S3 ~ curve >;
```

Such an approach is quite expensive and inefficient, because the curve function is repeatedly evaluated to get its three component functions. So, for the sake of efficiency, we suggest maintaining a coordinate representation as a *sequence* of scalar-valued functions, and then `CONS` it into a single *vector-valued* function only when it is strictly necessary.

3.2 Reparametrization

A *smooth curve* is defined as a curve whose coordinate functions are smooth. If $\boldsymbol{c} : I \to \mathbb{E}^d$, with $I \subset \mathbb{R}$, is a smooth curve and $\rho : I \to I$ is a smooth invertible function, i.e. a *diffeomorphism*, then also

$$\boldsymbol{c}_\rho = \boldsymbol{c} \circ \rho$$

is a smooth curve. It is called a *reparametrization* of $\boldsymbol{c}$. A very simple reparametrization is the *change of origin*. For example $\boldsymbol{c}_\rho$ is called a change of origin when

$$\rho = \mathrm{id} + \underline{c}.$$

A reparametrization $\boldsymbol{c}_\tau$ by an affine function

$$\tau = \underline{a}\,\mathrm{id} + \underline{c},$$

with $a \neq 0$, is called an *affine reparametrization*.

Example 3.2 (Circle reparametrization)
The circle with the center in the origin and unit radius may be parametrized on different intervals:

$$\begin{aligned} \boldsymbol{c}_1(u) &= \begin{pmatrix} \cos u & \sin u \end{pmatrix}, & u \in [0, 2\pi] \\ \boldsymbol{c}_2(u) &= \begin{pmatrix} \cos 2\pi u & \sin 2\pi u \end{pmatrix}, & u \in [0, 1] \end{aligned}$$

or with a different starting point:

$$\boldsymbol{c}_3(u) = \begin{pmatrix} \cos(2\pi u + \frac{\pi}{2}) & \sin(2\pi u + \frac{\pi}{2}) \end{pmatrix}, \qquad u \in [0, 1]$$

The reparametrization becomes evident if we use a variable-free representation:

$$\begin{aligned} \boldsymbol{c}_1 &= \begin{pmatrix} \cos & \sin \end{pmatrix}, \\ \boldsymbol{c}_2 &= \begin{pmatrix} \cos & \sin \end{pmatrix} \circ (\underline{2}\,\underline{\pi}\,\mathrm{id}), \\ \boldsymbol{c}_3 &= \begin{pmatrix} \cos & \sin \end{pmatrix} \circ (\underline{2}\,\underline{\pi}\,\mathrm{id} + \underline{\pi}/\underline{2}). \end{aligned}$$

Orientation Two curves with the same image can be distinguished by the sense in which the image is traversed for increasing values of the parameter. Two curves which are traversed in the same way are said to have the same *orientation*. Actually, an orientation is an equivalence class of parametrizations.

A *reversed orientation* of a curve $\boldsymbol{c} : \mathbb{R} \to \mathbb{E}^d$, with image $\boldsymbol{c}[a, b]$, is given by the affine reparametrization $\boldsymbol{c}_\lambda = \boldsymbol{c} \circ \lambda$, where $\lambda : \mathbb{R} \to \mathbb{R}$ such that $x \mapsto -x + (a + b)$. This map can be written as:

$$\lambda = \underline{a + b} - \mathrm{id}.$$

Example 3.3 (Reversing orientation)
It is useful to have a `PLaSM` function `REV`, which reverses the orientation of any curve parametrized on the interval $[a, b] \subset \mathbb{R}$. The `REV` function will therefore be abstracted with respect to the bounds `a` and `b`, which are real numbers.

In Script 3.3 we also give a polyhedral approximation of the boundary of the unit circle centered at the origin, as seen from the angle interval $[0, \frac{\pi}{2}]$. The curve is a map from $[0, 1]$ with reversed orientation.

Script 3.3

```
DEF REV (a,b::IsReal) = K:(a+b) - ID;
DEF alpha = [ COS, SIN ] ~ (K:(PI/2) * ID);

MAP:(alpha ~ REV:<0,1> ~ S1):(interval:<0,1>:20);
```

---

4 Transfinite methods

Transfinite blending stands for interpolation or approximation in *functional spaces*. In this case a bi-variate mapping is generated by blending some univariate maps with a suitable basis of polynomials. Analogously, a three-variate

mapping is generated by blending some bi-variate maps with a polynomial basis, and so on. To implement transfinite blending with `PLaSM` consists mainly in using functions without variables that can be combined, e.g. multiplied and added, exactly as numbers.

Definition A *d-variate parametric mapping* is a point-valued polynomial function $\mathbf{\Phi} : U \subset X \to Y$ with degree k, domain U, support $X = \mathbb{R}^d$ and range $Y = \mathbb{E}^n$.

As commonly used in Computer Graphics and CAD, such point-valued polynomials $\mathbf{\Phi} = (\Phi_j)_{j=1,\ldots,n}$ belong component-wise to the vector space $\mathbb{P}_k[\mathbb{R}]$ of polynomial functions of bounded integer degree k over the $\mathbb{R}$ field.

Coordinates Since the set $\mathbb{P}_k$ of polynomials is also a vector space $\mathbb{P}_k[\mathbb{P}_k]$ over $\mathbb{P}_k$ itself *as a field*, then each mapping component Φ_j, $1 \leq j \leq n$, can be expressed uniquely as a linear combination of $k+1$ basis elements $\phi_i \in \mathbb{P}_k$ with *polynomial coordinates* $\xi_j^i \in \mathbb{P}_k$, so that

$$\Phi_j = \xi_j^0 \phi_0 + \cdots + \xi_j^k \phi_k, \qquad 1 \leq j \leq n.$$

Hence a unique coordinate representation

$$\Phi_j = \left(\xi_j^0, \ldots, \xi_j^k\right)^T, \qquad 1 \leq j \leq n$$

of the mapping is given, after a basis $\{\phi_0, \ldots, \phi_k\} \subset \mathbb{P}_k$ has been chosen.

Choice of a basis If the basis elements are collected into a vector $\boldsymbol{\phi} = (\phi_i)$, then it is possible to write:

$$\mathbf{\Phi} = \mathbf{\Xi}\ \boldsymbol{\phi}, \qquad \boldsymbol{\phi} \in \mathbb{P}_k^{k+1},\ \mathbf{\Phi} \in \mathbb{P}_k^n.$$

where

$$\mathbf{\Xi} = \left(\xi_j^i\right), \qquad 1 \leq i \leq n,\ 0 \leq j \leq k.$$

is the coordinate representation of a linear operator in $Lin[n \times (k+1), \mathbb{P}_k]$ that maps the $k+1$ basis polynomials of k degree, into the n polynomials which transform the vectors in the domain $U \subset \mathbb{R}^d$ into the $\mathbb{E}^n$ points of the manifold. A quite standard choice in computer-aided geometric modeling is $U = [0,1]^d$. The *power* basis, the *cardinal* (or *Lagrange*) basis, the *Hermite* basis, the *Bernstein/Bézier* basis and the *B-spline* basis are the most common and useful choices for the $\boldsymbol{\phi} = (\phi_i)$ basis. This approach can be readily extended to the space $\mathcal{Z}_k$ of rational functions of degree k.

Blending operator The *blending operator* $\mathbf{\Xi}$ specializes the maps generated by a certain basis, to fit and/or to approximate a given set of specific data (points, tangent vectors, boundary curves or surfaces, boundary derivatives, and so on). Its coordinate functions ξ_j^i may be easily generated, as will be shown in the following, by either

1. transforming the *geometric handles* of the mapping into vectors of constant functions, in the standard (non-transfinite) case. These are usually points or vectors $\boldsymbol{x}_j = (x_j^i) \in \mathbb{E}^n$ to be interpolated or approximated by the set $\boldsymbol{\Phi}(U)$;

2. assuming directly as ξ_j^i the components of the curve (surface, etc.) maps to be fit or approximated by $\boldsymbol{\Phi}$, in the transfinite case.

Notation For the sake of readability, only greek letters, either capitals or lower-case, are used here to denote functions. Latin letters are used for numbers and vectors of numbers. As usually in this book, bold letters denote vectors, points or tensors. Please remember that B and H are also Greek capitals for β and η, respectively.

4.1 Uni-variate case

Let consider the uni-variate case $\boldsymbol{\Phi} : U \subset X \to Y$, where the dimension d of support space X is one. To generate the coordinate functions ξ_j^i it is sufficient to transform each data point $\boldsymbol{x}_i = (x_j^i) \in Y$ into a vector of constant functions, so that

$$\xi_j^i = \kappa(x_j^i), \quad \text{where} \quad \kappa(x_j^i) : U \to Y : u \mapsto x_j^i.$$

We remember that, using the functional notation with explicit variables, the *constant* function $\kappa : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$ is defined such that

$$\kappa(x_j^i)(u) = x_j^i$$

for each parameter value $u \in U$. The `PLaSM` implementation clearly uses the constant functional `K` at this purpose.

Example 4.1 (Cubic Bézier curve in the plane)
The cubic Bézier plane curve depends on four points `p0,p1,p2,p3` $\in E^2$, which are given in Script 4.1 as formal parameters of the function `Bezier3`, that generates the Φ mapping by linearly combining the basis functions with the coordinate functions. The local functions `b0,b1,b2,b3` implement the cubic Bernstein/Bézier basis functions $\beta_i^3 : \mathbb{R} \to \mathbb{R}$ such that $u \mapsto \binom{3}{i} u^i (1-u)^{3-i}$, $0 \leq i \leq 3$.

The `x` and `y` functions, defined as composition of a selector with the constant functional `K`, respectively select the first (second) component of their argument sequence and transform such a number in a constant function. The reader should notice that `+` and `*` are used as operators *between functions*.

4.2 Multi-variate case

When the dimension d of the support space X is greater than one, two main approaches can be used to construct a parametric mapping Φ. The first approach is the well-known *tensor-product* method. The second approach, discussed below, is called here *transfinite blending*.

Script 4.1

```
DEF Bezier3 (p0,p1,p2,p3::IsSeqOf:isReal) =
   [ (x:p0 * b0) + (x:p1 * b1) + (x:p2 * b2) + (x:p3 * b3),
   (y:p0 * b0) + (y:p1 * b1) + (y:p2 * b2) + (y:p3 * b3) ]
WHERE
   b0 = u1 * u1 * u1,
   b1 = K:3 * u1 * u1 * u,
   b2 = K:3 * u1 * u * u,
   b3 = u * u * u,
   x = K ~ S1, y = K ~ S2, u1 = K:1 - u, u = S1
END;
```

Transfinite blending Let consider a polynomial mapping $\mathbf{\Phi} : U \to Y$ of degree k_d, where U is d-dimensional and Y is n-dimensional. Since $\mathbf{\Phi}$ depends on d parameters, in the following will be denoted as ${}^d\mathbf{\Phi} = ({}^d\Phi_j)$, $1 \le j \le n$. In this case ${}^d\mathbf{\Phi}$ is computed component-wise by linear combination of coordinate maps, depending on $d-1$ parameters, with the *uni-variate* polynomial basis $\boldsymbol{\phi} = (\phi_i)$ of degree k_d. Formally we can write:

$$ {}^d\Phi_j = {}^{d-1}\Phi_j^0\, \phi_0 + \dots {}^{d-1}\Phi_j^{k_d}\, \phi_{k_d}, \qquad 1 \le j \le n. $$

The coordinate representation of ${}^d\Phi_j$ with respect to the basis $(\phi_0, \dots, \phi_{k_d})$ is so given by $k_d + 1$ maps depending on $d-1$ parameters:

$$ {}^d\Phi_j = \left({}^{d-1}\Phi_j^0, \dots, {}^{d-1}\Phi_j^{k_d} \right). $$

In matrix notation, after a polynomial basis $\boldsymbol{\phi}$ has been chosen, it is

$$ {}^d\Phi = \mathbf{\Xi}\ \boldsymbol{\phi}, \quad \text{where} \quad \mathbf{\Xi} = ({}^{d-1}\Phi_j^i), \quad \boldsymbol{\phi} = (\phi_i), \quad 0 \le i \le k_d,\ 1 \le j \le n. $$

Example As an example of transfinite blending consider the generation of a bicubic Bézier surface mapping $B(u_1, u_2)$ as a combination of four Bézier cubic curve maps $B_i(u_1)$, with $0 \le i \le 3$, where some curve maps may possibly reduce to a constant point map:

$$ B(u_1, u_2) = \sum_{i=0}^{3} B_i(u_1)\ \beta_i^3(u_2) $$

where

$$ \beta_i^3(u) = \binom{3}{i} u^i\, (1-u)^{3-i}, \qquad 0 \le i \le 3, $$

is the Bernstein/Bézier cubic basis. Analogously, a three-variate Bézier solid body mapping $B(u_1, u_2, u_3)$, of degree k_3 on the last parameter, may be generated by uni-variate Bézier blending of surface maps $B_i(u_1, u_2)$, some of which possibly reduced to a curve map or even to a constant point map:

$$ B(u_1, u_2, u_3) = \sum_{i=0}^{k_3} B_i(u_1, u_2)\ \beta_i^{k_3}(u_3) $$

Note The more interesting aspects of transfinite blending are *flexibility* and *simplicity*. Conversely than in tensor-product method, there is no need that all component geometries have the same degree, and even neither that were all generated using the same function basis. For example, a quintic Bézier surface map may be generated by blending both Bézier curve maps of lower (even zero) degree together with Hermite and Lagrange curve maps. Furthermore, it is much simpler to combine lower dimensional geometries (i.e. maps) than to meaningfully assembly the multi-index tensor of control data (i.e. points and vectors) to generate multi-variate manifolds with tensor-product method.

4.3 Transfinite Bézier

The full power of the `PLaSM` functional approach to geometric programming is used in this section, where dimension-independent transfinite Bézier blending of any degree is implemented in few lines of code, by easily combining coordinate maps which may depend on an arbitrary number of parameters.

We remark that the `Bezier` : $[0,1]^d \to \mathbb{E}^n$ mapping given here can be used:

1. to blend points to give curve maps;
2. to blend curve maps to give surface maps;
3. to blend surface maps to give solid maps;
4. and so on ...

Notice also that the given implementation is independent on the dimensions d and n of support and range spaces.

Implementation At this purpose, first a small toolbox of related functions is needed, to compute the factorial function, the binomial coefficients, the $\boldsymbol{\beta}^k = (\beta_i^k)$ Bernstein basis of degree k, and the β_i^k Bernstein/Bézier polynomials.

Script 4.2 (Transfinite Bézier toolbox)

```
DEF Pred = C:-:1;
DEF Fact (n::IsNat) = IF:< C:EQ:0, K:1, * ~ INTSTO >;
DEF Choose (n,k::IsNat) = IF:<
   OR ~ [C:EQ:0 ~ S2, EQ], K:1, Choose ~ AA:Pred * / >:< n,k >;
DEF Bernstein (u::IsFun)(n::IsInt)(i::IsInt) =
   ~ [K:(Choose:<n,i>),** ~ [ID,K:i], ** ~ [- ~ [K:1,ID],K:(n-i)]] ~ u;
DEF BernsteinBasis (u::IsFun)(n::IsInt) = AA:(Bernstein:u:n):(0..n);
```

Then the `Bezier:u` function is given, to be applied on the sequence of `Data`, which may contain either control points $\boldsymbol{x}_i = (x_j^i)$ or control maps ${}^{d-1}\Phi_i = ({}^{d-1}\Phi_j^i)$, with $0 \leq i \leq k$, $1 \leq j \leq n$. In the former case each component x_j^i of each control point is firstly transformed into a constant function.

The body of the `Bezier:u` function just linearly combines component-wise the sequence (ξ_j^i) of coordinate maps generated by the expression

```
(TRANS~fun):Data
```

with the basis sequence (β_i^k) generated by `BernsteinBasis:u:degree`, where `degree` equates the number of geometric handles minus one.

Script 4.3 (Dimension-independent transfinite Bézier mapping)

```
DEF Bezier (u::IsFun) (Data::IsSeq) = (AA:InnerProd ~ DISTR):
   < (fun ~ TRANS):Data, BernsteinBasis:u:degree >
WHERE
   degree = LEN:Data - 1,
   fun = (AA ~ AA):(IF:< IsFun, ID, K >)
END;
```

It is much harder to explain in few words what actual argument to pass (and why) for the formal parameter `u` of the `Bezier` function. As a rule of thumb let pass either the selector `S1` if the function must return a uni-variate (curve) map, or `S2` to return a bi-variate (surface) map, or `S3` to return a three-variate (solid) map, and so on.

Example 4.2 (Bézier curves and surface)
Four Bézier $[0,1] \to \mathbb{E}^3$ maps `C1`, `C2`, `C3`, and `C4`, respectively of degrees $1, 2, 3$ and 2 are defined in Script 4.4.

It may be useful to notice that the control points have three coordinates, so that the generated maps `C1`, `C2`, `C3` and `C4` will have three component functions. Such maps can be blended with the Bernstein/Bézier basis to produce a cubic transfinite bi-variate (i.e. surface) mapping:

$$B(u_1, u_2) = \texttt{C0}(u_1)\beta_0^3(u_2) + \texttt{C1}(u_1)\beta_1^3(u_2) + \texttt{C2}(u_1)\beta_2^3(u_2) + \texttt{C3}(u_1)\beta_3^3(u_2).$$

Such a linear combination of coordinate functions with the Bézier basis (here working on the second coordinate of points in $[0,1]^2$) is performed by the `PLaSM` function `Surf1`, defined by using again the `Bezier` function.

A simplicial approximation (with triangles) of the surface $B[0,1]^2 \subset \mathbb{E}^3$ is finally generated by evaluating the last expression of Script 4.4.

Script 4.4

```
DEF C0 = Bezier:S1:<<0,0,0>,<10,0,0>>;
DEF C1 = Bezier:S1:<<0,2,0>,<8,3,0>,<9,2,0>>;
DEF C2 = Bezier:S1:<<0,4,1>,<7,5,-1>,<8,5,1>,<12,4,0>>;
DEF C3 = Bezier:S1:<<0,6,0>,<9,6,3>,<10,6,-1>>;

DEF Surf1 = Bezier:S2:<C0,C1,C2,C3>;

MAP:Surf1:(Intervals:1:20 * Intervals:1:20);
```

According to the semantics of the `MAP` operator, `Surf1` is applied to all vertices of the automatically generated simplicial decomposition Σ of the 2D

product ($\texttt{Intervals} : \texttt{1} : \texttt{20} * \texttt{Intervals} : \texttt{1} : \texttt{20}) \subset \mathbb{R}^2$. A simplicial approximation $B(\Sigma)$ of the surface $B([0,1]^2) \subset \mathbb{E}^3$ is finally produced and displayed in Figure 1c.

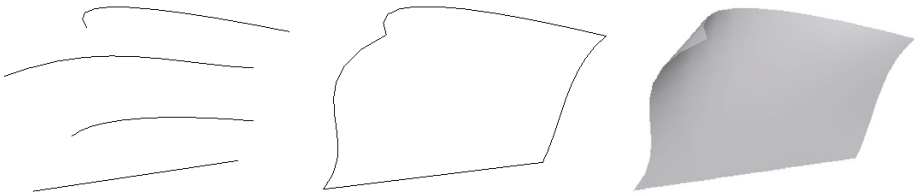

Figure 1: (a) Graphs of four Bézier curve maps `c0`, `c1`, `c2` and `c3`; (b) graphs of `c0` and `c3` together with graphs of bicubic maps `b0` and `b1` generated by extreme control points; (c) graph of `Surf1` surface

The four generating curves and the generated cubic blended surface are displayed in Figures 1a. It is possible to see (Figure 1c) that such surface interpolates the four boundary curves defined by the extreme control points, exactly as in the case of tensor-product method, but obviously with much greater generality, since any defining curve may be of any degree.

4.4 Transfinite Hermite

The cubic Hermite uni-variate map is the unique cubic polynomial $\boldsymbol{H} : [0,1] \to \mathbb{E}^n$ which matches two given points $p_0, p_1 \in \mathbb{E}^n$ and derivative vectors $t_0, t_1 \in \mathbb{R}^n$ for $u = 0, 1$ respectively. Let denote as $\eta^3 = \left(\eta_0^3, \eta_1^3, \eta_2^3, \eta_3^3\right)$ the cubic Hermite function basis, with

$$\eta_i^3 : [0,1] \to \mathbb{R}, \qquad 0 \leq i \leq 3,$$

and such that

$$\eta_0^3(u) = 2u^3 - 3u^2 + 1, \quad \eta_1^3(u) = 3u^2 - 2u^3, \quad \eta_2^3(u) = u^3 - 2u^2 + u, \quad \eta_3^3(u) = u^3 - u^2.$$

Then the mapping $\boldsymbol{H}$ can be written, in vector notation, as

$$\begin{aligned} \boldsymbol{H} &= \boldsymbol{\xi}_0\ \eta_0^3 + \boldsymbol{\xi}_1\ \eta_1^3 + \boldsymbol{\xi}_2\ \eta_2^3 + \boldsymbol{\xi}_3\ \eta_3^3 \\ &= \kappa(\boldsymbol{p}_0)\ \eta_0^3 + \kappa(\boldsymbol{p}_1)\ \eta_1^3 + \kappa(\boldsymbol{t}_0)\ \eta_2^3 + \kappa(\boldsymbol{t}_1)\ \eta_3^3. \end{aligned}$$

It is easy to verify, for the uni-variate case, that:

$$\begin{aligned} \boldsymbol{H}(0) &= \kappa(\boldsymbol{p}_0)(0) = \boldsymbol{p}_0, \quad \boldsymbol{H}(1) = \kappa(\boldsymbol{p}_1)(1) = \boldsymbol{p}_1, \\ \boldsymbol{H}'(0) &= \kappa(\boldsymbol{t}_0)(0) = \boldsymbol{t}_0, \quad \boldsymbol{H}'(1) = \kappa(\boldsymbol{t}_1)(1) = \boldsymbol{t}_1, \end{aligned}$$

and that the image set $\boldsymbol{H}[0,1]$ is the desired curve in $\mathbb{E}^n$.

A multi-variate transfinite Hermite map $\boldsymbol{H}^n$ can be easily defined by allowing the blending operator $\boldsymbol{\Xi} = \left(\boldsymbol{\xi}_j\right) = \left(\xi_j^i\right)$ to contain maps depending at most on $d-1$ parameters.

Implementation A transfinite `CubicHermite` mapping is implemented here, with four data objects given as formal parameters. Such data objects may be either points/vectors, i.e. sequences of numbers, or $1/2/3/d$-variate maps, i.e. sequences of (curve/surface/solid/etc) component maps, or even mixed sequences, as will be shown in the following examples.

Script 4.5 (Dimension-independent transfinite cubic Hermite)

```
DEF fun = (AA ~ AA):(IF:<IsFun,ID,K>);

DEF HermiteBasis (u::IsFun) = < h0,h1,h2,h3 >
WHERE
   h0 = k:2 * u3 - k:3 * u2 + k:1,
   h1 = k:3 * u2 - k:2 * u3,
   h2 = u3 - k:2 * u2 + u,
   h3 = u3 - u2, u3 = u*u*u, u2 = u*u
END;

DEF CubicHermite (u::IsFun) (p1,p2,t1,t2::IsSeq) =
   (AA:InnerProd ~ DISTL):
      < HermiteBasis:u, (TRANS ~ fun):<p1,p2,t1,t2> >;
```

4.5 Connection surfaces

The creation of surfaces smoothly connecting two given curves with assigned derivative fields by cubic transfinite blending is discussed in this section. The first applications concern the generation of surfaces in 3D space, the last ones concern the generation of planar grids as 1-skeletons of 2D surfaces, according to the dimension-independent character of the given `PLaSM` implementation of transfinite methods.

The curve maps $\texttt{c1}(u)$ and $\texttt{c2}(u)$ of Example 4.5 are here interpolated in 3D by a `Surf2` mapping using the cubic Hermite basis $\eta^3 = (\eta^3_j)$, $0 \leq j \leq 3$, with the further constraints that the tangent vector field $\texttt{Surf2}^v(u, 0)$ along the first curve are constant and parallel to $(0, 0, 1)$, whereas $\texttt{Surf2}^v(u, v)$ along the second curve is also constant and parallel to $(0, 0, -1)$. The resulting map

$$\texttt{Surf2} : [0, 1]^2 \to \mathbb{E}^3$$

has unique vector representation in $\mathbb{P}^3_3[\mathbb{P}_3]$ as

$$\texttt{Surf2} = \texttt{c1}\ \eta^3_0 + \texttt{c2}\ \eta^3_1 + (\kappa(0), \kappa(0), \kappa(1))\ \eta^3_2 + (\kappa(0), \kappa(0), \kappa(-1))\ \eta^3_3.$$

Example 4.3 (Surface interpolation of curves)
Such a map is very easily implemented by the following `PLaSM` definition. A simplicial approximation $\texttt{Surf2}(\Sigma)$ of the point-set $\texttt{Surf2}([0, 1]^2)$ is generated by the `MAP` expression in Script 4.6, and is shown in Figure 2.

Script 4.6 (Surface by Hermite's interpolation (1))

```
DEF Surf2 = CubicHermite:S2:< c1,c2,<0,0,1>,<0,0,-1> >;
MAP: Surf2:  (Domain:14 * Domain:14);
```

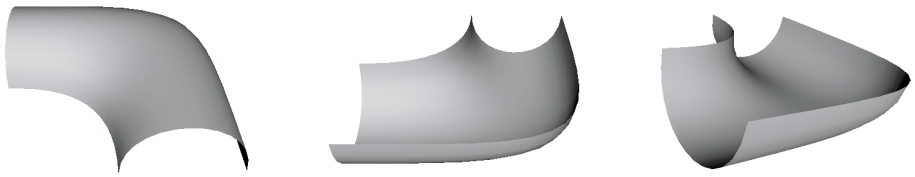

Figure 2: Some pictures of the surface interpolating two plane Hermite curves with constant vertical tangent vectors along the curves.

Example 4.4 (Surface interpolation of curves)
A different surface interpolation of the two plane curves `c1` and `c2` is given in Script 4.7, where the boundary tangent vectors are constrained to be constant and parallel to $(1, 1, 1)$ and $(-1, -1, -1)$, respectively. Some pictures of the resulting surface are given in Figure 3.

Example 4.5 (Grid generation)
Two planar Hermite curve maps `c1` and `c2` are defined, so that the curve images `c1`$[0, 1]$ and `c2`$[0, 1]$.

Some different grids are easily generated from the plane surface which interpolates the curves `c1` and `c2`. At this purpose it is sufficient to apply the `CubicHermite:S2` function to different tangent curves.

The grids generated by maps `grid1`, `grid2` and `grid3` are shown in Figure 4. The tangent map `d` is simply obtained as component-wise difference of the curve maps `c2` and `c1`.

It is interesting to notice that the map `grid1` can be also generated as linear (transfinite) Bézier interpolation of the two curves, as given below. Clearly the solution as cubic Hermite is more flexible, as it is shown by Figures 4b and 4c.

Figure 3: Some pictures of a new surface interpolating the same Hermite curves with constant oblique tangent vectors.

Script 4.7 (Surface by Hermite's interpolation (2))

```
DEF Surf3 = CubicHermite:S2:<c1,c2,<1,1,1>,<-1,-1,-1>>;
MAP: Surf3:  (Domain:14 * Domain:14);
```

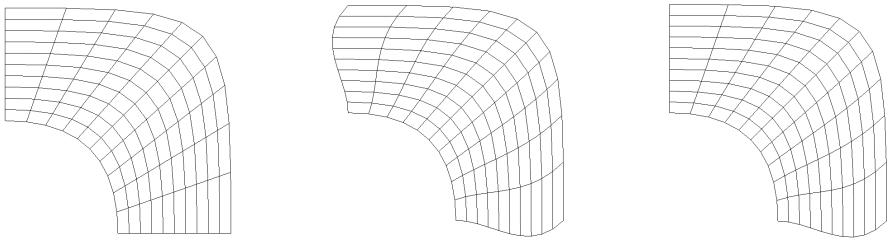

Figure 4: The simplicial complexes generated by the `MAP` operator on the `grid1`, `grid2` and `grid3` maps given in Example 4.5.

```
DEF grid1 = Bezier:S2:<c1,c2>;
MAP:(CONS:grid1):(Domain:8 * Domain:8);
```

References

[1] Backus, J., Williams, J., and Wimmers, E. An introduction to the programming language FL. In *Research Topics In Functional Programming*, D. Turner, Ed. Addison-Wesley, Reading, Massachusetts, 1990, ch. 9, pp. 219–247.

[2] Bartels, R., Beatty, J., and Barsky, B. *An Introduction to Splines for Use in Computer Graphics & Geometric Modeling*. Morgan Kaufmann, Los Altos, CA, 1987.

[3] Coons, S. Surfaces for computer-aided design of space forms. Tech. Rep. MAC-TR-41, MIT, Cambridge, MA, 1967.

[4] Goldman, R. The role of surfaces in solid modeling. In *Geometric Modeling: Algorithms and New Trends*, G. Farin, Ed. SIAM Publications, Philadelphia, Pennsylvania, 1987.

Script 4.8 (Examples of planar grids)

```
DEF c1 = CubicHermite:S1:<<1,0>,<0,1>,<0,3>,<-3,0>>;
DEF c2 = CubicHermite:S1:<<0.5,0>,<0,0.5>,<0,1>,<-1,0>>;
DEF d = (AA:- ~ TRANS):<c2,c1>;

DEF grid1 = CubicHermite:S2:<c1,c2,d,d>;
DEF grid2 = CubicHermite:S2:<c1,c2,<-0.5,-0.5>,d>;
DEF grid3 = CubicHermite:S2:<c1,c2,<S1:d,-0.5>,d>;
```

[5] Gordon, W. Blending function methods of bivariate and multivariate interpolation and approximation. Tech. Rep. GMR-834, General Motors, Warren, Michigan, 1968.

[6] Gordon, W. Spline-blended surface interpolation through curve networks. *Journal of Mathematical Mechanics 18* (1969), 931–952.

[7] Jones, A., Gray, A., and Hutton, R. *Manifolds and Mechanics*. No. 2 in Australian Mathematical Society Lecture Series. Cambridge University Press, Cambridge, UK, 1987.

[8] Lancaster, P., and Salkauskas, K. *Curve and Surface Fitting. An Introduction*. Academic Press, London, UK, 1986.

[9] Paoluzzi, A. *Geometric programming for computer aided design*. J. Wiley & Sons, Chichester, UK, 2003.

[10] Paoluzzi, A., Pascucci, V., and Vicentino, M. Geometric programming: a programming approach to geometric design. *ACM Transactions on Graphics 14*, 3 (July 1995), 266–306.

[11] Rvachev, V. L., Sheiko, T. I., Shapiro, V., and Tsukanov, I. Transfinite interpolation over implicitly defined sets. *Computer Aided Geometric Design 18* (2001), 195–220.

[12] Wolfram, S. *A new kind of science*. Wolfram Media, Inc., 2002.