B-splines pour l'optimisation de forme

Mehmet Celikbas

▶ **To cite this version:**

Mehmet Celikbas. B-splines pour l'optimisation de forme. RR-4968, INRIA. 2003. inria-00071611

HAL Id: inria-00071611
https://inria.hal.science/inria-00071611

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

B-splines pour l'optimisation de forme

Mehmet Celikbas

N° 4968

Octobre 2003

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4968--FR

INRIA
SOPHIA ANTIPOLIS

B-splines pour l'optimisation de forme

Mehmet Celikbas

Thème 2 — Génie logiciel
et calcul symbolique
Projet GALAAD

Rapport de recherche n° 4968 — Octobre 2003 — 33 pages

Résumé : Ce rapport décrit mon travail de stage dont l'objectif était l'implémentation d'une bibliothèque sur les courbes et surfaces B-splines et l'utilisation de cette bibliothèque pour manipuler les courbes qui interviennent dans des problèmes d'optimisation géométrique.

Mots-clés : Courbes et surfaces B-spline, C++, SYNAPS, approximation B-spline, paramétrisation, optimisation géométrique

Ce travail a été réalisé dans le cadre de la COLORS Forme

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

B-spline for shape optimisation

Abstract: This report describes the work done during my internship in the project GALAAD. The subject was the implementation of a module for B-spline curves and surfaces and its use in the treatment of curves, involved in shape optimisation problems.

Key-words: B-spline curve and surface, C++, SYNAPS, approximation, parameterisation, shape optimisation

Table des matières

1 Courbes et surfaces B-splines **4**
1.1 Introduction 4
1.2 Fonctions de bases des B-splines 4
1.2.1 Définition 4
1.2.2 Propriétés 5
1.2.3 Choix du vecteur nodal 5
1.2.4 Exemples 5
1.3 Courbes B-splines 6
1.3.1 Définition 6
1.3.2 Propriétés 7
1.3.3 Exemples 8
1.3.4 Algorithmes de bases sur les courbes B-splines 8
1.4 Surfaces B-splines 16
1.4.1 Définition 16
1.4.2 Propriétés 16
1.4.3 Exemple 17
1.4.4 Algorithmes de bases sur les surfaces B-splines 17

2 Implémentation **20**
2.1 Intérêt du C++ 20
2.2 SYNAPS 21
2.2.1 Présentation 21
2.2.2 Architecture de SYNAPS 21
2.3 Mise en oeuvre 22
2.3.1 Le conteneur *bspline::curve* 22
2.3.2 Le conteneur *bspline::surface* 22
2.3.3 Exemples d'utilisation 23

3 Application à l'optimisation géométrique **26**
3.1 Présentation 26
3.2 Optimisation couplée à une adaptation de la paramétrisation avec des courbes de Bézier (Rappels) 26
3.3 Adaptation de la paramétrisation avec des courbes B-splines. 27
3.4 Optimisation couplée à une adaptation de la paramétrisation avec des courbes B-spline 29
3.4.1 Optimisation du critère J 29
3.4.2 Résultats 30

1 Courbes et surfaces B-splines

1.1 Introduction

En C.A.O (Conception Assistée par Ordinateur) les objets utilisés sont souvent des polynômes : courbes polynomiales et surfaces polynomiales; mathématiquement, ces polynômes doivent s'écrire dans une base donnée. Le choix de cette base est important vis-à-vis des propriétés de ces courbes ou surfaces. En effet, prenons l'exemple d'une courbe de Bézier (i.e. écrite dans la base de Bernstein) :

$$P(t) = \sum_{i=0}^{n} B_i^n(t) P_i$$

où

$$B_i^n(t) = \mathbb{C}_n^i t^i (1-t)^{n-i}, \quad P_i \in \mathbb{R}^2, \quad t \in [0,1]$$

Ainsi $P(t)$ est constituée d'une seule courbe polynomiale si bien que $P(t)$ a une continuité de classe $\mathcal{C}^\infty$. Dans ces conditions, il paraît difficile de modéliser des courbes ayant des points anguleux ie. des courbes dont un ou plusieurs points n'ont qu'une continuité $\mathcal{C}^0$. Nous allons voir dans la suite que l'introduction de la base des B-splines (en anglais B*asis*-splines) permet de remédier à ce problème.

1.2 Fonctions de bases des B-splines

1.2.1 Définition

On définit les fonctions de bases de la façon suivante :

- par un ensemble U de $m+1$ réels u_i non décroissants : $u_0 \leq u_1 \leq ... \leq u_m$.
  U est appelé vecteur nodal, les u_i sont appelés noeuds.
- par un degré d.

La i-ème fonction de bases B-splines de degré d (ou d'ordre $d+1$) est alors définie par récurrence :

$$\begin{aligned} N_i^0(u) &= \chi_{[u_i, u_{i+1}[} = \begin{cases} 1 & si\ x \in [u_i, u_{i+1}[\\ 0 & sinon \end{cases} \\ N_i^d(u) &= \frac{u - u_i}{u_{i+d} - u_i} N_i^{d-1}(u) + \frac{u_{i+d+1} - u}{u_{i+d+1} - u_{i+1}} N_{i+1}^{d-1}(u) \end{aligned} \tag{1}$$

1.2.2 Propriétés

- $N_i^d(u)$ est un polynôme de degré d à support compact, de support $[u_i,u_{i+d+1}]$.
- $N_i^d(u)$ a pour dérivée :

$$\frac{d}{du}N_i^d(u) = d\left(\frac{N_i^{d-1}(u)}{u_{i+d} - u_i} - \frac{N_{i+1}^{d-1}(u)}{u_{i+d+1} - u_{i+1}}\right) \tag{2}$$

- Si l'on regroupe les noeuds u_i par paquets de valeurs identiques $\tau_1 < ... < \tau_k$, chaque τ_j étant répétée m_j fois dans le vecteur nodal U avec $\sum_{j=0}^{k} m_j = m + 1$. Alors $N_i^d(u)$ en chaque point τ_j est continue de classe $\mathcal{C}^{d-m_j}$. Ceci a pour conséquence qu'augmenter la multiplicité m_j d'un noeud, diminue la continuité en ce point.
- Si l'on veut définir $n + 1$ fonctions de bases de degré d à partir d'un vecteur nodal de longueur $m + 1$ alors n doit vérifier la contrainte fondamentale suivante :

$$m = n + d + 1 \tag{3}$$

  En effet, la dernière B-spline $N_n^d(u)$ a pour support $[u_n,u_{n+d+1}]$, comme c'est la dernière B-spline, u_{n+d+1} doit être égale au dernier noeud u_m.
  Ainsi, si l'on dispose d'un vecteur nodal de longueur $m + 1$ et d'un degré d alors il y a exactement $n + 1$ fonctions de bases avec $n = m - d - 1$.

1.2.3 Choix du vecteur nodal

Le choix du vecteur nodal est conditionné par 3 et par le fait qu'il doit être croissant. Dans la pratique[1], on prend plutôt :

$$u_0 = ... = u_d < u_{d+1} < ... < u_{n-1} < u_n = ... = u_{n+d+1} \tag{4}$$

i.e on répète $d + 1$ fois les valeurs extrêmes.

1.2.4 Exemples

Prenons le vecteur nodal $u_0 = u_1 = u_2 = u_3 = 0, u_4 = 1, u_5 = 2, u_6 = u_7 = u_8 = u_9 = 3$ et construisons les fonctions de bases de degré 3, on a donc $n = m - d - 1 = 9 - 3 - 1 = 5$, i.e 6 fonctions de bases. Pour obtenir ces fonctions de bases on applique la relation 1, ce qui nous donne les fonctions suivantes:

1. Nous verrons plus loin pourquoi.

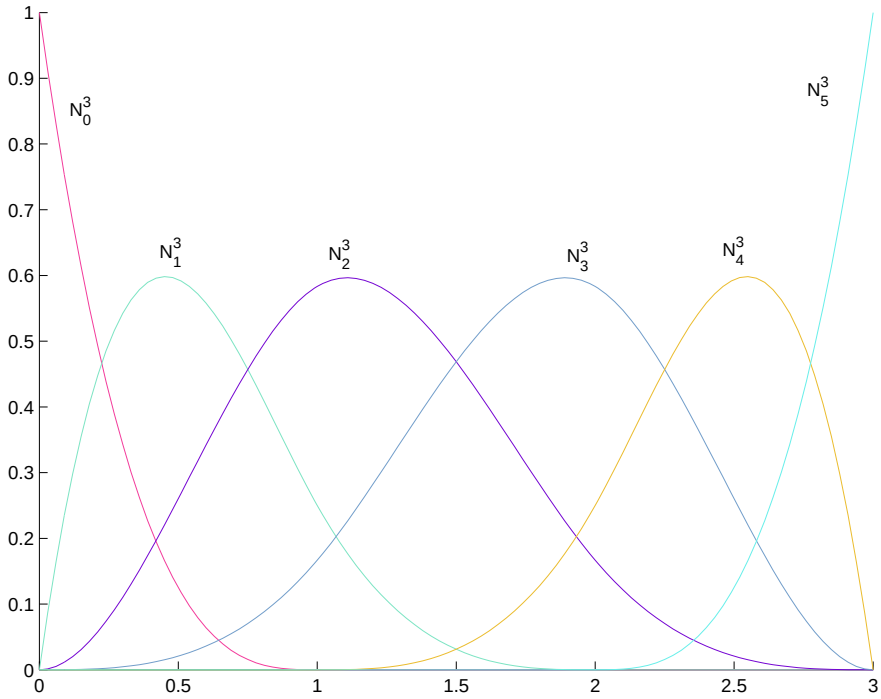

FIG. 1 – *Fonctions de bases pour le vecteur nodal* [0,0,0,0,1,2,3,3,3,3]

1.3 Courbes B-splines

1.3.1 Définition

Une courbe B-spline de degré d (ou d'ordre $d+1$) est la courbe paramétrée définie par :

$$P(u) = \sum_{i=0}^{n} N_i^d(u) P_i \tag{5}$$

où $P_i \in \mathbb{R}^d$, ($d = 2$ ou $d = 3$).

Pour construire une courbe B-spline de degré d à partir de $n+1$ points P_i dit points de contrôle, il faut donc se donner $m+1$ noeuds où $m = n + d + 1$, permettant de définir les fonctions de bases $N_i^d(u)$.

1.3.2 Propriétés

- Chaque $P([u_k,u_{k+1}])$ définie un arc de courbe délimité par les points $P(u_k)$ et $P(u_{k+1})$. Sur chaque arc de courbe $P(u)$ est polynomiale de degré d.
- En un point u distinct d'un noeud, $P(u)$ est de classe $\mathcal{C}^\infty$(polynôme) et en un noeud u_k de multiplicité m_k, $P(u)$ est de classe $\mathcal{C}^{d-m_k}$.
- Dérivée d'une courbe B-spline :

$$P'(u) = d \sum_{i=0}^{n-1} \frac{N_{i+1}^{d-1}(u)}{u_{i+d+1} - u_{i+1}} (P_{i+1} - P_i) \tag{6}$$

  La dérivée d'une courbe B-spline est donc aussi une courbe B-spline.
- Pour un u donné, $P(u)$ appartient à l'enveloppe convexe des points P_0,...,P_n.
- Compte tenu des supports des fonctions de bases, si $u \in [u_k,u_{k+1}]$ alors les seules fonctions de bases non nulles sont N_{k-d}^d, ..., N_k^d . Donc :

$$P(u) = \sum_{i=k-d}^{k} N_i^d(u) P_i, \text{ pour } u \in [u_k,u_{k+1}]$$

  et $P(u)$ appartient à l'enveloppe convexe des points P_{k-d},...,P_k.
- Pour assurer la coïncidence de la courbe avec les points de contrôle extrêmes, on prend le vecteur nodal suivant :

$$u_0 = ... = u_d < u_{d+1} < ... < u_{n-1} < u_n = ... = u_{n+d+1}$$

  En effet, on a alors $N_i^d(u_0) = \begin{cases} 1 & si\ i = 0 \\ 0 & sinon \end{cases}$ donc $P(u_0) = 0$ et $P'(u_0) = \frac{d}{u_{d+1}-u_1}(P_1 - P_0)$; de même pour $P(u_{n+d+1}) = P_n$ et $P'(u_{n+d+1}) = \frac{d}{u_{n+d}-u_n}(P_n - P_{n-1})$.

1.3.3 Exemples

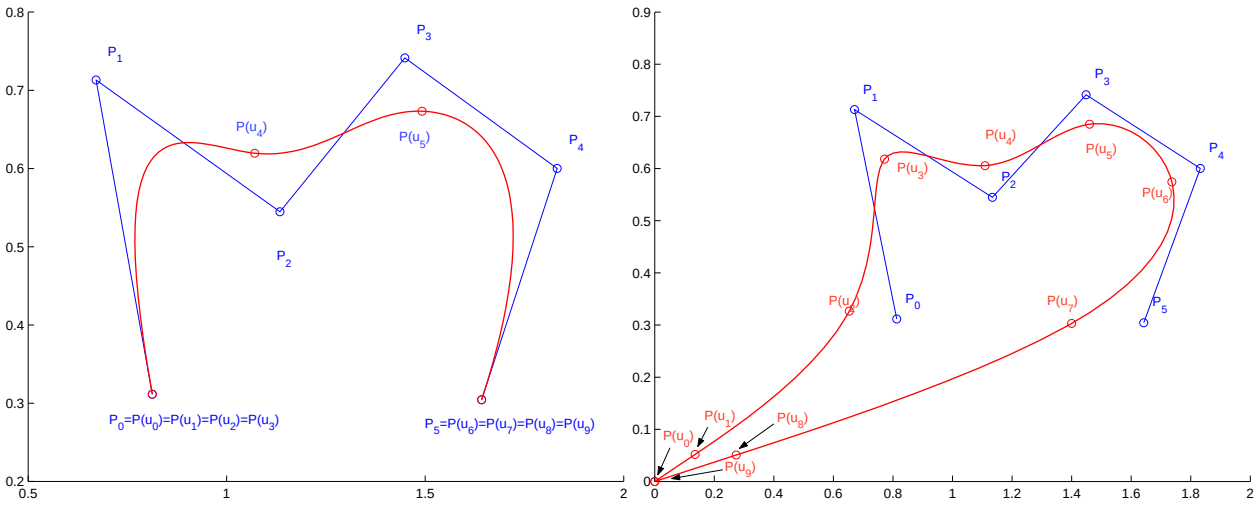

FIG. 2 – *Courbe B-spline de degré 3, avec le vecteur nodal* [0,0,0,0,1,2,3,3,3,3] *(à gauche); avec le vecteur nodal* [0,1,2,3,4,5,6,7,8,9] *(à droite)*

La figure de droite montre un point important : si l'on a pas des noeuds extrêmes de multiplicités maximales alors la B-spline s'annule forcément à l'origine en les noeuds extrêmes.

1.3.4 Algorithmes de bases sur les courbes B-splines

Calculs des fonctions de bases. Pour calculer les $\left(N_i^d(u)\right)_{i=0..n}$, on va utiliser la relation de récurrence (1); en appliquant cette relation on obtient le schéma suivant :

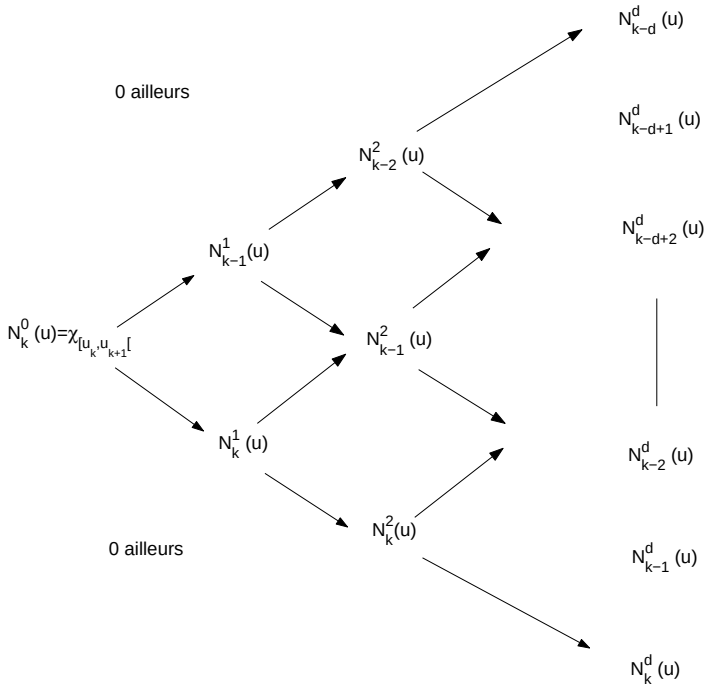

FIG. 3 – *calculs des B-splines de bases*

Ce qui nous donne l'algorithme suivant :

Algorithm 1 calculs des B-splines de bases

Entrée : d, u, $U = [u_0,...,u_m]$
Sortie : $N = [N_0^d(u),...,N_n^d(u)]$
Algorithme :

```
n:=m-d-1;
initialiser N[0..n] à 0;

Si u = u_0 alors
   Si u_0 est de multiplicité d+1 alors N[0]:=1; Fin Si;
   retourner N;
Fin Si;

Si u = u_m alors
   Si u_m est de multiplicité d+1 alors N[n]:=1; Fin Si;
   retourner N;
Fin Si;

Trouver k tel que u_k ≤ u < u_{k+1};
N[k]:=1;

Pour p de 1 à d faire
    N[k-d]:=(u_{k+1}-u)/(u_{k+1}-u_{k-d+1}) N[k-d+1];
    temp:=(u-u_{k-d+1})/(u_{k+1}-u_{k-d+1});

    Pour i de k-d+1 à k-1 faire
        N[i]:=temp*N[i];
        temp:=(u-u_{i+1})/(u_{i+d+1}-u_{i+1});
        N[i]:=N[i]+(1-temp)*N[i+1];
    Fin Pour;

    N[k]:=(u-u_k)/(u_{k+d}-u_k) N[k];
Fin Pour;
```

Remarque : Cet algorithme est utilisé pour remplir les matrices qui interviennent dans des problèmes d'interpolations ou d'approximations mais on ne l'utilise pas pour évaluer la courbe en un point donné. Si l'on veut évaluer $P(u)$ on utilise plutôt l'algorithme de De Boor (ci-dessous) qui est beaucoup plus stable numériquement et moins coûteux en opérations.

Algorithme d'évaluation de De Boor. Je ne démontrerais pas cet algorithme (cf. [1] ou [2] ou [3] pour une démonstration).

Algorithm 2 Algorithme de De Boor

Entrée : d, u, $U = [u_0,...,u_m]$, $[P_0,...,P_n]$
Sortie : $P(u)$
Algorithme :

```
Trouver k tel que u_k ≤ u < u_{k+1};

Pour i de k-d à k faire
    P_i^0 := P_i;
Fin Pour;

Pour j de 1 à d faire
    Pour i de k à k-d+j par pas de -1 faire
        α_i^j := (u − u_{i+1}) / (u_{i+d+1−j} − u_i);
        P_i^j := (1 − α_i^j) P_{i−1}^{j−1} + α_i^j P_i^{j−1};
    Fin Pour;
Fin Pour;

retourner P_k^d;
```

Remarque :

- Il est bien sur possible d'optimiser cet algorithme en ne stockant pas tous les points intermédiaires mais en écrasant à chaque étape les points obtenus à l'étape précédente.
- Cet algorithme suppose que $\left\{ \begin{array}{l} k \geq d \\ k \leq n \end{array} \right.$ qui équivaut à $u \in [u_d,u_n]$, ce qui est en général le cas pour des vecteurs nodaux du type (4). Pour des vecteurs nodaux qui ne sont pas de ce type, une technique consiste à :
  - ajouter au vecteur nodal autant de u_0 pour le rendre de multiplicité $d+1$ et à en ajouter autant de points de contrôle nulle aux début de la liste des points de contrôles.
  - idem pour u_m.
    Exemple :
    si $d = 3$, $P = [P_0,P_1,P_2,P_3]$,$U = [0,1,2,3,4,5,6,6]$ alors avant d'injecter U et P dans l'algorithme de De Boor, on fait :
    $U := [0,0,0,0,1,2,3,4,5,6,6,6,6]$ et $P := [0,0,0,P_0,P_1,P_2,P_3,0,0]$.

– Les points P_i^j intermédiaires générés par cet algorithme permettent de faire l'algorithme de subdivision ci-dessous.

Subdivision. L'algorithme de De Boor opère selon le schéma suivant (les flèches indiquent une combinaison affine entre 2 points) :

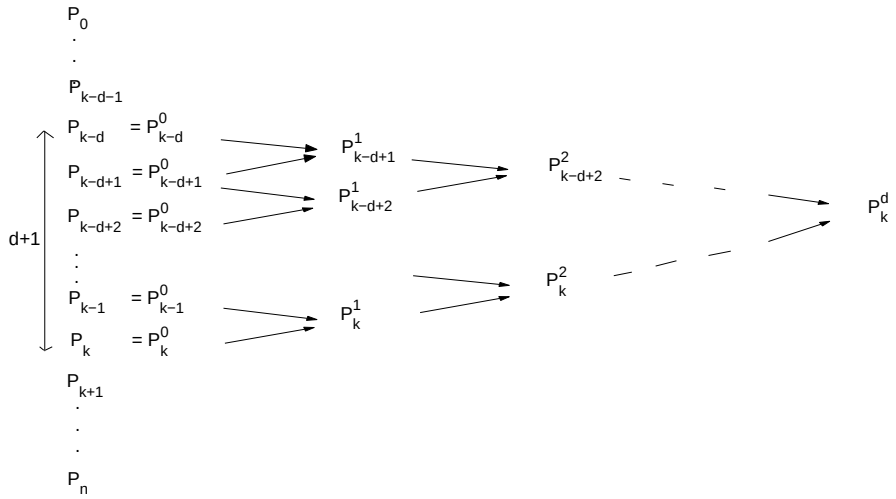

FIG. 4 – *Schéma de l'algorithme de De Boor*

L'algorithme de subdivision stocke les points extrêmes générés par l'algorithme de De Boor ($[P_0,...,P_{k-d-1},P^0_{k-d},P^1_{k-d+1},...,P^d_k]$ d'une part, $[P^d_k,P^{d-1}_k,...,P^0_k,P_{k+1},...,P_n]$ d'autre part) pour contruire 2 autres courbes B-splines de même degré que la courbe B-spline initiale indépendantes l'une de l'autre et ces 2 courbes mises bout-à-bout redonnent la courbe initale (cf l'exemple ci-dessous).

L'algorithme est le suivant :

Pour mieux comprendre l'algorithme de subdivision, appliquons-le sur un exemple :

Entrée : d, u, $U = [u_0,...,u_m]$, $P = [P_0,...,P_n]$
Sortie :
U_1, P_1 correspondant à la 1ère courbe;
U_2, P_2 correspondant à la 2ème courbe.
Algorithme :

`Trouver k tel que` $u_k \leq u < u_{k+1}$;

`// ` U_1 `est de taille k+d+2`
$U_1[0..k] := U[0..k]$;
$U_1[k+1..k+d+1] := u$;

`// ` U_2 `est de taille n+2d-k+1`
$U_2[0..d] := x$;
$U_2[d+1..n+2d-k] := U[k+1..m]$;

`// ` P_1 `est de taille k+1`
$P_1[0..k-d-1] := P[0..k-d-1]$;
$P_1[k-d+1..k] := [P^0_{k-d}, P^1_{k-d+1}, ..., P^d_k]$;

`// ` P_2 `est de taille n-k+d`
$P_2[0..d] := [P^d_k, P^{d-1}_k, ..., P^0_k]$;
$P_2[d+1..n-k+d-1] := P[k+1..n]$;

FIG. 5 – *Algorithme de subdivision pour les courbes B-splines*

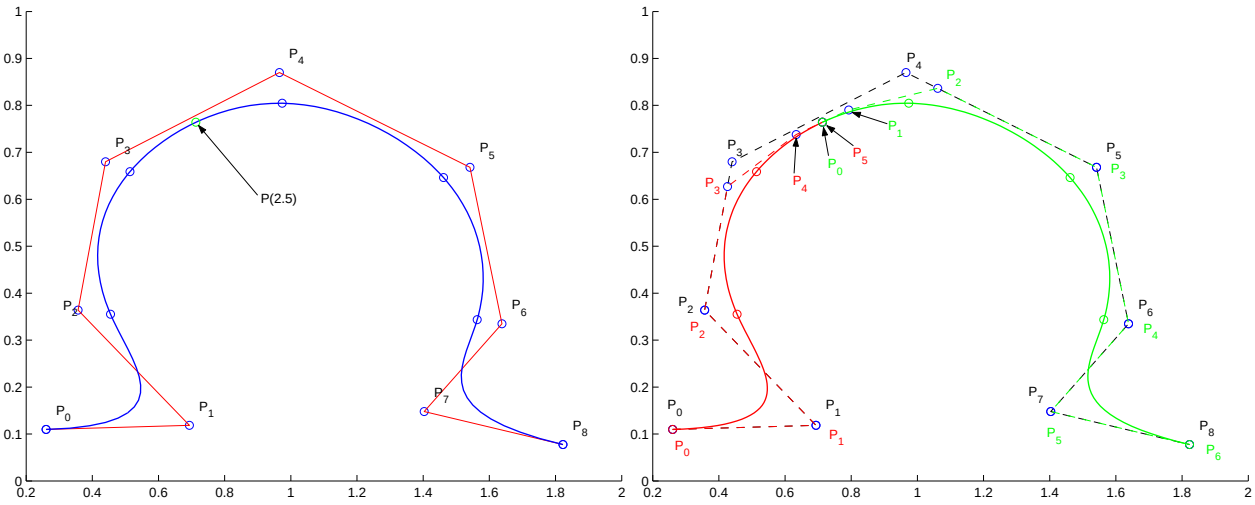

FIG. 6 – *Courbe B-spline pour le vecteur nodal* [0,0,0,0,1,2,3,4,5,6,6,6,6]*(courbe initiale à gauche et subdivision en* $u = 2.5$ *à droite)*

Sur la courbe de droite nous voyons bien que l'on a 2 courbes B-splines distinctes et que certains points de contrôle sont restés inchangés par rapport à la figure initiale.

Insertion d'un noeud. Le but de cet algorithme est d'insérer un nouveau noeud sans changer la forme ni le degré de la courbe B-spline; à cause de la contrainte fondamentale (3), si l'on insére un nouveau noeud en laissant le degré inchangé alors on doit forcément ajouter un nouveau point de contrôle. En fait, nous allons voir que certains points de contrôle sont supprimés et remplacés par d'autres. Le fait d'insérer un nouveau noeud ajoute des nouveaux paramètres (point de contrôle+noeud) et donc ajoute de la fléxibilité à notre courbe (i.e. augmente le degré de liberté de la courbe).

Algorithm 3 Insertion d'un noeud pour les courbes B-splines

Entrée : d, u, $U = [u_0,...,u_m]$, $P = [P_0,...,P_n]$
Sortie : $\overline{U}$, $\overline{P}$ correspondant à la nouvelle courbe;
Algorithme :

Trouver k tel que $u_k \leq u < u_{k+1}$;

// $\overline{U}$ est de taille m+2
$\overline{U}[0..k] := U[0..k]$;
$\overline{U}[k+1] := u$;
$\overline{U}[k+2..m+1] := U[k+1..m]$;

// $\overline{P}$ est de taille n+2
Pour i de 1 à n+1 faire

$$\alpha_i := \begin{cases} 1 & si \quad i \leq k-d \\ \frac{u-u_i}{u_{i+d}-u_i} & si \quad k-d+1 \leq i \leq k \\ 0 & si \quad i \geq k+1 \end{cases} ;$$

$\overline{P}_i := \alpha_i P_i + (1-\alpha_i) P_{i-1}$;
Fin Pour;

Exemple :

Élévation de degré. Les algorithmes de subdivision et d'insertion de noeuds permettent d'augmenter le degré de liberté d'une courbe B-spline. Un autre moyen d'y arriver est d'utiliser l'algorithme d'élévation de degré qui, comme les 2 algorithmes précédents, laisse la courbe inchangée mais ajoute des paramètres supplémentaires. Plus précisément, soit k le nombre de noeuds distincts dans le vecteur nodal, alors l'algorithme d'élévation de degré ajoute k noeuds au vecteur nodal (en augmentant la multiplicité de chaque noeud) et $k-1$ points de contrôle. Je ne donnerais pas de description détaillée de l'algorithme que j'ai implémenté. Le lecteur trouvera cet algorithme ainsi que la preuve de cet algorithme dans [4].

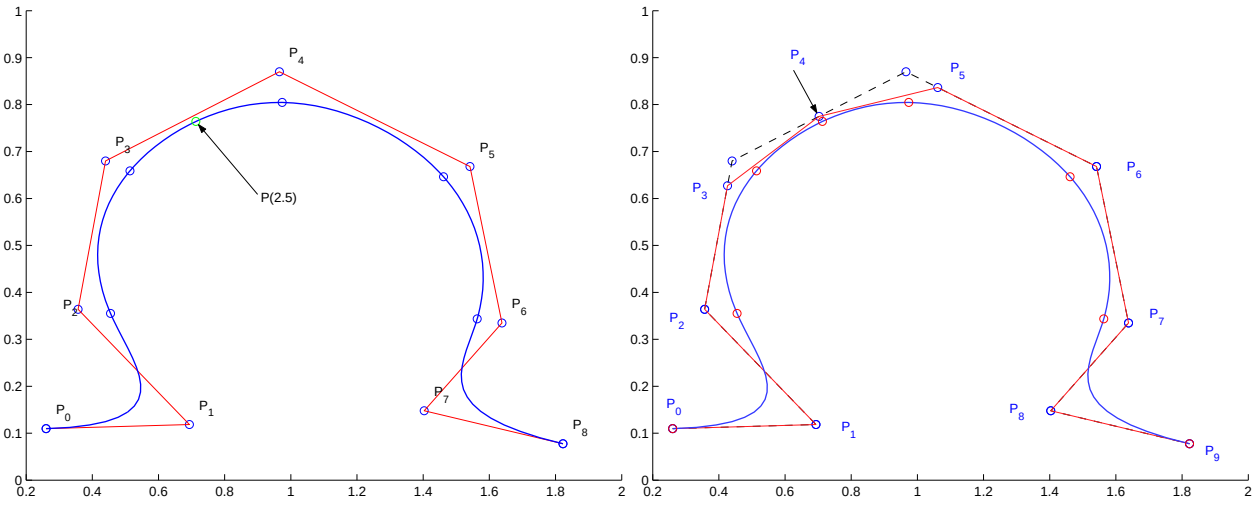

FIG. 7 – *Insertion d'un noeud en* $u = 2.5$

Exemple :

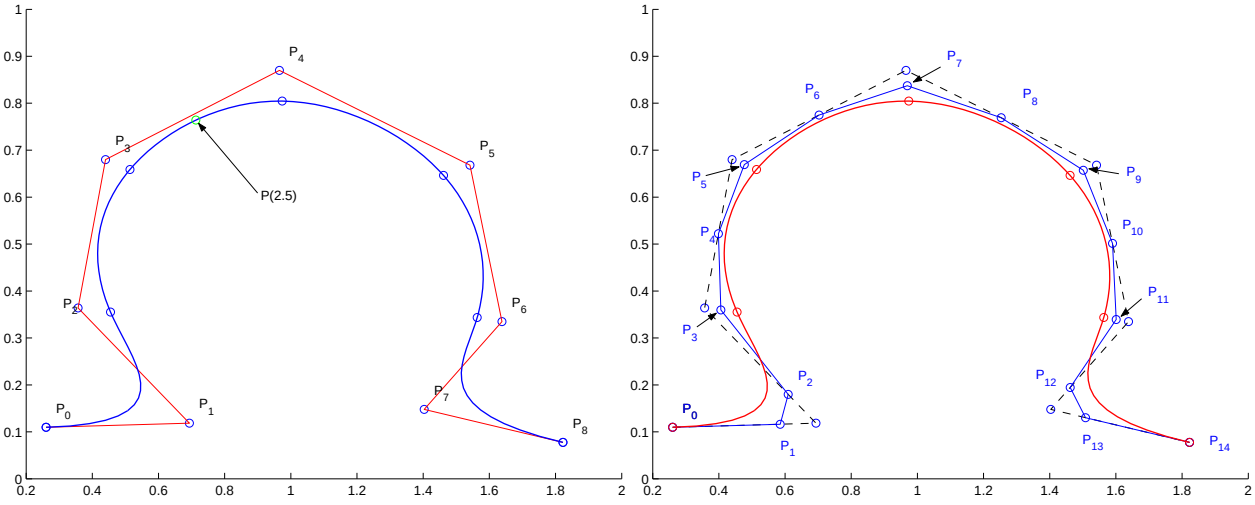

FIG. 8 – *Élévation de degré : courbe initiale à gauche (degré 3) et de degré 4 à droite*

Remarquez qu'après élévation de degré le polygone de contrôle s'est rapproché de la courbe. En répétant l'opération, on obtient ainsi une "bonne" approximation de la courbe :

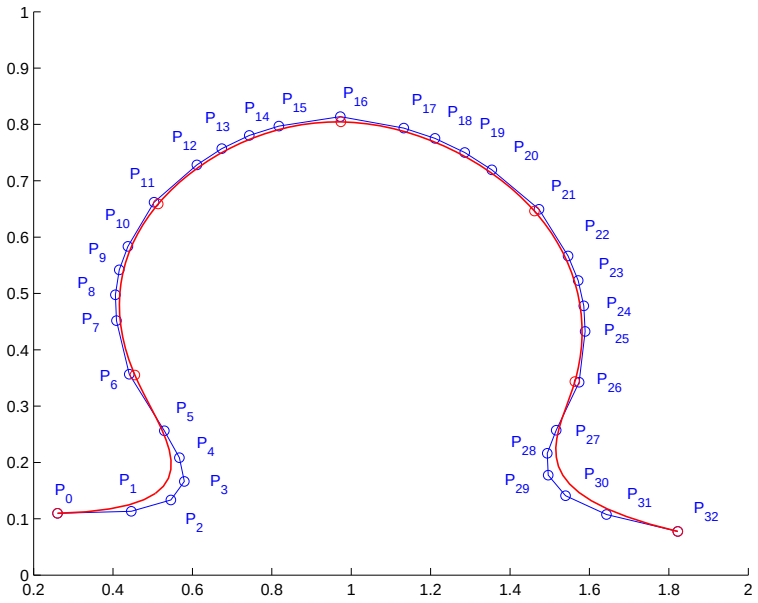

FIG. 9 – *Même courbe après 4 élèvations de degré (courbe de degré 7).*

1.4 Surfaces B-splines

1.4.1 Définition

Une surface B-spline de degrés (d_1,d_2) est la surface paramétrée définie par le produit tensoriel suivant :

$$P(u,v) = \sum_{i=0}^{n_1} \sum_{j=0}^{n_2} N_i^{d_1}(u) N_j^{d_2}(v) P_{ij} \tag{7}$$

où $P_{ij} \in \mathbb{R}^d$, $(d = 2$ ou $d = 3)$.

Pour construire une surface B-spline de degrés (d_1,d_2) à partir des $(n_1+1)(n_2+1)$ points de contrôle P_{ij}, il faut donc se donner $(m_1+1)+(m_2+1)$ noeuds où $m_1 = n_1 + d_1 + 1$ et $m_2 = n_2 + d_2 + 1$, permettant de définir les fonctions de bases $N_i^{d_1}(u)$ et $N_j^{d_2}(v)$.

1.4.2 Propriétés

En remarquant que $P(u,v)$ peut s'écrire :

$$P(u,v) = \sum_{i=0}^{n_1} N_i^{d_1}(u) \left(\sum_{j=0}^{n_2} N_j^{d_2}(v) P_{ij} \right) = \sum_{i=0}^{n_1} N_i^{d_1}(u) \widetilde{P}_i(v) \tag{8}$$

Toutes les propriétés sur les courbes B-splines s'appliquent aux surfaces B-splines en chaque direction u et v.

1.4.3 Exemple

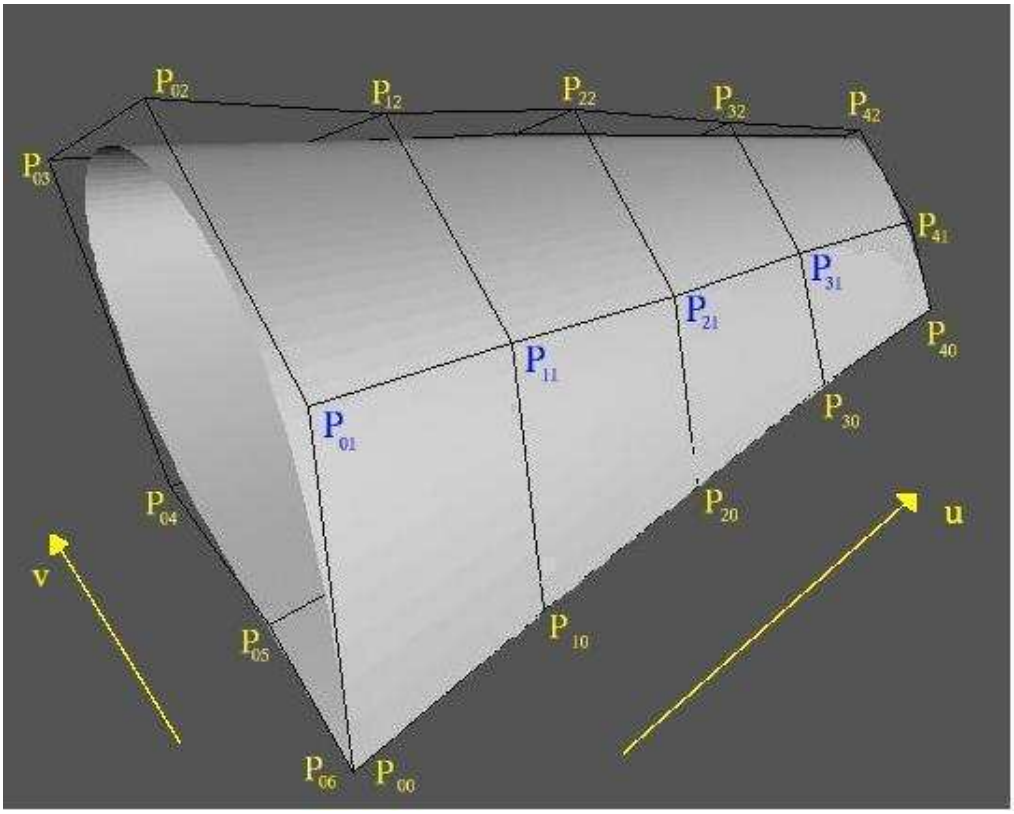

FIG. 10 – *Exemple de surface B-spline.*

1.4.4 Algorithmes de bases sur les surfaces B-splines

Algorithme d'évaluation. De la relation (8), on en déduit l'algorithme suivant :

Algorithm 4 Évaluation d'une surface en un point (u,v)

Entrée : d_1, d_2, (u,v), $U = [u_0,...,u_{m_1}]$, $V = [v_0,...,v_{m_2}]$, $P = \begin{bmatrix} P_{00} & \cdots & P_{0n_2} \\ \vdots & \ddots & \vdots \\ P_{n_1 0} & \cdots & P_{n_1 n_2} \end{bmatrix}$

Sortie : $P(u,v)$

Algorithme :

Évaluer les $n_2 + 1$ points $\widetilde{P}_i(v) = \sum_{j=0}^{n_2} N_j^{d_2}(v) P_{ij}$ à l'aide de l'algorithme de De Boor;

Évaluer $P(u,v) = \sum_{i-0}^{n_1} N_i^{d_1}(u) \widetilde{P}_i(v)$ à l'aide de l'algorithme de De Boor;

Dans la suite, je ne donnerais que des exemples; pour obtenir les algorithmes, il suffit d'appliquer les algorithmes de bases sur les courbes B-splines sur chaque direction u et v.

Subdivision. Exemples :

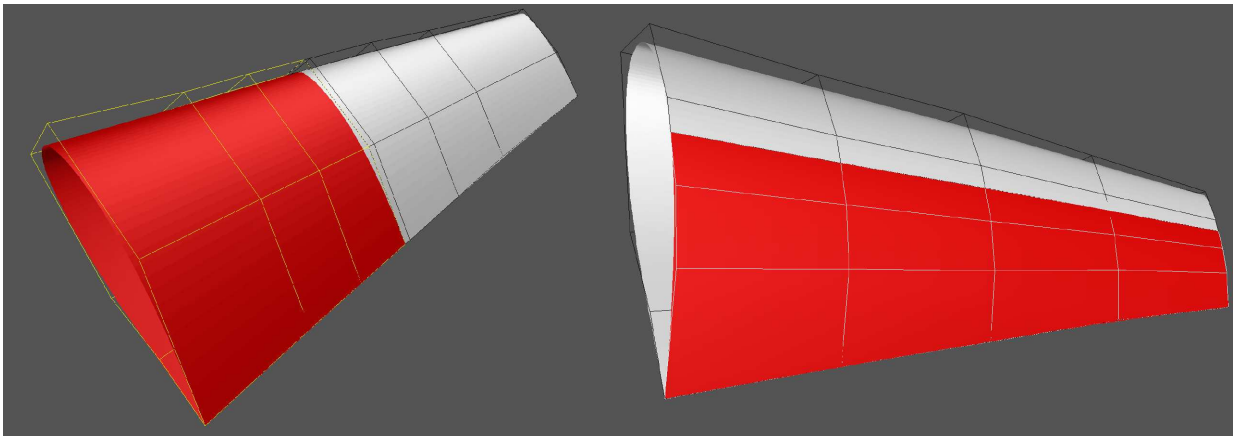

FIG. 11 – *Exemple de subdivision pour les surfaces B-splines : en* $u = 0.9$ *(à gauche) en* $v = 0.6$ *(à droite)*

Insertion d'un noeud. Exemples :

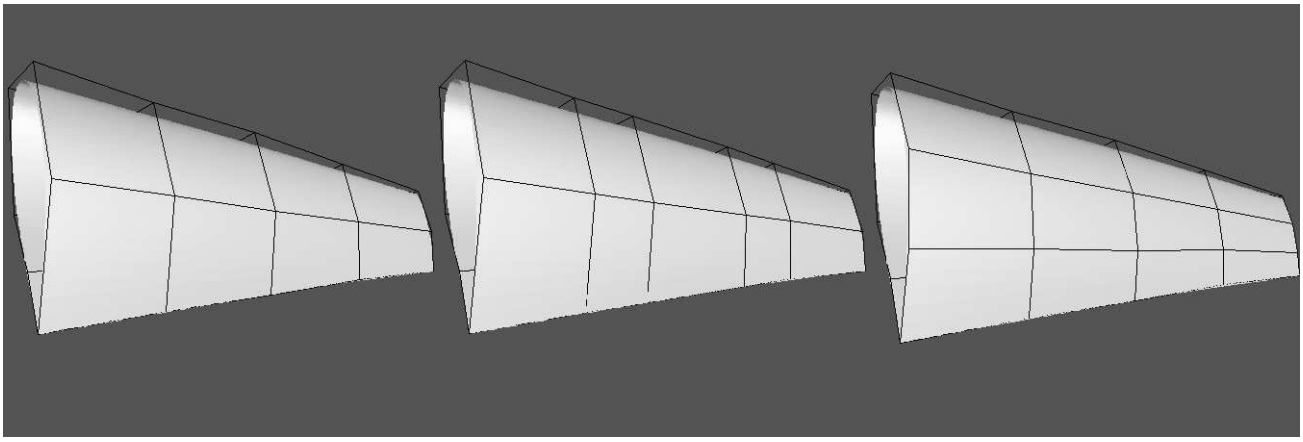

FIG. 12 – *Insertion d'un noeud : courbe initiale (à gauche), insertion en* $u = 0.9$ *(au milieu), en* $v = 0.6$ *(à droite)*

Élévation de degré. Exemples :

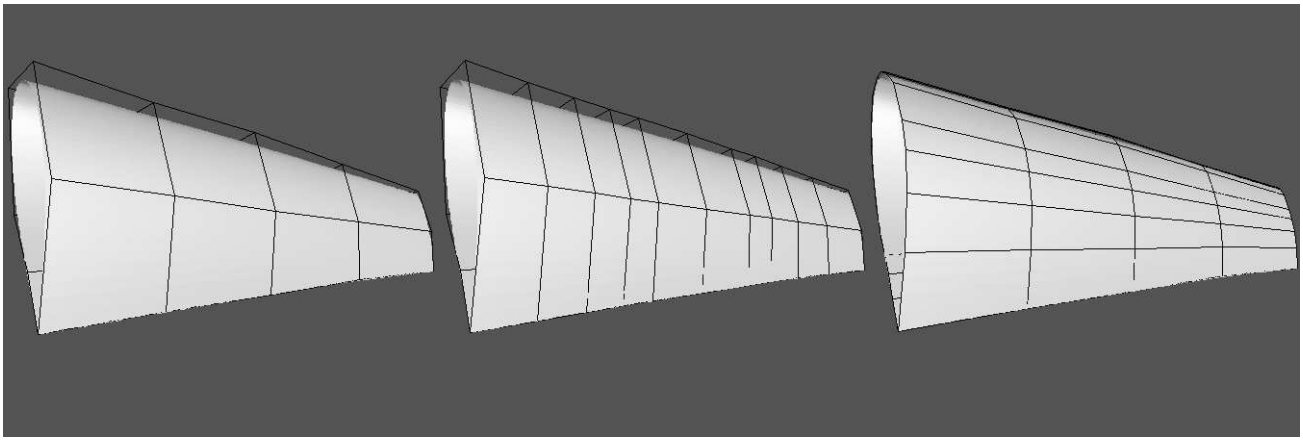

FIG. 13 – *Élévation de degré : courbe initiale (à gauche), 3 élévations de degré selon* u *(au milieu), 3 élévations de degré selon* v *(à droite)*

2 Implémentation

Le langage à utiliser pour l'implémentation (C++) ainsi que le schéma de code à suivre (SYNAPS) étaient imposés par le sujet de stage. Loin d'être un frein, nous allons voir que ces choix ont été d'une grande aide pour le développement d'une bibliothèque qui doit être facilement utilisable dans des codes externes.

2.1 Intérêt du C++

Le langage C++ est de plus en plus utilisé pour l'écriture de librairies mathématiques, non pas pour son caractère orienté objet mais pour la généricité qu'il offre. La facilité de surcharge des opérateurs prédéfinis permet de traduire un algorithme en langage C++ en utilisant des notations assez intuitives (c.f. l'exemple ci-dessous), ce qui a le grand avantage de rendre le code clair et de repousser la frontière entre les types et fonctions définis par l'utilisateur et ceux prédéfinis du langage. De plus le C++ en tant que langage orienté objet permet le polymorphisme, mais il a été choisi, pour des raisons d'efficacité de ne pas utiliser de classes abstraites ni de fonctions virtuelles qui définissent un polymorphisme dynamique. En effet, l'appel d'une fonction virtuelle est une décision prise pendant l'exécution, et même si un bon compilateur minimisera autant que possible le coût d'une telle fonction, celui-ci devient non-négligeable lors d'appels intensifs. De plus une fonction virtuelle ne peut pas être déclarée en ligne, puisque le type de l'objet n'est pas connu à la compilation. C'est pour cette raison que la librairie SYNAPS [5] utilisent de nouvelles techniques inspirées de la STL, bibliothèque appartenant à la distribution standard C++, à base de *template*, ce qui permet un polymorphisme "statique". Elles sont utilisées pour guider la compilation afin d'obtenir un code efficace. Ces nouvelles techniques, appelées *template expressions*, permettent de construire des programmes C++ meilleurs en performance que leurs analogue FORTRAN (c.f. [6]). Pour une analyse poussée de ces techniques de programmation en calcul scientifique on peut lire [7].

Exemple de code C++ templaté surchargeant l'opérateur "*":

```
template <class Vect>
typename Vect::value_type operator*(const Vect& v1, const Vect& v2)
{
   typename Vect::value_type result = 0.0;
   for (int i=0; i<v1.size(); i++)
       result += v1[i]*v2[i];
   return result;
}
```

Le code ci-dessus définit une implémentation du produit scalaire entre 2 vecteurs mais le type de ces vecteurs ainsi que le type de retour ne sont connus par cette fonction qu'au moment de son utilisation car ils sont donnés par l'utilisateur. Les seules contraintes sur ces types sont qu'ils doivent fournir les opérateurs "[]" et "+=" et la fonction membre "size()" ce qui est le cas par exemple pour les types *std::vector¡double¿* et *std::vector¡double¿::value_type*.

2.2 SYNAPS

2.2.1 Présentation

SYNAPS (SYmbolic and Numeric APplicationS)[2] est une bibliothèque développée au sein du projet GALAAD de Sophia-Antipolis en collaboration avec l'équipe Spaces de l'INRIA Lorraine/LIP6. Elle fournit des outils pour manipuler des vecteurs, matrices, polynômes à une ou plusieurs variables, etc... SYNAPS à été développée avec le soucis d'en faire une bibliothèque active dans le sens où l'on a jamais spécifié le type des différents arguments de ces fonctions. C'est le compilateur qui instancie les fonctions dont le programme a effectivement besoin et non le programmeur. Le programmeur se contente de donner des canevas de programme que le compilateur va se charger de compléter. SYNAPS fournie donc des fonctions sous forme d'une implémentation générique et une myriade de spécialisations chacune adaptées à un type de stockage spécifique. La généricité dans SYNAPS est obtenue par le biais d'un usage extrêmement intensif du mécanisme de *template* décrit dans la section précédente.

2.2.2 Architecture de SYNAPS

Le niveau le plus bas, les *conteneurs* et les *domaines* :

Un *conteneur* dans SYNAPS est la définition de la représentation interne d'un objet, le *conteneur* fournit les fonctions de base dont une abstraction aura besoin, se créer, se parcourir, accéder a des données contenues etc... Les *conteneurs* sont implémentés en tant que *class* souvent paramétrées par le type des coefficients qu'elle devront contenir. (exemple : rep1d...)
Un *domaine* est un *namespace* auquel sont attachés des *conteneurs*. Il implémente aussi des algorithmes spécialisés pour ces *conteneurs*. (exemple : linalg...)

Le niveau au dessus, les *vues* :

Les *vues* sont les interprétations que l'on peut donner à un *conteneur*. Elles définissent la sémantique mathématique du *conteneur*. Par exemple, un *conteneur rep1d* peut aussi bien être interprété comme un polynôme univarié (*UPolDse¡double, linalg::rep1d¡double¿ ¿*), que comme un vecteur en algèbre linéaire (*VectDse¡double, linalg::rep1d¡double¿ ¿*).

Les *modules* :

Les *modules* sont des collections de fonctions qui s'appliquent à des objets partageant des propriétés communes.

2. http://www-sop.inria.fr/galaad/logiciels/synaps/

2.3 Mise en oeuvre

Concrètement, j'avais donc à développer un *module* BSPLINE et un *domaine* bspline. Le *module* BSPLINE contient les implémentations génériques des algorithmes qui sont décrits dans le chapitre précédent. Le *domaine* bspline contient un *conteneur* pour les courbes B-splines (*bspline::curve*) et un autre pour les surfaces B-splines (*bspline::surface*), ainsi que les fonctions qui s'appliquent à ces *conteneurs*; ces fonctions sont simplement des interfaces pour pouvoir utiliser les fonctions implémentées dans le *module* BSPLINE avec ces conteneurs.

2.3.1 Le conteneur *bspline::curve*

D'après le chapitre précédent, nous savons que pour représenter une courbe B-spline, il nous faut : un degré, un vecteur nodal et des points de contrôle. Pour définir un objet de type "courbe B-spline", il suffit alors de construire la *class* bspline::curve suivante :

```
namespace bspline
{
   template<class C, class N, class Pt>
   struct curve
   {
      int degree_;
      Pt points_;
      N nodes_;
   };
}
```

Remarquez que cette *class* est templaté par 3 autres *class* représentant le type des scalaires (typiquement *C=double*), le type du vecteur nodal (typiquement *N=VectDse¡C¿*) et le type des points de contrôle (typiquement *Pt=VectDse¡VectDse¡C¿ ¿*). Notre *class* bspline::curve fournit bien sûre les constructeurs et opérateurs (affectations, ..) nécessaires à une bonne manipulation de cet objet. Pour un exemple d'utilisation de cette *class*, c.f. la section "exemples d'utilisation" ci-dessous.

2.3.2 Le conteneur *bspline::surface*

Pour représenter une surface B-spline, il nous faut : 2 degrés, 2 vecteurs nodaux et des points de contrôles :

```
namespace bspline
{
   template<class C, class N, class Pt>
   struct surface
   {
      int degree1_;
```

```
        int degree2_;
        Pt points_;
        N nodes1_;
        N nodes2_;
    };
}
```

Pour un exemple d'utilisation de cette *class*, c.f. la section "exemples d'utilisation" ci-dessous.

2.3.3 Exemples d'utilisation

Je fournis ici des exemples de manipulations de courbes et surfaces B-splines à l'aide de ma bibliothèque. Le lecteur trouvera aussi en annexe les prototypes des *class* et fonctions que j'ai implémenté.

Manipulations de courbes bsplines.

```
#include "synaps/geometry/bspline.H"
#include <iostream>
#include "synaps/linalg/array1d.H"
#include "synaps/linalg/VectDse.H"

typedef VectDse<double> N;
typedef VectDse< VectDse<double> > Pt;
using namespace std;
//-------------------------------------------------------------------------
int main()
{
   typedef bspline::curve<double,N,Pt> bspline;
   N nodes(10, "0 0 0 0 1 2 3 3 3 3");
   VectDse<double> P0(2, "0.41705 0.10673"), P1(2, "0.28802 0.60088"),
                   P2(2, "0.90553 0.46053"), P3(2, "1.3203 0.79678"),
                   P4(2, "1.8041 0.63012"),  P5(2, "1.477 0.13012");
   Pt points(6);
   points[0] = P0; points[1] = P1; points[2] = P2;
   points[3] = P3; points[4] = P4; points[5] = P5;

   // declaration et definition de la courbe B-spline
   bspline b;
   b.degree_ = 3;
   b.nodes_  = nodes;
   b.points_ = points;
```

```
    // ecriture dans un fichier pour affichage
    ofstream out_file;

    out_file.open("curve.dat");
    drawCurve(out_file,b);
    out_file.close();

    out_file.open("polyline.dat");
    drawPoly(out_file,b);
    out_file.close();

    // insertion d'un noeud en u=1.5
    insert1Node(b, 1.5);

    // elevation de degre
    bspline bb;
    degreeElevate(bb, b);

    // etc....
    return 0;
}
```

On peut alors afficher la courbe avec par exemple un logiciel tel que Gnuplot.

Manipulations de surfaces bsplines.

```
#include "synaps/geometry/bspline.H"
#include <iostream>
#include "synaps/linalg/array1d.H"
#include "synaps/linalg/MatrDse.H"

typedef VectDse<double> N;
typedef MatrDse< VectDse<double> > Pt;
using namespace std;
//-----------------------------------------------------------------------------
int main()
{
  typedef bspline::surface<double,N,Pt> bspline;
  N nodes1(6, "0 0 0 1 1 1");
  N nodes2(6, "0 0 0 1 1 1");
  VectDse<double> P00(3, "-0.390000 -0.309921 -1.000000"),
                  P10(3, "0.030000 -0.790000 0.350000"),
                  P20(3, "-0.260000 0.200000 -0.020000"),
                  P01(3, "0.650000 -0.250000 -0.490000"),
```

```
                    P11(3, "0.290000 -0.400000 -0.650000"),
                    P21(3, "0.600000 0.980000 0.580000"),
                    P02(3, "-0.363017 0.758630 -0.243552"),
                    P12(3, "0.430000 0.410000 -0.500000"),
                    P22(3, "-0.110000 -0.340000 0.960000 ");
    Pt points(3,3);
    points(0,0) = P00; points(0,1) = P01; points(0,2) = P02;
    points(1,0) = P10; points(1,1) = P11; points(1,2) = P12;
    points(2,0) = P20; points(2,1) = P21; points(2,2) = P22;

    // declaration et definition de la surface B-spline
    bspline b;
    b.degree1_ = 2;
    b.degree2_ = 2;
    b.nodes1_  = nodes1;
    b.nodes2_  = nodes2;
    b.points_  = points;

    // ecriture dans un fichier pour affichage
    ofstream out_file;


    out_file.open("surface.dat");
    drawSurface(out_file,b);
    out_file.close();


    out_file.open("polyline.dat");
    drawPoly(out_file,b);
    out_file.close();

    // elevation de degre selon la direction u
    bspline bb;
    degreeElevate1(bb, b);


    // insertion d'un noeud en v=0.5
    insert1Node2(b,0.5);

   return 0;
  }
```

Il suffit alors de taper la commande "geomwiew surface.dat polyline.dat" pour obtenir un affichage graphique de la surface.

3 Application à l'optimisation géométrique

Disposant désormais d'une bibliothèque complète sur les courbes et surfaces B-splines, la deuxième partie de mon stage consistait à utiliser cette bibliothèque pour transposer aux courbes B-splines ce qui a été fait avec les courbes de Bézier sur la paramétrisation adaptative pour l'optimisation de forme (c.f [8] et [9]).

3.1 Présentation

Je reprends ici l'exemple traité par J.-A. Désidéri et F. Bélahcène pour simuler une optimisation couplée à une adaptation de la paramétrisation :

- Trouver un arc de courbe $\mathcal{C}$ ayant une tangente verticale en $x = 0$, reliant les points (0,0) et (1,0) et minimisant le critère suivant : $J = \frac{\mathcal{P}^2}{\mathcal{A}}$ où $\mathcal{P}$ est le périmètre de $\mathcal{C}$ et $\mathcal{A}$ la surface sous la courbe.
- Minimiser J revient à minimiser P et maximiser A. On cherche donc une courbe ayant la plus grande surface sous la contrainte d'avoir le plus petit périmètre. On montre que la partie supérieur du cercle de centre $(\frac{1}{2},0)$ et de rayon $\frac{1}{2}$ est l'unique solution de ce problème. J admet alors pour minimum 2π. On s'intéresse au cas au l'espace de recherche de f est l'ensemble des courbes B-spline (c.f [8] et [9] pour le cas où l'ensemble de recherche est celui des courbes de Bézier).

3.2 Optimisation couplée à une adaptation de la paramétrisation avec des courbes de Bézier (Rappels)

La méthode mise en oeuvre dans [8] et [9] pour résoudre le problème précédent consiste à construire un procédé itératif qui va converger vers la solution :

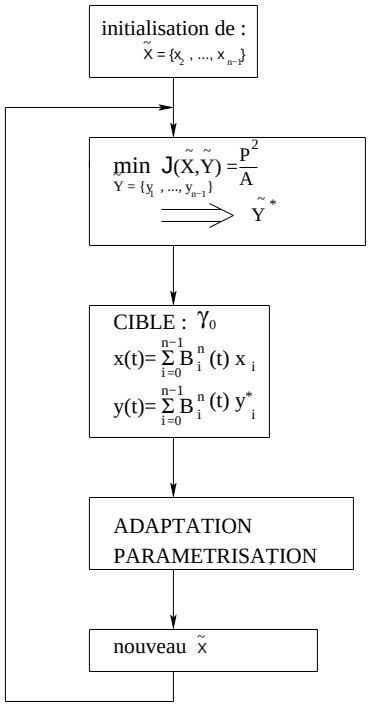

FIG. 14 – *Simulation d'une optimisation couplée à une adaptation de la paramétrisation dans le cas des courbes de Bézier*

La pierre angulaire de cette méthode est l'adaptation de la paramétrisation. Elle consiste à trouver parmi toutes les paramétrisations de la courbe cible γ_0, celle qui est la plus régulière au sens où son polygone de contrôle varie le moins possible.

3.3 Adaptation de la paramétrisation avec des courbes B-splines.

Disposant d'une courbe B-spline

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \sum_{i=0}^{n} N_i^d(t) \begin{pmatrix} \widetilde{x}_i \\ \widetilde{y}_i \end{pmatrix}, t \in [0,1]$$

(en fait on dispose d'un certain nombre de points $(Q_i = (Qx_i, Qy_i))_{i=0..N}$), on désire trouver une nouvelle paramétrisation de la même courbe

$$P(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \sum_{i=0}^{n} N_i^d(t) \begin{pmatrix} x_i \\ y_i \end{pmatrix}, t \in [0,1] \qquad (9)$$

avec un polygone de contrôle qui ne varie pas trop.

On a donc des points $(Q_i)_{i=0..N}$et on cherche une courbe b-spline qui passe près de ces points sous la contrainte que son polygone de contrôle varie le moins possible. On peut exprimer ceci par le problème des moindres carrés suivant :

$$\min_{P_0,\ldots,P_n} \sum_{i=0}^{n} \|P(t_i) - Q_i\|_2^2 \tag{10}$$

Minimiser $E = \sum_{k=0}^{N} \|P(t_k) - Q_k\|_2^2 = \sum_{k=0}^{N} P(t_k)^2 - 2P(t_k).Q_k + Q_k^2$ revient à annuler son gradient :

$$\frac{\partial E}{\partial P_i} = 0 \Longleftrightarrow \sum_{k=0}^{N} P(t_k) N_i^d(t_k) = \sum_{k=0}^{N} Q_k N_i^d(t_k) \tag{11}$$

avec :

$$P(t_k) = \sum_{j=0}^{n} N_j^d(t_k) P_j \tag{12}$$

On prend de plus $x_0 = x_1 = y_0 = y_n = 0$ et $x_n = 1$ pour que la courbe passe en (0,0) et (1,0) avec une tangente verticale en (0,0).

En injectant (12) dans (11) on obtient les 2 systèmes linéaires suivants :

$$\left\{ \begin{array}{l} A^T A X = A^T W_x \\ B^T B Y = B^T W_y \end{array} \right. \tag{13}$$

avec :

$$A = \left(N_j^d(t_i)\right)_{i=0..N,\ j=2..n-1},\ X = (x_i)_{i=2..n-1},\ W_x = \left(Qx_i - N_n^d(t_i)\right)_{i=0..N}$$

$$B = \left(N_j^d(t_i)\right)_{i=0..N,\ j=1..n-1},\ Y = (y_i)_{i=1..n-1},\ W_y = (Qy_i)_{i=0..N}$$

Pour construire les matrices qui interviennent dans ces systèmes, nous devons choisir des $(t_i)_{i=0..N}$ et des noeuds $(u_i)_{i=0..n+d}$. Un choix classique consiste à prendre :

$$t_0 = 0, t_j = \frac{\sum_{i=1}^{j} \|Q_i - Q_{i-1}\|}{\sum_{i=1}^{N} \|Q_i - Q_{i-1}\|}, \text{ pour } j = 1...N$$

Une fois ce choix fait on prend alors :

$$u_0 = ... = u_d,\ u_{n+1} = ... = u_{n+d+1} \text{ et } u_{k+d} = \frac{t_k + ... + t_{k+d+1}}{d},\ k = 1..n-d$$

On montre que pour ce choix, les matrices A^TA et B^TB qui interviennent dans les systèmes précédents sont inversibles.

En résolvant ainsi ces systèmes, on obtient les nouveaux points de contrôle qui nous donnent une paramétrisation de la courbe B-spline qui passent près (au sens des moindres carrées) des points Q_i.

Remarque : je n'ai pas ajouté explicitement la contrainte "le polygone de contrôle varie le moins possible" dans (10); ceci par manque de temps et parce qu'à première vue les résultats obtenues (c.f ci-dessous) montrent que la variation du polygone de contrôle reste raisonnable pour de faibles degrés ($d \leq 5$):

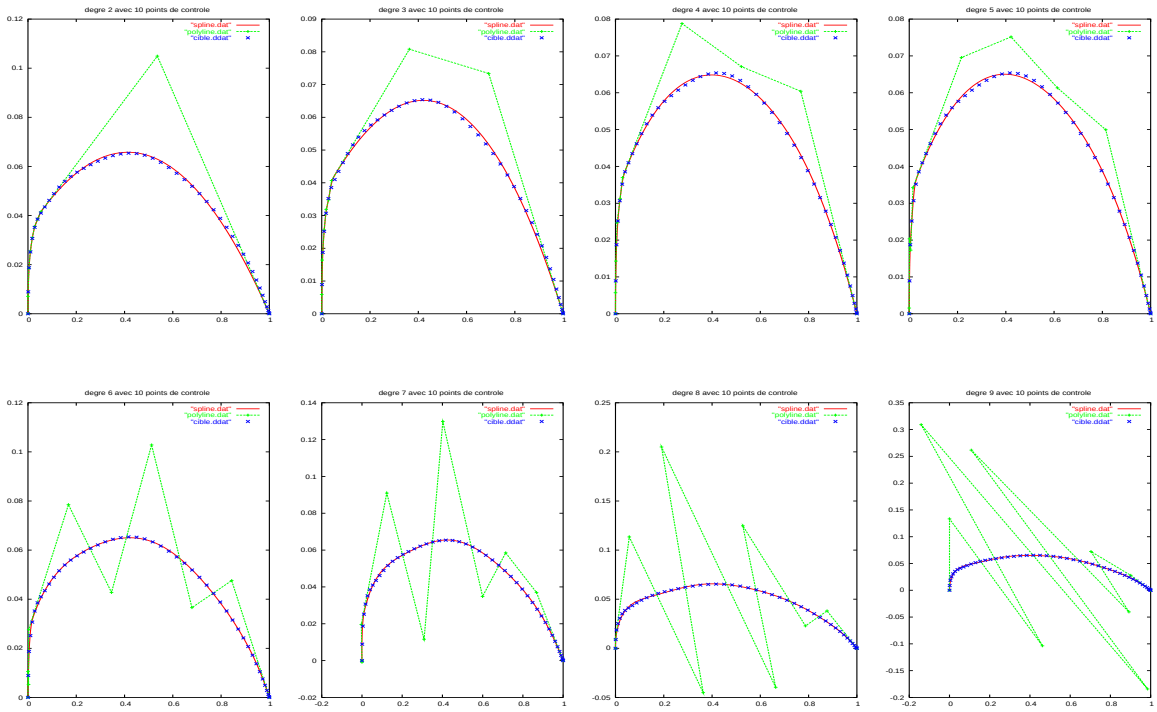

FIG. 15 – *Teste de la paramétrisation*

3.4 Optimisation couplée à une adaptation de la paramétrisation avec des courbes B-spline

3.4.1 Optimisation du critère J

Pour évaluer J, il nous faut calculer le périmètre $\mathcal{P}$ et la surface $\mathcal{A}$ d'une courbe B-spline $(x(t),y(t))$ de type (9):

$$\mathcal{P} = \int_0^1 f(t)dt \text{ avec } f(t) = \sqrt{x'(t)^2 + y'(t)^2}$$

Pour calculer cette intégrale, j'ai utilisé la méthode de Simpson (qui est d'ordre 4) avec la méthode suivante pour évaluer $f(t)$: calculer la dérivée de la spline, évaluer cette spline en t avec l'algorithme de De Boor, prendre sa norme.

$$\mathcal{A} = \int_0^1 y(t)x'(t)dt$$

En remplaçant $y(t)$ et $x(t)$ par leur expression, le calcul de $\mathcal{A}$ nous amène à calculer des intégrales de la forme :

$$\int_0^1 N_i^d(t)N_j^{d-1}(t)dt \tag{14}$$

En remarquant que l'expression à l'intérieur de l'intégrale est un polynôme de degré $2d-1$, en utilisant la méthode de Gauss-Legendre sur d points, on obtient la valeur exacte de cette intégrale. De plus, compte-tenu des supports des fonctions de bases :

$$N_i^d N_j^{d-1} \neq 0 \Longleftrightarrow t \in supp\left(N_i^d\right) \bigcap supp\left(N_j^d\right) = [u_i,u_{i+d+1}] \bigcap [u_j,u_{j+d}]$$

Ces 2 intervalles ne s'intersectent pas si :

$$(u_{i+d+1} \leq u_j \text{ ou } u_{j+d} \leq u_i) \Longleftrightarrow (i+d+1 \leq j \text{ ou } j+d \leq i)$$

donc ils intersectent si :

$$(i+d+1 > j \text{ et } j+d > i) \Longleftrightarrow i+1-d \leq j \leq i+d$$

Ce qui nous réduit considérablement le nombre d'intégrales du type (14) à calculer.

Sachant maintenant calculer $J(X_0,Y)$, pour calculer son minimum, j'ai utilisé la méthode du simplex (Nelder-Mead Simplex, c.f. [10]) qui ne fait que des évaluations de J et aucun calculs directs ou indirects du gradient de J.

3.4.2 Résultats

En couplant l'optimisation de J avec l'adaptation de la paramétrisation comme expliqué plus haut, j'obtiens les résultats suivants :

itération	J
1	6.3632
2	6.2941
3	6.2903
4	6.2842
5	6.284
6	6.284
7	6.2839
8	6.2839

TAB. 1 – *Convergence de J ($n = 10$, $d = 3$, $N = 10$)*

On note donc bien que le critère J converge vers $2\pi \simeq 6.2832$ avec une erreur de l'ordre de 10^{-4} qui est aussi l'erreur faite par la méthode du simplex. A l'issu de ces itérations , on obtient la figure suivante :

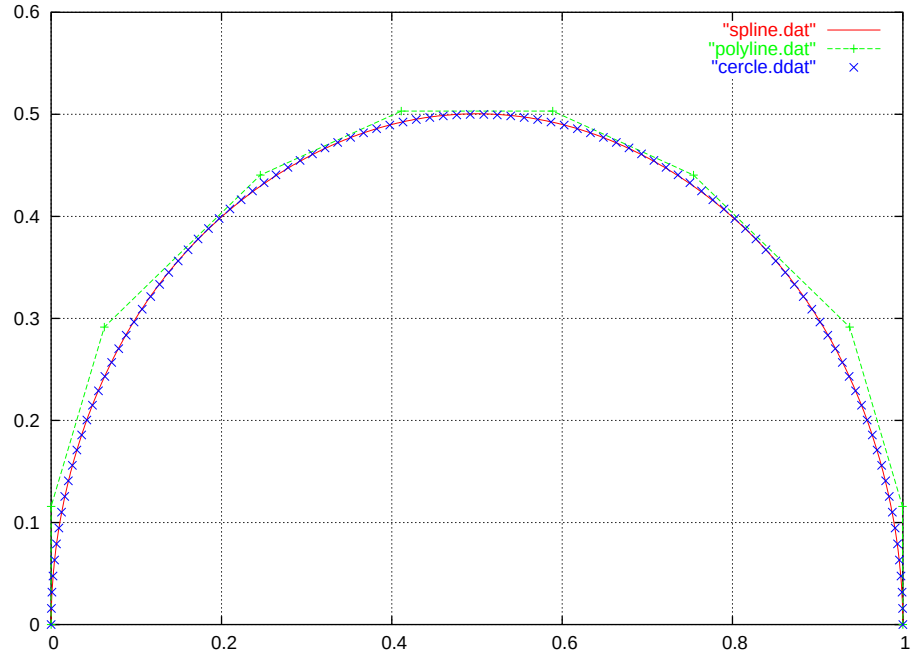

FIG. 16 – *Solution exacte (cercle) et solution approchée*

Cette méthode a cependant des limites qui viennent bien sur d'une part du fait que j'ai choisi un critère des moindres carrées discrets, d'autre part du fait que je n'ai pas posé explicitement la contrainte de faible variation du polygone de contrôle. En effet, j'observe que pour n petit la méthode semble converger puis diverge (et même quelquefois osciller autour d'une valeur) et pour un degré d grand, le polygone de contrôle varie énormément :

itération	J
1	6.3361
2	6.5839
3	6.6306
4	6.6883
5	6.7551
6	6.828
7	6.9004
8	6.9644

TAB. 2 – *Convergence puis divergence de J ($n = 10$, $d = 5$, $N = 10$)*

Remarque : D'un point de vue théorique, l'optimisation couplée à une adaptation de la paramétrisation n'a aucune raison de converger vers le minimum de J. On "espère" juste que l'on se rapproche près de ce minimum et que l'influence de l'adaptation de la paramétrisation ne prime pas sur l'optimisation. L'intérêt de cette méthode est bien sur de diminuer par 2 le nombre d'inconnus qui interviennent dans l'optimisation (fixer X, optimiser sur Y).

Conclusion

Ce stage que j'ai eu le plaisir d'effectuer à l'INRIA de Sophia-Antipolis m'a permis de développer une bibliothèque mathématique (sur les B-splines) de A à Z. Ce stage a donc constitué une magnifique application et une extension du cours "modélisation géométrique" que j'ai suivis durant ma quatrième année à l'INSAT. Il m'a fallut part ailleurs m'initier à un nouveau langage informatique, le C++ que je ne connaissais pas. Enfin, ce stage m'a permis d'aborder de manière simplifiée le domaine de l'optimisation géométrique et l'optimisation de forme que je ne connaissais pas non plus.

Remerciements

Je remercie vivement M. Bernard Mourrain qui m'a permis d'effectuer ce stage et qui m'a accordé son aide et sa confiance pendant ces trois mois. Je remercie également tout le personnel et les stagiaires du projet GALAAD pour leur aide et avec qui j'ai eu des discussions intéressantes et fructueuses.

Références

[1] Gérald E. Farin, *Curves and surfaces for CAGD a practical guide.* - 5th ed - Morgan Kaufmann - 2002.

[2] Hartmut Prautzsch, *Bézier and B-spline techniques* - Springer - 2002.

[3] Jean-Jacques Risler, *Méthodes mathématiques pour la CAO* - Masson - 1991.

[4] Elaine Cohen, Tom Lyche et Larry L.Schumaker, *Algorithms for Degree-Raising of Splines* - ACM Transactions on Graphics, Vol. 4, No. 3, Pages 171-181 - July 1985.

[5] G. Dos Reis, B. Mourrain, F. Rouillier et Ph. Trébuchet, *An environment for Symbolic and Numeric Computation* . - INRIA Sophia-Antipolis - Avril 2002.

[6] T. Veldhuizen, *Blitz++* - http://monet.uwaterloo.ca/blitz/ - 1999.

[7] G. Dos Reis, *Vers une nouvelle approche du calcul scientifique en C++.* - Rapport de Recherche 3362, INRIA - 1998.

[8] A. Clarich et J.-A. Désidéri, *Self-adaptative parameterisation for aerodynamic optimum-shape design* - Rapport de recherche 4428 - INRIA - Mars 2002.

[9] *Hierarchical optimum-shape algorithms using embedded Bézier parameterizations.* NUMERICAL METHODS FOR SCIENTIFIC COMPUTING VARIATIONAL PROBLEMS AND APPLICATIONS Y. Kuznetsov, P. Neittanmaki and O. Pironneau (Eds.) CIMNE, Barcelona - 2003.

[10] Lagarias, J.C., J. A. Reeds, M. H. Wright, and P. E. Wright, *Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions* - SIAM Journal of Optimization, Vol. 9 Number 1, pp.112-147 - 1998.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399