FRANCISCO CARLOS VAZQUEZ DE GARCIA

FABRICAÇÃO DE PROPULSORES NAVAIS PLÁSTICOS AUXILIADA POR COMPUTADOR

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

São Paulo
1999

FRANCISCO CARLOS VAZQUEZ DE GARCIA

FABRICAÇÃO DE PROPULSORES NAVAIS PLÁSTICOS AUXILIADA POR COMPUTADOR

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Mestre em Engenharia.

Área de concentração:
Engenharia Naval

Orientador:
Prof. Dr. Toshi-ichi Tachibana

São Paulo
1999

À minha família
À família Zyngier

AGRADECIMENTOS

Ao prof. Dr. Célio Taniguchi.

Ao meu orientador Prof. Dr. Toshi-ichi Tachibana.

Ao IPT S.A. que financiou horas deste trabalho.

Ao prof. Dr. Marcos Tsuzuki, por suas valiosas sugestões bibliográficas.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

AutoCAD	Software gráfico produzido pela Autodesk Inc.
ARX	"Autocad Run-time Extension" - biblioteca de rotinas para programação no ambiente do AutoCAD
CAD	"Computer Aided Design" - projeto auxiliado por computador
DLL	"Dynamic Link Library" - biblioteca de "lincagem" dinâmica
MS-Visual C++	Software de programação na linguagem C++ desenvolvido pela Microsoft Corporation
SDK	"Software Development Kit" - pacote de desenvolvimento de software

Lista de Símbolos

E^3	Espaço Euclidiano tridimensional
R^3	Espaço tridimensional linear dos números reais
a, **b**, ...	Pontos do espaço Euclidiano E^3, ou vetores do espaço tridimensional linear R^3
a, b, c, ...	Números reais ou pontos do espaço Euclidiano unidimensional
Φ	Mapa afim
A, B, ...	Matrizes
E^1	Espaço Euclidiano unidimensional
R	Conjunto dos números reais
B	Polinômio de Bernstein
C^1	Continuidade da primeira derivada
C^2	Continuidade da segunda derivada
Δ(**a**, **b**)	Função que fornece a distância entre os pontos **a** e **b**
k0	Ordem de uma B-spline
n0	Número de pontos de controle de uma B-spline
$N_{i,k0}$	Função base B-spline
l0	Ordem de uma superfície B-spline na direção v
m0	Número de pontos de controle de uma superfície B-spline na direção v
$\underline{u}$	nó a ser inserido num vetor nó existente
X	eixo das abscissas do sistema de referência cartesiano global
Y	eixo das ordenadas do sistema de referência cartesiano global
Z	eixo vertical do sistema de referência cartesiano global
R	Coordenada radial do sistema de referência de coordenadas cilíndricas

θ	Coordenada angular do sistema de referência de coordenadas cilíndricas
ϕ	Ângulo de passo
x^L	Eixo das abscissas do sistema de referência cartesiano local
y^L	Eixo das ordenadas do sistema de referência cartesiano local
BA	Bordo de ataque do perfil de um fólio
$\mathbf{X}_k$	Abscissa global do centro da corda de um fólio
ρ	Caimento longitudinal ("rake")
σ	Caimento lateral ("skew")
BF	Bordo de fuga
P	Passo do propulsor
R_p	Raio do propulsor
θ_p	Coordenada angular de um ponto no sistema de referência de coordenadas cilíndricas
θ_k	Coordenada angular medida no sistema de coordenadas cilíndricas da origem do sistema de coordenadas cartesianas local
f	Função erro a ser minimizada no método dos mínimos quadrados
$\mathbf{X}_{min}$	Menor abscissa do bosso
$\mathbf{Y}_c$	Meia-corda de intersecção com o bosso
$\mathbf{X}_{max}$	Maior abscissa do bosso
$\underline{\mathbf{b}}_j$	Ponto de controle correspondente a uma região suavizada de uma curva B-spline

Resumo

Este trabalho apresenta uma proposta de modelagem matemática para interpolar a superfície da pá de hélices navais utilizando o método B-spline sugerido por PIEGL; TYLER (1997) e por NOWACKI et al. (1995), denominado "skinning". As B-splines empregadas são do tipo não-uniformes e, por isso, a geração da superfície dependerá da uniformização dos vetores nós das curvas iniciais, utilizadas na primeira etapa do processo. Assim, será proposto um método de uniformização dos vetores nós que permitirá grande economia de espaço de memória e conseqüente aumento de desempenho. Finalmente, é feita a implementação computacional do método, utilizando o compilador MS-Visual C++ 5.0 para a criação de uma DLL ("dynamic link library") seguindo o padrão exigido pelas bibliotecas ARX ("Autocad Runtime Extension") do programa AutoCAD R14. Após a modelagem gráfica tri-dimensional, são produzidas seções planas que permitirão a construção de um molde tri-dimensional tipo fêmea que poderá ser usado na confecção das pás em resina termofixa.

Abstract

This work presents a mathematical approach based on the B-Spline method suggested by PIEGL; TYLER (1997) and NOWACKI et al. (1995), named "skinning", to interpolate the surface of a marine screw propeller blade. The B-Spline type employed is non-uniform, so that the generation of the surface will depend on the uniformization of the knot vectors of the initial curves used in the first step of the process. Then, a method of uniformization of the knot vectors is proposed that allows for a lower memory need and greater performance. Finally, the method is implemented in a computer program compiled by MS-Visual C++ 5.0 into a DLL ("dynamic link library") complying with the ARX ("AutoCAD Runtime Extension") library standard used by the AutoCAD R14 software package. After the 3D graphical modeling, planar sections are produced that will allow for the construction of a hollow 3D cast die that may be used to manufacture thermofix resin blades.

1 - INTRODUÇÃO

A complexidade da geometria dos propulsores tipo hélice dificulta a sua elaboração e, em especial, quando se trata de propulsores fora de produção em série, como aqueles utilizados em ensaios em tanque de provas, o custo de aquisição torna-se elevado.

Os modelos reduzidos são necessários quando se pretende obter as características operacionais de uma nova geometria de hélice, ocasião na qual se efetua um ensaio de "água aberta", ou quando se necessita obter os coeficientes de interação casco-hélice, ocasião em que se efetua o ensaio de auto-propulsão com modelo reduzido. Em geral, as escalas empregadas podem variar desde 1:6 até 1:100 ou mais, em função das dimensões do protótipo e das restrições físicas do tanque de provas onde será feito o ensaio. Assim, elevada precisão de fabricação é exigida, pois um erro dimensional no modelo será multiplicado pelo valor da escala.

Neste estudo será proposto um método semi-automático para fabricação de modelos de propulsores em plástico reforçado com fibra de vidro. A ênfase será dada na modelagem matemática para aproximar a geometria de uma pá

utilizando o método B-Spline implementado na forma de programa de computador integrado ao ambiente do AutoCAD®.

No presente estudo será apresentado:

a) a descrição matemática da geometria do hélice;

b) o método matemático B-Spline para aproximação de curvas uniformes e não-uniformes;

c) a implementação de classes C++ para a geometria do hélice da série Kaplan e para a teoria B-Spline;

d) um método de uniformização dos vetores nós de um conjunto de curvas B-Spline;

e) a aplicação do método denominado "skinning" para a geração de uma superfície B-Spline a partir de um conjunto de curvas;

f) a implementação de classes C++ para a geometria do hélice da série Kaplan, para curvas B-Spline e para superfícies B-Spline;

g) a implementação de programa ARX (AutoCAD Runtime eXtension) para utilização das classes do item f no ambiente do AutoCAD R14;

h) um exemplo de modelagem de um propulsor da série Kaplan;

i) a impressão de alguns moldes de fabricação.

2 - MODELAGEM MATEMÁTICA

A geometria de um propulsor tipo hélice de série sistemática é fornecida na forma de perfis tipo asa desenhados sobre superfícies cilíndricas definidas em frações do raio do hélice. Essa forma de apresentação discreta não permite obter, com precisão, coordenadas de perfis definidos em raios intermediários, o que ocasiona grandes dificuldades quando se pretende automatizar o processo de fabricação de uma pá. Neste capítulo, é lançada a base do método que permitirá definir matematicamente qualquer ponto da superfície de uma pá.

2.1 - Definições

As definições a seguir descritas foram propostas por FARIN (1996).

a) Pontos e vetores

Um ponto identifica uma posição no espaço Euclidiano E^3. Vetores são elementos do espaço tri-dimensional linear R^3.

Neste estudo será adotada a mesma notação proposta por FARIN (1996). Assim, pontos e vetores serão representados por letras latinas minúsculas em negrito, e. g., **a**, **b**, etc. E, no caso de se adotar um sistema de referência cartesiano, ambos serão representados por triplas de números reais, ou coordenadas, apresentados na forma de coluna, e. g.:

$$\mathbf{a} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix}^T \tag{2-1}$$

A diferença entre pontos e vetores pode ser mostrada a partir da relação existente entre eles. Dados dois pontos **a** e **b**, existe um único vetor que aponta de **a** para **b**, que pode ser calculado pela subtração das suas coordenadas:

$$\mathbf{a} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix}^T \quad \mathbf{b} = \begin{bmatrix} r \\ s \\ t \end{bmatrix} = \begin{bmatrix} r & s & t \end{bmatrix}^T \tag{2-2}$$

$$\mathbf{v} = \mathbf{b} - \mathbf{a} = \begin{bmatrix} (r-x) & (s-y) & (t-z) \end{bmatrix}^T ; \ \mathbf{a},\mathbf{b} \in E^3, \mathbf{v} \in R^3 \tag{2-3}$$

Por outro lado, dado um vetor **v**, existem infinitos pares de pontos **a** e **b** tais que **v=b-a**. Como prova, basta adicionar aos pontos um vetor arbitrário **w**

para obter dois novos pontos **a**+**w** e **b**+**w**, o vetor porém mantem-se inalterado, pois **v**=(**b**+**w**)-(**a**+**w**)=**b**-**a**.

b) "Combinação baricêntrica". Operação fundamental entre pontos.

Segundo FARIN (1996), "combinação baricêntrica" ou "combinação afim" é definida como:

$$\mathbf{b} = \sum_{j=0}^{n} a_j \mathbf{b}_j; \qquad \mathbf{b}_j \in E^3; \qquad a_0 + \ldots + a_n = 1 \tag{2-4}$$

A expressão (2-4) pode ser reescrita na forma vetorial, obtendo-se:

$$\mathbf{b} = \mathbf{b}_0 + \sum_{j=1}^{n} a_j (\mathbf{b}_j - \mathbf{b}_o) \tag{2-5}$$

c) "Casco convexo" ("convex hull")

Define-se "combinação convexa" como o caso particular de "combinação baricêntrica" onde $a_j > 0 \ \forall \ j$. Quando isso ocorre, o ponto resultante estará sempre contido na poligonal envoltória dos pontos. Assim, o conjunto dos pontos formado por todas as "combinações convexas" possíveis de um conjunto de pontos é definido como "casco convexo".

d) "Mapa afim"

O termo "mapa afim" é devido a Euler, e pode ser asssim definido:

Um mapa afim é um mapa Φ que mapeia E^3 nele mesmo deixando "combinações baricêntricas" invariantes.

O enunciado anterior pode ser escrito matematicamente. Seja:

$$\mathbf{x} = \sum_{j=0}^{n} a_j \mathbf{a}_j;\ \ \mathbf{a}_j \in E^3; \qquad a_0 + \ldots + a_n = 1 \quad , \tag{2-6}$$

uma "combinação baricêntrica". Se Φ é um "mapa afim", então:

$$\Phi\mathbf{x} = \sum_{j=0}^{n} a_j \Phi\mathbf{a}_j;\ \ \Phi\mathbf{x},\ \ \Phi\mathbf{a}_j \in E^3 \tag{2-7}$$

Um "mapa afim" pode ser escrito matematicamente como:

$$\Phi\mathbf{x} = A\mathbf{x} + \mathbf{v} \quad , \tag{2-8}$$

onde A é uma matriz 3x3 e $\mathbf{v}$ é um vetor.

Para mostrar que um "mapa afim" pode ser escrito dessa forma pode-se aplicar a definição:

$$\Phi\sum a_j \mathbf{a}_j = A\left(\sum a_j \mathbf{a}_j\right) + \mathbf{v} \qquad (2\text{-}9)$$

$$= A\left(\sum a_j A\mathbf{a}_j\right) + \sum a_j \mathbf{v}, \text{ pois } \sum a_j = 1$$

$$= \sum a_j (A\mathbf{a}_j + \mathbf{v})$$

$$= \sum a_j \Phi \mathbf{a}_j$$

Alguns exemplos de "mapa afim" são:

- Identidade: **v**=0; A=I; I=matriz identidade
- Translação: **v**≠0; A=I
- Escala: **v**=0; A=matriz diagonal
- Rotação: **v**=0 e

$$A = \begin{bmatrix} \cos a & -\sin a & 0 \\ \sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Cisalhamento: **v**=0 e

$$A = \begin{bmatrix} 1 & a & b \\ 0 & 1 & c \\ 0 & 0 & 1 \end{bmatrix}$$

- Projeção paralela: **v**=0 e

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Todo "mapa afim" complexo pode ser decomposto em translações, rotações, cisalhamentos e escalas.

Um caso particular de "mapa afim" é o mapa Euclidiano, também conhecido por movimento de corpo rígido. Isso ocorre quando $A^TA = I$. Incluem-se nesse caso translações e rotações.

2.2 - Interpolação linear

Considerem-se dois pontos distintos **a** e **b** no espaço E^3 e a "combinação baricêntrica" a seguir:

$$\mathbf{x} = \mathbf{x}(u) = (1-u) * \mathbf{a} + u * \mathbf{b} \qquad (2\text{-}10)$$

O conjunto de todos os pontos $\mathbf{x}(u)$ é chamado de linha reta que passa por **a** e **b**.

Considerando três pontos 0, u, 1 do espaço unidimensional E^1 pode-se observar que a mesma combinação baricêntrica é ainda válida em E^1: O espaço unidimensional E^1 , por definição, é idêntico ao espaço dos números reais R.

$$u = (1-u) * 0 + u * 1 \qquad (2\text{-}11)$$

Logo, u se relaciona com 0 e 1 pela mesma "combinação baricêntrica" que relaciona **x** com **a** e **b**. Assim os três pontos do espaço tri-dimensional **x**, **a** e **b** são um "mapa afim" dos três pontos 0, u e 1 do espaço unidimensional. Portanto, define-se interpolação linear como um "mapa afim" da reta real sobre uma linha no espaço E^3.

Os conceitos que se seguem estão apoiados nessa definição. Com essa idéia será possível construir parábolas do segundo e terceiro grau na forma de curvas Bézier, e, por último, na forma de curvas B-Spline.

2.3 - Curvas Bézier

Será mostrado neste tópico que a generalização do conceito de interpolação linear aplicado a pontos no espaço E^3 pode produzir curvas mais complexas, como parábolas do segundo e terceiro grau.

A construção das curvas de Bézier mostrada a seguir está fundamentada no algoritmo devido a de Casteljau.

Sejam três pontos $\mathbf{a}_0$, $\mathbf{a}_1$ e $\mathbf{a}_2$ do espaço E^3 e u do espaço E^1.

É possível obter dois novos pontos $\mathbf{a}_0^1$ e $\mathbf{a}_1^1$ por interpolação linear, fazendo:

$$\mathbf{a}_0^1(u) = (1-u) * \mathbf{a}_0 + u * \mathbf{a}_1 \qquad (2\text{-}12)$$

$$\mathbf{a}_1^1(u) = (1-u) * \mathbf{a}_1 + u * \mathbf{a}_2 \qquad (2\text{-}13)$$

Um terceiro ponto $\mathbf{a}_0^2$ pode ser obtido pelo mesmo processo a partir dos dois anteriores, fazendo:

$$\mathbf{a}_0^2(u) = (1-u) * \mathbf{a}_0^1 + u * \mathbf{a}_1^1 \qquad (2\text{-}14)$$

Tal procedimento pode ser visualizado na fig. 2.1.

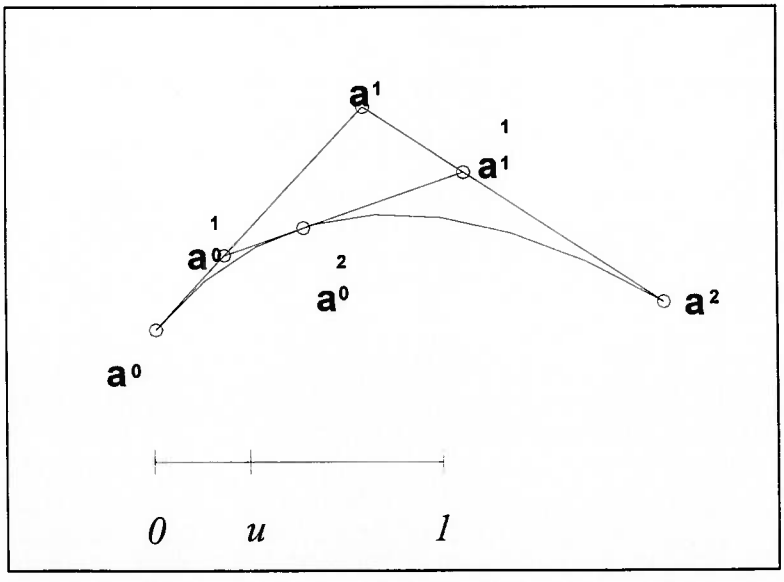

Fig. 2.1 - Interpolação linear repetitiva - teorema de de Casteljau

Substituindo as expressões (2-12) e (2-13) em (2-14), vem:

$$\mathbf{a}_0^2(u) = (1-u)^2 * \mathbf{a}_0 + 2 * u * (1-u) * \mathbf{a}_1 + u^2 * \mathbf{a}_2 \qquad (2\text{-}15)$$

A expressão (2-15) é matematicamente a expressão de uma parábola do segundo grau.

Esse procedimento é definido como interpolação linear repetitiva. Vale observar que o ponto $\mathbf{a}_0^2(u)$ é uma combinação baricêntrica dos pontos $\mathbf{a}_0$, $\mathbf{a}_1$ e $\mathbf{a}_2$, pois:

$$(1-u)^2 + 2 * u * (1-u) + u^2 = 1 \qquad (2\text{-}16)$$

Assim, pode-se afirmar que uma parábola é uma curva plana, pois foi construída a partir de três pontos, portanto, inadequada para representar curvas mais complexas no espaço.

O procedimento acima pode ser generalizado no algoritmo de de Casteljau:

Dados $\mathbf{a}_0,\ldots,\mathbf{a}_n \in E^3$ e $u \in R$, e fazendo:

$$\mathbf{a}_i^r(u) = (1-u) * \mathbf{a}_i^{r-1}(u) + u * \mathbf{a}_{i+1}^{r-1}(u) \begin{cases} r = 1,\ldots,n \\ i = 0,\ldots,n-r \end{cases} \quad (2\text{-}17)$$

e: $$\mathbf{a}_i^0(u) = \mathbf{a}_i \quad (2\text{-}18)$$

Então o ponto $\mathbf{a}_i^0(u)$ é o ponto de parâmetro u na curva de Bézier.

Os pontos $\mathbf{a}_0$, ..., $\mathbf{a}_n$ são chamados de polígono de Bézier ou polígono de controle. Os pontos $\mathbf{a}_0^n$ são os pontos da curva Bézier.

No exemplo a seguir tem-se a aplicação do teorema de de Casteljau para a parábola cúbica, mostrada na fig 2.2

$$\mathbf{a}_0^1(u) = (1-u) * \mathbf{a}_0 + u * \mathbf{a}_1$$
$$\mathbf{a}_1^1(u) = (1-u) * \mathbf{a}_1 + u * \mathbf{a}_2$$

$$\mathbf{a}_2^1(u) = (1-u) * \mathbf{a}_2 + u * \mathbf{a}_3$$

$$\mathbf{a}_0^2(u) = (1-u) * \mathbf{a}_0^1 + u * \mathbf{a}_1^1$$

$$\mathbf{a}_0^2(u) = (1-u) * \mathbf{a}_0^1 + u * \mathbf{a}_1^1$$
$$\mathbf{a}_1^2(u) = (1-u) * \mathbf{a}_1^1 + u * \mathbf{a}_2^1$$

$$\mathbf{a}_0^3(u) = (1-u) * \mathbf{a}_0^2 + u * \mathbf{a}_1^2$$

$$\mathbf{a}_0^3(u) = (1-u) * \left[(1-u) * \mathbf{a}_0^1 + u * \mathbf{a}_1^1\right] + u * \left[(1-u) * \mathbf{a}_1^1 + u * \mathbf{a}_2^1\right]$$
$$\mathbf{a}_0^3(u) = (1-u) * \left[(1-u) * \left((1-u) * \mathbf{a}_0 + u * \mathbf{a}_1\right) + u * \left((1-u) * \mathbf{a}_1 + u * \mathbf{a}_2\right)\right] +$$
$$u * \left[(1-u) * \left((1-u) * \mathbf{a}_1 + u * \mathbf{a}_2\right) + u * \left((1-u) * \mathbf{a}_2 + u * \mathbf{a}_3\right)\right]$$
$$\mathbf{a}_0^3(u) = -u^3 * (\mathbf{a}_1 - 3\mathbf{a}_2 + 3\mathbf{a}_3 - \mathbf{a}_4) + 3u^2 * (\mathbf{a}_1 - 2\mathbf{a}_2 + \mathbf{a}_3) - 3u * \left(\mathbf{a}_1 - \mathbf{a}_2\right) + \mathbf{a}_1$$

(2-19)

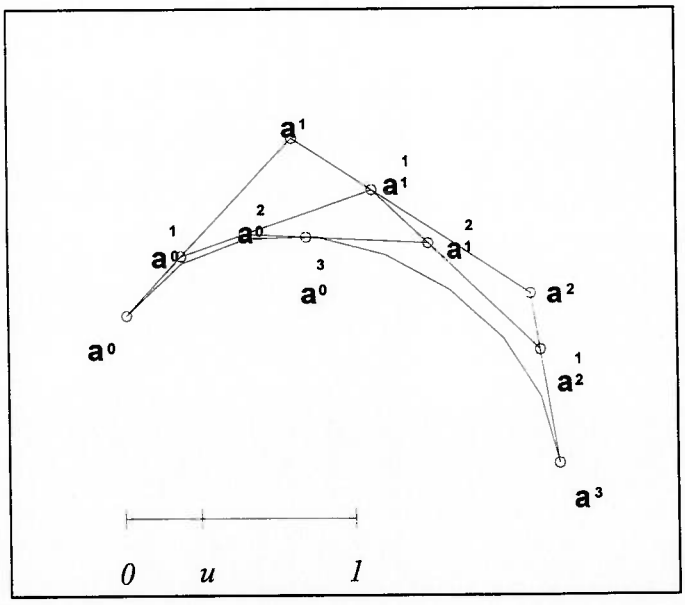

Fig. 2.2 - Interpolação linear repetitiva - curva Bézier

A forma paramétrica, aqui apresentada de representar uma curva, apresenta várias vantagens em relação à representação funcional de uma curva:

a) independe do sistema de referência, pois apresenta a propriedade de invariância a transformações afim;
b) é possível modificar a curva modificando a posição dos pontos de controle, i.e., a modificação da curva pode ser feita por manipulação geométrica direta dos pontos de controle.

2.4 - A curva Bézier expressa em polinômios de Bernstein

Segundo FARIN (1996), é possível escrever uma curva Bézier utilizando os polinômios de Bernstein.

Os polinômios de Bernstein são definidos pela expressão:

$$\boldsymbol{B}_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i} \quad , \tag{2-20}$$

onde:

$$\binom{n}{i} = \begin{cases} \dfrac{n!}{i!(n-i)!} & 0 \le i \le n \\ 0 & caso\ \ contrário \end{cases} \qquad (2\text{-}21)$$

É importante notar algumas de suas propriedades:

a) recursividade:

$$\boldsymbol{B}_i^n(u) = (1-u)\boldsymbol{B}_i^{n-1}(u) + u\boldsymbol{B}_{i-1}^{n-1}(u) \qquad (2\text{-}22)$$

sendo:

$$\boldsymbol{B}_0^0(u) = 1 \quad e \quad \boldsymbol{B}_j^n(u) = 0 \quad para \quad j \notin \{0,\ldots,n\} \qquad (2\text{-}23)$$

A demonstração vem de:

$$\begin{aligned} \boldsymbol{B}_i^n(u) &= \binom{n}{i} u^i (1-u)^{n-i} \\ &= \binom{n-1}{i} u^i (1-u)^{n-i} + \binom{n-1}{i-1} u^i (1-u)^{n-i} \\ &= (1-u)\boldsymbol{B}_i^{n-1}(u) + u\boldsymbol{B}_{i-1}^{n-1}(u) \end{aligned}$$

b) partição da unidade

A somatória dos polinômios de Bernstein forma uma partição da unidade, i.e.,

$$\sum_{j=0}^{n} \boldsymbol{B}_j^n(u) = 1$$

, pois:

$$1 = \left[u + (1-u)\right]^n = \sum_{j=0}^{n} \binom{n}{j} u^j (1-u)^{n-j} = \sum_{j=0}^{n} \boldsymbol{B}_j^n(u)$$

Aplicando a expressão do teorema de de Casteljau é possível expressar os pontos intermediários $\mathbf{a}_i^r$ em função dos polinômios de Bernstein de grau r:

$$\mathbf{a}_i^r(u) = \sum_{j=0}^{n} \mathbf{a}_{i+j} \boldsymbol{B}_j^r(u) \qquad r \in \{0,\ldots,n\},\ \ i \in \{0,\ldots,n-r\} \tag{2-24}$$

Quando $r=n$, o ponto resultante está sobre a curva Bézier, i. e.,

$$\mathbf{a}^n(u) = \mathbf{a}_0^n(u) = \sum_{j=0}^{n} \mathbf{a}_j \boldsymbol{B}_j^n(u) \tag{2-25}$$

2.5 - B-splines

A abordagem anterior permite construir polinômios de grau n a partir de n+1 pontos de controle. No caso de curvas no espaço o grau mínimo seria 3, ou seja, n+1=4, porém é fácil encontrar curvas no espaço que não podem ser

aproximadas por um polinômio de grau 3, e. g., curvas com mais de uma inflexão, ou mais de dois mínimos ou máximos locais. Nesses casos, então, a solução seria adotar curvas representadas por polinômios de grau mais elevado. Porém, há dois grandes inconvenientes:

a) polinômios de grau elevado tendem a se tornar instáveis computacionalmente nos casos em que os valores absolutos dos coeficientes de cada termo tornam-se grandes;
b) a alteração de um ponto de controle produz alterações ao longo de toda a curva, condição que pode ser indesejada quando se buscam pequenos ajustes locais.

A solução encontrada para esse caso foi utilizar segmentos de polinômios de tal forma que fosse garantida a continuidade C^1 e C^2 nos pontos de união de dois segmentos contíguos.

Dentre as várias formas apresentadas pelos diversos autores para apresentar o conceito de B-Spline, aquela proposta por FARIN (1996) é geometricamente mais intuitiva.

Define-se uma curva spline (B-spline) como sendo "um mapa contínuo de uma coleção de intervalos $u_1<...<u_L$ no espaço E^3, onde cada intervalo $[u_i, u_{i+1}]$ é mapeado num segmento polinomial da curva".

A definição acima pode ser melhor visualizada com auxílio da fig. 2.3.

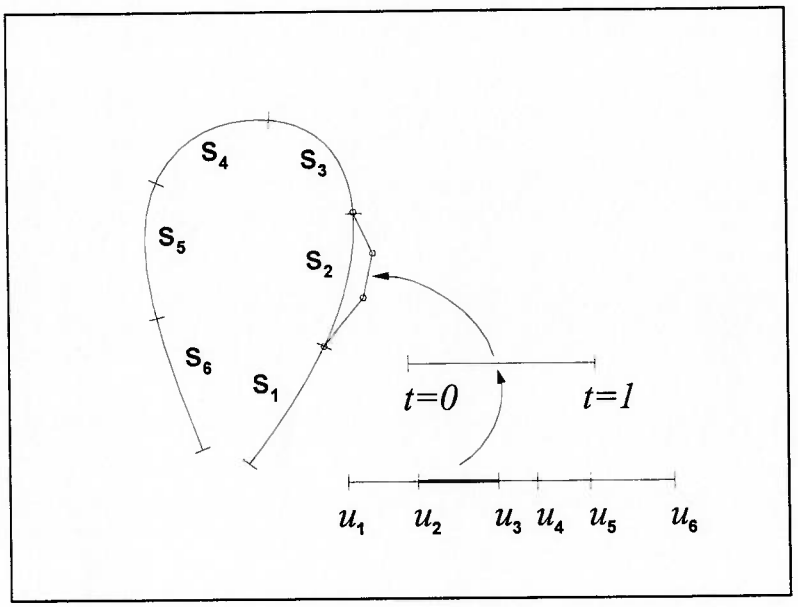

Fig. 2.3 - Geometria de uma curva spline

Cada número real u_i é chamado de "nó" e a coleção dos "nós", $[u_O<...<u_L]$ é chamada de "vetor nó".

Cada segmento de curva $\boldsymbol{s}_i$ é uma curva Bézier.

Dado u contido no intervalo $[u_i, u_{i+1}]$, é possível obter um parâmetro local t, definido por:

$$t = \frac{u - u_i}{u_{i+1} - u_i} \qquad (2\text{-}26)$$

Pode-se observar na expressão acima que t varia entre 0 e 1 conforme u varia entre u_i e u_{i+1}.

Os pontos $\mathbf{s}(u_i)=\mathbf{s}_i(0)=\mathbf{s}_{i-1}(1)$ são chamados de pontos de junção.

Uma curva assim construída seria especialmente útil se fosse possível garantir a continuidade também da primeira e da segunda derivada, i. e., se a mesma pudesse ser C^1 e C^2.

Considerem-se dois segmentos $\mathbf{s}_i$ e $\mathbf{s}_{i+1}$ adjacentes, os respectivos intervalos do vetor nó $[u_{i-1}, u_i]$ e $[u_i, u_{i+1}]$ e os pontos de controle $\mathbf{a}_{n-1}$, $\mathbf{a}_n$ e $\mathbf{a}_{n+1}$, adjacentes à junção representados na fig. 2-4. A continuidade C^1 estará garantida se forem satisfeitas as seguintes condições:

a) se $\mathbf{a}_{n-1}$, $\mathbf{a}_n$ e $\mathbf{a}_{n+1}$ forem colineares;

b) se a proporção entre as distâncias (representada por Δ, na fórmula abaixo) dos nós adjacentes até a junção for igual a proporção da distância entre os nós, i. e.:

$$\frac{\Delta(\mathbf{a}_{n-1},\mathbf{a}_n)}{\Delta(\mathbf{a}_n,\mathbf{a}_{n+1})}=\frac{u_i-u_{i-1}}{u_{i+1}-u_i} \qquad (2\text{-}27)$$

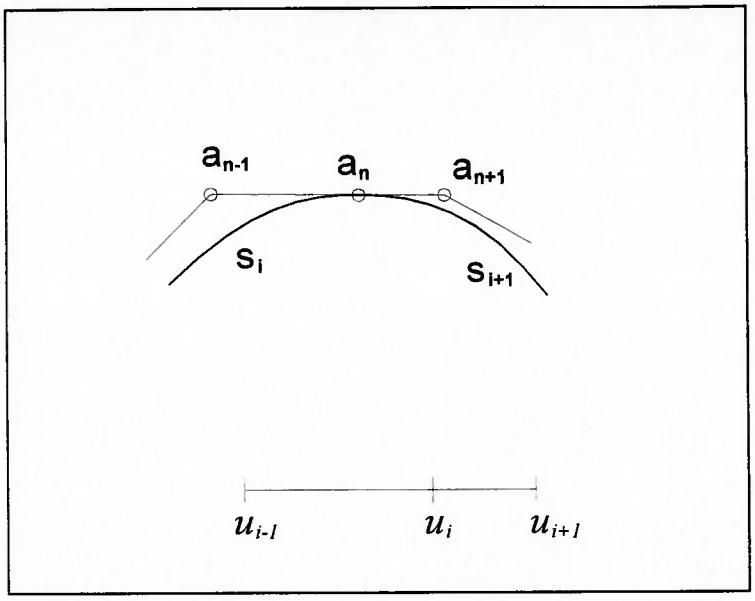

Fig. 2.4 - Continuidade C^1

Considerem-se agora um ponto de controle adicional em cada segmento: $\mathbf{a}_{n-2}$ e $\mathbf{a}_{n+2}$ e um ponto auxiliar **b**, obtido do cruzamento das retas definidas por ($\mathbf{a}_{n-2}$, $\mathbf{a}_{n-1}$) e por ($\mathbf{a}_{n+1}$, $\mathbf{a}_{n+2}$), conforme representado na fig. 2.5. A continuidade C^2 estará garantida se:

$$\frac{\Delta(\mathbf{a}_{n-2}, \mathbf{a}_{n-1})}{\Delta(\mathbf{a}_{n-1}, \mathbf{b})} = \frac{\Delta(\mathbf{b}, \mathbf{a}_{n+1})}{\Delta(\mathbf{a}_{n+1}, \mathbf{a}_{n+2})} \qquad (2\text{-}28)$$

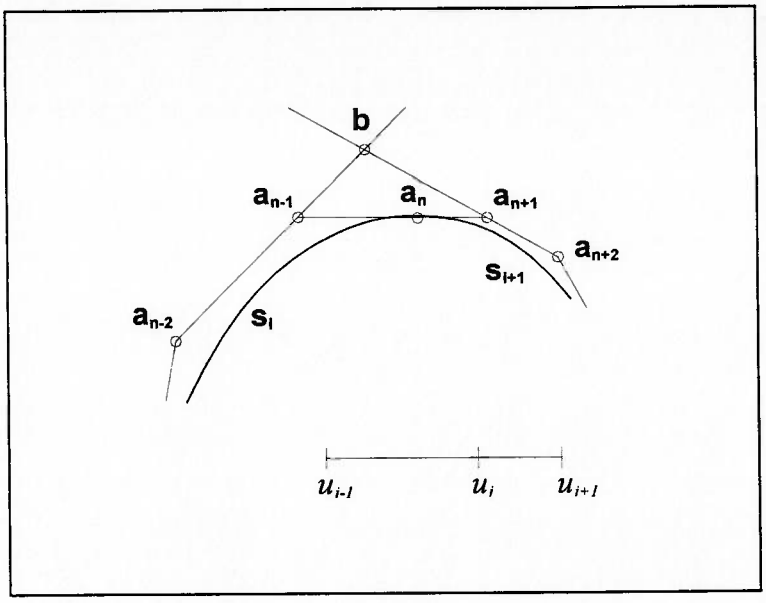

Fig. 2.5 - Continuidade C^2

O conjunto dos pontos auxiliares $\mathbf{b}_i$, definidos em cada ponto de junção e os pontos iniciais e finais da curva como um todo definem o polígono dos pontos de controle da curva B-Spline. Assim sendo, uma curva B-spline estará perfeitamente definida se forem fornecidos o polígono dos pontos de controle e um vetor nó.

A abordagem apresentada acima facilita o entendimento teórico do problema graças ao forte apelo geométrico, porém, computacionalmente torna-se necessária outra abordagem.

Matematicamente uma curva B-spline pode ser representada pela seguinte expressão:

$$\mathbf{s}(u) = \sum_{i=1}^{n0} N_{i,k0}(u).\mathbf{b}_i \quad , \tag{2-29}$$

onde:

$\mathbf{b}_i$ são os pontos de controle;

$k0$ = ordem = grau dos polinômios +1;

$n0$ é o número de pontos de controle;

$N_{i,k0}$ é a função base B-spline

As funções base $N_{i,k0}$ podem ser obtidas de várias formas, porém, será adotada aqui a fórmula recursiva proposta por COX; DE BOOR; MANSFIELD apud FARIN (1996).

$$N_{i,1}(u) = \begin{Bmatrix} 1 & para \quad u \in [u_i, u_{i+1}] \\ 0 & , \ caso \quad contrario \end{Bmatrix} \tag{2-30}$$

$$N_{i,j}(u) = \frac{u - u_i}{u_{i+j-1} - u_i} N_{i,j-1}(u) + \frac{u_{i+j} - u}{u_{i+j} - u_{i+1}} N_{i+1,j-1}(u),$$
$$j \in [1, k0]; \quad i \in [0, n0] \tag{2-31}$$

Na formulação acima adota-se, por razões numéricas, que 0/0=0.

O vetor nó pode ser construído fazendo:

$$U = \left\{ \underbrace{a,\ldots,a}_{k0}, \underbrace{u_k,\ldots,u_{k+n0-k0-1}}_{n0-k0}, \underbrace{b,\ldots,b}_{k0} \right\} \tag{2-32}$$

2.6 - Superfície B-spline

Uma superfície B-Spline pode ser obtida a partir da equação (2-33).

$$\mathbf{s}(u,v) = \sum_{j=1}^{m0}\sum_{i=1}^{n0} N_{i,k0}(u) N_{j,l0}(v) \mathbf{b}_{i,j} \qquad (2\text{-}33)$$

onde :

u: parâmetro

v : parâmetro na direção ortogonal a u

$k0$: ordem na direção u

$l0$: ordem (=grau+1) na direção v

$n0$: número de pontos de controle na direção u

$m0$: número de pontos de controle na direção v

$N_{i,k0}$ e $N_{j,l0}$ são bases B-spline

$\mathbf{b}_{i,j}$: pontos de controle

PIEGL; TYLER (1997) e NOWACKI et al. (1995) mostram que é possível obter pontos de controle de uma superfície a partir dos pontos de controle de algumas curvas interpoladas da superfície, num processo denominado de "skinning", desde que sejam satisfeitas as seguintes condições:

a) todas as curvas possuem a mesma parametrização baseada num mesmo vetor nó;

b) todas as curvas são do mesmo grau.

Tais condições são satisfeitas quando quando é feita a aproximação individual dos perfis de uma pá, desde que se imponha um número fixo de pontos de controle e o mesmo vetor nó para cada perfil. É aqui que reside a maior dificuldade operacional do método.

3 - ALGORITMOS FUNDAMENTAIS

Conhecida a base matemática do metódo, torna-se necessário introduzir alguns algoritmos fundamentais que permitirão aplicá-lo, os quais serão mostrados nos tópicos que se seguem.

3.1 - Construção do vetor nó uniforme

A forma mais simples de construir um vetor nó a partir da expressão 2-32 é fazer:

a=0

u_1=1

$$u_k = u_{k-1}+1 \quad \text{para} \quad k0+1 \leq k \leq n0-k0 \tag{3-1}$$

$$b = n0+k0-1 \tag{3-2}$$

Para exemplificar, sejam $k0$=4 e $n0$=10, assim:

U = {0,0,0,0,1,2,3,4,5,6,6,6,6,6}

Logo, um vetor nó qualquer será uniforme quando:

$$u_{k+1}-u_k=\text{constante para } k0+1 \leq k \leq n0 \qquad (3\text{-}3)$$

3.2 - Construção do vetor nó não-uniforme

Um vetor nó não-uniforme pode ser qualquer seqüência crescente de números reais obedecendo a expressão 2-32. Porém, PIEGL; TYLER (1997) e FARIN (1996) sugerem que se adote a construção de um vetor que considere a distribuição espacial dos pontos pelos quais se pretende passar a curva. Como a quantidade de pontos fornecidos e a quantidade de pontos de controle não estão diretamente relacionadas, PIEGL; TYLER (1997) sugerem o método da média local ("averaging") descrito abaixo.

Dados:

$\mathbf{p}_k$, k=1,..., np pontos no espaço pelas quais se pretende aproximar ou interpolar uma curva B-spline

$n0$ - número de pontos de controle

$k0$ - ordem

Com base na expressão 2-32 , o vetor nó pode ser construído fazendo:

a=0;

$$u_i = \frac{\sum_{j=i-k0+1}^{i-1} \left(\frac{\sum_{k=1}^{j} |p_{k+1} - p_k|}{\sum_{k=1}^{np-1} |p_{k+1} - p_k|} \right)}{k0-1}, \quad k0+1 \le i \le n0 \qquad (3\text{-}4)$$

b=1

3.3 - Cálculo das funções base não-nulas

O cálculo das funções de base não-nulas pode ser feito a partir das expressões 2-30 e 2-31 e empregando-se o algoritmo ALG 3-1 proposto por PIEGL; TYLER (1997), que apresenta a qualidade de evitar divisões por zero:

```
bsp(double N[][], double v[],  int k, int ip, double u)
{
 int l, j;
 double M[], left[], right[], saved, temp;
 M[0]=1;
 for (j=1; j<=k-1; j++)
 {
            left[j]=u-v[ip+1-j];
            right[j]=v[ip+j]-u;
            saved=0;
            for(i=0; i<j; i++)
            {
                        temp=M[i]/(right[i+1]+left[j-i]);
                        M[i]=saved+right[i+1]*temp;
                        saved=left[j-i]*temp;
            }
            M[j]=saved;
 }
 for(j=1; j<=k; j++)N[j][k]=M[j-1];
}
```

(ALG 3-1)

3.4 - Cálculo de um ponto na curva B-Spline

O cálculo de um ponto na curva B-spline, para um dado valor do parâmetro u, pode ser feito aplicando-se diretamente a expressão 2-29. Deve-se observar, porém, que há somente $k0$ bases não nulas no segmento ip do vetor nó que contém u, logo a expressão 2-29 pode ser reescrita:

$$\mathbf{s}(u) = \sum_{i=ip-k0+1}^{ip} N_{i,k0}(u).\mathbf{b}_i \qquad (3\text{-}5)$$

Assim o algoritmo, após a localização do segmento ip, é:

```
{
 p=k0+1;
 while (u> knots[ip]) ip++;
 ip--;
 bsp(N,knots,k0,ip,u);
 sum=0;
 for (i=1 ;i<=k0; i++) sum=sum+b[ip-k0+i]*N[i][k0];
 return sum;
}
```

(ALG 3-2)

3.5 - Inserção de um nó

Considerando uma curva B-spline de ordem $k0$, definida pelos pontos de controle $\mathbf{b}_i$, i=1,..., $n0$ e pelo vetor nó U=u_k, k=1,...,$n0$+$k0$, a inserção de um nó consiste em acrescentar um número real $\underline{u}$ no vetor nó e obter um novo

conjunto de pontos de controle c_i, i=1,..., $n0$+1 tal que a curva permaneça inalterada, i. e.:

$$\sum_{i=1}^{n0} N_{i,k0}(u)\mathbf{b}_i = \sum_{i=1}^{n0+1} N_{i,k0}(u)\mathbf{c}_i \tag{3-6}$$

PIEGL; TYLER (1997) propoem uma solução para o sistema de equações lineares acima que consiste em:

$$\begin{aligned} &\mathbf{c}_i = \mathbf{b}_i, \qquad 1 \le i \le ip - k0 + 1 \\ &\mathbf{c}_i = w_i \mathbf{b}_i + (1 - w_i)\mathbf{b}_{i-1} \qquad ip - k0 + 2 \le i \le ip \\ &\mathbf{c}_{i+1} = \mathbf{b}_i \qquad ip + 1 \le i \le n0 \end{aligned} \tag{3-7}$$

$$w_i = \frac{u - u_i}{u_{i+k0-1} - u_i} \tag{3-8}$$

3.6 - Remoção de um nó

Considerando uma curva B-spline de ordem $k0$, definida pelos pontos de controle $\mathbf{b}_i$, i=1,..., $n0$ e pelo vetor nó U=u_k, k=1,...,$n0$+$k0$, a remoção de um nó consiste em retirar o nó u_r do vetor nó e obter um novo conjunto de pontos de controle $\mathbf{c}_i$, i=1,..., $n0$-1 tal que a curva permaneça inalterada, i. e.:

$$\sum_{i=1}^{n0} N_{i,k0}(u)\mathbf{b}_i = \sum_{i=1}^{n0-1} N_{i,k0}(u)\mathbf{c}_i \tag{3-9}$$

O sistema de equações acima nem sempre apresenta solução, pois é possível imaginar situações em que um nó não possa ser removido. PIEGL; TYLER (1997) propoem um algoritmo de solução, que aqui foi simplificado para B-splines de ordem 4, conforme abaixo:

$$i=r-k0+1 \tag{3-10}$$

$$j=r-1 \tag{3-11}$$

$$w_r = \frac{u_r - u_{r-k0+2}}{u_{r+2} - u_{r-k0+2}} \tag{3-12}$$

$$w_j = \frac{u_r - u_j}{u_{j+k0} - u_j} \tag{3-13}$$

$$\mathbf{c}_i = \frac{\mathbf{b}_i - (1-w_i)\mathbf{c}_{i-1}}{w_i} \tag{3-14}$$

$$\mathbf{c}_j = \frac{\mathbf{b}_j - w_j\mathbf{c}_{i+1}}{(1-w_j)} \tag{3-15}$$

O nó poderá ser removido se:

$$\left\| \mathbf{b}_{r-k0+2} - \left[w_r \mathbf{c}_j + (1 - w_r) \mathbf{c}_i \right] \right\| \leq TOL \quad , \tag{3-16}$$

onde TOL é a precisão desejada.

4 - GEOMETRIA DO PROPULSOR

Tradicionalmente a geometria de um hélice é representada através de um desenho de cinco vistas semelhante ao mostrado na fig. 4.1. Porém, para descrever matematicamente a superfície da pá de um propulsor tipo hélice no espaço tri-dimensional é necessário definir um sistema de coordenadas que facilite o tratamento matemático do problema. Devido à complexidade da superfície de uma pá, são utilizados três sistemas de referência, um cartesiano global, um cilíndrico e outro cartesiano local, inspirando-se na proposta de KLEIN (1975), com algumas modificações, sendo a principal delas, a substituição dos eixos **Y** e **Z,** um pelo outro, no sistema de referência cartesiano global a fim de facilitar a integração com o aplicativo CAD.

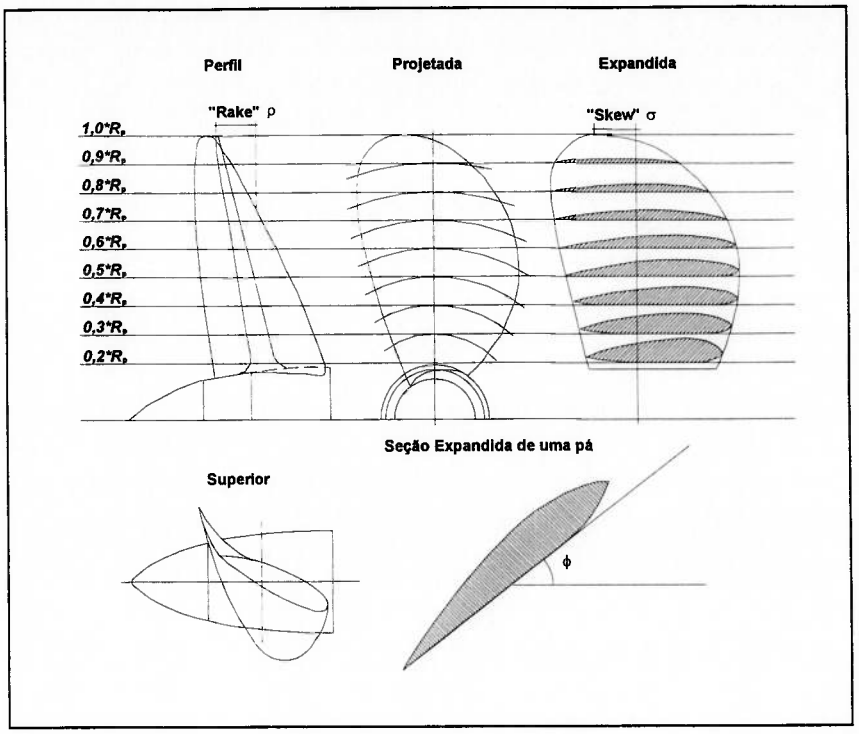

Fig. 4.1 - Representação tradicional da geometria de um hélice

4.1 - Sistemas de referência

São assumidas as seguintes hipóteses e convenções para definir os sistemas de referência:

a) o observador está posicionado a ré do hélice olhando para a proa do navio alinhado com o eixo de rotação do hélice;

b) a pá em estudo está na posição vertical, de forma que seu eixo de passo coincide com a direção vertical;

c) se o hélice for direito, o bordo de ataque estará à direita (boreste) do observador e o sentido de rotação será horário;

4.1.1 - Sistema de referência cartesiano global

É composto por três eixos tri-ortogonais assim definidos:

a) o eixo **X** coincide com eixo de rotação do hélice, sendo o sentido positivo de ré para vante;

b) o eixo **Z** coincide com o eixo de passo, sendo o sentido positivo de baixo para cima;

c) o eixo **Y** resulta do produto vetorial **Z** x **X**;

d) o ponto de encontro dos três eixos é a coordenada (0,0,0)

O sistema de referência cartesiano global pode ser melhor visualizado com auxílio da fig. 4.2.

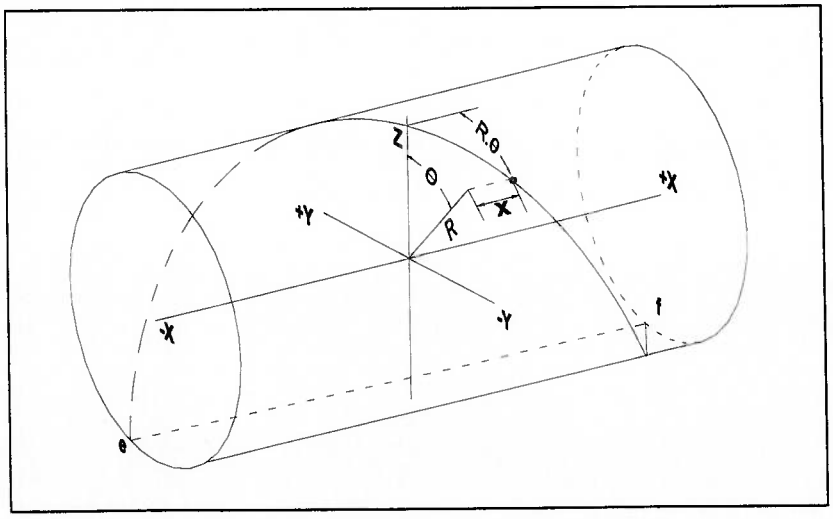

Fig. 4.2 - Sistema de referência cartesiano global

4.1.2 - Sistema de referência de coordenadas cilíndricas

Em geral, a geometria de um propulsor é definida a partir de um número finito de perfis tipo asa definidos sobre uma superfície cilíndrica que intercepta a pá num determinado raio. Para poder relacionar as coordenadas de uma pá definidas nessa superfície com as coordenadas cartesianas utiliza-se o sistema de referência definido na fig. 4.3, que pode ser obtido "cortando-se" a superfície cilíndrica da fig. 4.2 entre os pontos **e** e **f** e tornando-a plana.

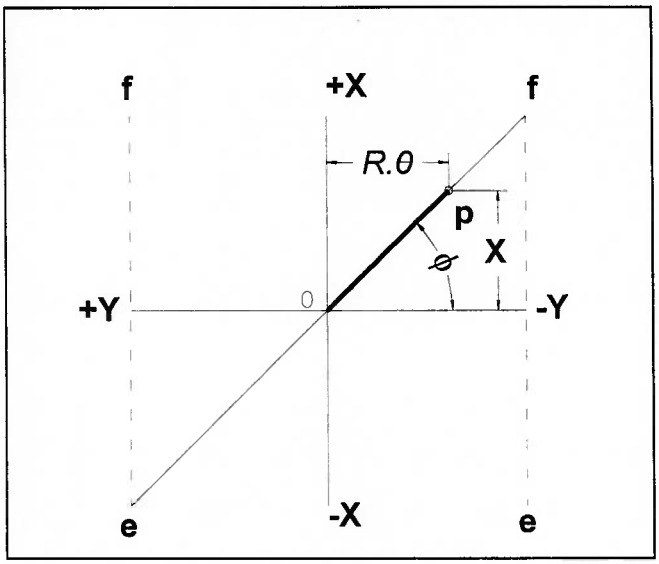

Fig. 4.3 - Sistema de referência de coordenadas cilíndricas

4.1.3 - Sistema de referência cartesiano local

O sistema de referência cartesiano local, mostrado na fig. 4.4, está definido sobre o plano expandido a partir de um eixo x^L paralelo à linha de passo e tangente à face da pá. A origem $\mathbf{X}_k$ está na metade da distância entre as projeções das extremidades dos bordos de ataque (BA) e de fuga (BF) sobre o eixo x^L. O eixo y^L é perpendicular ao eixo x^L de forma que o produto vetorial x^L x y^L tem a mesma direção e sentido do eixo **Z** global.

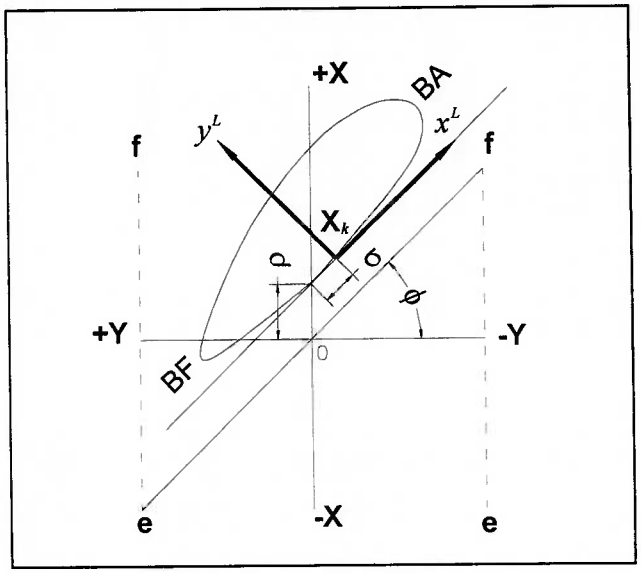

Figura 4.4 - Sistema de referência cartesiano local

4.2 - Equações de conversão de coordenadas

As equações das coordenadas espaciais de um ponto no perfil ($\mathbf{X}_p$, $\mathbf{Y}_p$, $\mathbf{Z}_p$) podem, então, ser deduzidas a partir das coordenadas nos outros sistemas de referência, conforme mostrado nas expressões 4-1 a 4-7.

A abcissa pode ser obtida diretamente da expressão 4-1:

$$\mathbf{X}_p = \mathbf{X}_k + x^L \sin\phi + y^L \cos\phi \qquad (4\text{-}1)$$

onde:

$\mathbf{X}_k$: posição do centro da corda

ϕ: ângulo de passo

A posição do centro da corda pode ser calculada a partir do caimento lateral ("skew") σ, do caimento longitudinal ("rake") ρ e do ângulo de passo ϕ:

$$\mathbf{X}_k = -(\sigma \sin\phi + \rho) \qquad (4\text{-}2)$$

Na fórmula acima convenciona-se que o caimento lateral é positivo no sentido anti-horário, e que o caimento longitudinal é positivo à ré.

O ângulo de passo ϕ pode ser obtido a partir do passo (P) e do raio (R_p):

$$\tan\phi = \frac{P}{2\pi.R_p} \tag{4-3}$$

$$\mathbf{Y}_p = -R_p \sin\theta_p \tag{4-4}$$

$$zp = R\cos\theta_p \tag{4-5}$$

$$\theta_p = \theta_k + \frac{x^L \cos\phi}{R_p} - \frac{y^L \sin\phi}{R_p} \tag{4-6}$$

$$\theta_k = -\frac{\sigma\cos\phi}{R_p} \tag{4-7}$$

5 - APLICAÇÃO DO MÉTODO B-SPLINE PARA A DESCRIÇÃO MATEMÁTICA DA PÁ DO HÉLICE

A aplicação do método B-spline para descrever a superfície da pá do hélice permitirá obter uma expressão matemática, a partir da qual, qualquer ponto sobre a superfície poderá ser calculado com exatidão. Desta forma, a fabricação da pá, definida por essa superfície, será tão precisa quanto o processo de fabricação permitir. O método aqui empregado, sugerido por PIEGL; TYLER (1997) e NOWACKI et al. (1995), recebe o nome de "skinning" e consiste das seguintes etapas:

a) ajustar (interpolar ou aproximar) individualmente algumas curvas da superfície - neste caso, serão interpolados por B-splines não-uniformes os nove perfis cilíndricos da série sistemática Kaplan;
b) uniformizar os vetores nós de todas as curvas, de forma que contenham exatamente os mesmos nós - isso é necessário porque optou-se por empregar B-splines não-uniformes - o método de uniformização dos vetores nós é uma proposta original deste trabalho;

c) interpolar B-splines não-uniformes passando pelos pontos de controle correspondentes de cada curva criados na etapa anterior;

d) uniformizar os vetores nós das B-splines obtidas na etapa c) utilizando o mesmo processo utilizado em b);

5.1 - Ajuste de um perfil

Intuitivamente o primeiro impulso que surge ao se tentar ajustar (interpolar ou aproximar) um perfil de uma pá é fazê-lo diretamente no plano expandido, onde o número de graus de liberdade é menor. Porém, neste caso, todos os pontos de controle, por construção, estarão contidos na mesma superfície cilíndrica do próprio perfil quando forem colocados no espaço. Assim sendo, o "casco convexo" dos pontos de controle resultante será um conjunto de pontos internos à superfície cilíndrica. Como uma curva B-spline não passa por todos os pontos de controle, a curva resultante estará fora da superfície cilíndrica para a qual a mesma havia sido construída no plano expandido. Portanto, o ajuste deverá ser feito diretamente no espaço, de forma que pontos de controle possam ser definidos em superfícies cilíndricas com raios distintos do raio da superfície cilíndrica que contém o perfil sendo ajustado.

A dificuldade seguinte surge ao se tentar escolher o tipo de ajuste : interpolação ou aproximação, e o tipo de B-spline: uniforme ou não-uniforme.

Ocorre aproximação quando o número de pontos de controle é menor que o número de pontos a serem ajustados. Neste caso, o problema não tem solução exata, pois o número de equações do sistema representado pela expressão (5-1) é maior que o número de incógnitas.

ROGERS; ADAMS (1990) e PIEGL; TYLER (1997) sugerem que uma das formas de resolver o problema é empregar o método dos mínimos quadrados, onde as variáveis são os pontos de controle desejados.

$$\mathbf{s}(u) = \sum_{i=1}^{n0} N_{i,k0}(u)\mathbf{b}_i \qquad (5\text{-}1)$$

onde:

$\mathbf{s}(u)$ - ponto sobre a curva em função do parâmetro u

$n0$: quantidade de pontos de controle;

$N_{i,k0}$: funções-base B-Spline normalizadas;

$k0$: ordem (= grau +1);

$\boldsymbol{b}_i$: pontos de controle.

Considerem-se np pontos $\mathbf{d}_j$ a serem aproximados pela curva. Cada ponto possui um parâmetro u_j que pode ser calculado como sendo uma fração do comprimento total da poligonal formada pelos pontos $\{\mathbf{d}_1,...,\mathbf{d}_{np}\}$. Aplicando-

se a equação da curva acima, pode-se obter $\mathbf{s}(u_j)$ que é a aproximação do ponto $\mathbf{d}_j$:

$$\mathbf{s}(u_j) = N_{1,k0}(u_j)\mathbf{b}_1 + N_{2,k0}(u_j)\mathbf{b}_2 + \ldots + N_{n0,k0}(u_j)\mathbf{b}_{n0} \tag{5-2}$$

Pelo método dos mínimos quadrados deseja-se minimizar a função f definida como:

$$f = \sum_{j=1}^{np} \left| \mathbf{d}_j - \mathbf{s}(u_j) \right|^2 = \sum_{j=1}^{np} \left[\mathbf{d}_j \mathbf{d}_j - 2\mathbf{d}_j \sum_{i=1}^{n0} N_{i,k0}(u_j)\mathbf{b}_i + \left(\sum_{i=1}^{n0} N_{i,k0}(u_j)\mathbf{b}_i \right)^2 \right] \tag{5-3}$$

A função f é uma função do segundo grau em relação as variáveis do problema, que são os pontos de controle $\mathbf{b}_i$. Os coeficientes do termo do segundo grau são todos positivos, porque são bases B-spline, logo, a função possui apenas um mínimo. Para obtê-lo basta tomar as derivadas parciais em relação a cada um dos pontos de controle e igualá-las a zero. Desta forma, obtém-se np equações semelhantes a indicada a seguir, para um dado $\mathbf{b}_l$:

$$\frac{\partial f}{\partial \mathbf{b}_l} = \sum_{j=1}^{np} \left[-2N_{l,k0}(u_j)\mathbf{d}_j + 2N_{l,k0}(u_j)\mathbf{d}_j \sum_{i=1}^{n0} N_{i,k0}(u_j)\mathbf{b}_i \right] = 0 \tag{5-4}$$

Reescrevendo, vem:

$$\sum_{j=1}^{np}\sum_{i=1}^{n0} N_{1,k0}(u_j)N_{i,k0}(u_j)\mathbf{b}_i = \sum_{j=1}^{np} N_{1,k0}(u_j)\mathbf{d}_j \qquad (5\text{-}5)$$

Definindo uma matriz N como:

$$N = \begin{bmatrix} N_{1,k0}(u_1) & N_{2,k0}(u_1) & \cdots & N_{n0,k0}(u_1) \\ N_{1,k0}(u_2) & N_{2,k0}(u_2) & \cdots & N_{n0,k0}(u_2) \\ \cdots & \cdots & \cdots & \cdots \\ N_{1,k0}(u_{np}) & N_{2,k0}(u_{np}) & \cdots & N_{n0,k0}(u_{np}) \end{bmatrix} \qquad (5\text{-}6)$$

O sistema de equações representado pela equação 5-5 pode ser escrito na forma matricial como:

$$N^T NB = N^T D \qquad (5\text{-}7)$$

A matriz N^TN é quadrada e DE BOOR apud PIEGL; TYLER (1997) demonstra que é inversível. Assim:

$$B = \left[N^T N\right]^{-1} N^T D \qquad (5\text{-}8)$$

Ocorre interpolação quando o número de pontos de controle é igual ao número de pontos a serem ajustados. Neste caso, o número de equações é

igual ao número de incógnitas e a matriz N é quadrada. Logo, a solução pode ser obtida fazendo:

$$B = N^{-1} D \tag{5-9}$$

O método de escolha foi comparar entre si os resultados obtidos com cada uma das opções. Foram efetuados testes no espaço e no plano expandido obtendo-se, nos dois casos, resultados semelhantes, os quais podem ser visualizados com auxílio das fig. 5.1 a 5.5, que mostram o ajuste do perfil correspondente a $0{,}4R_p$ no plano expandido. Pode-se verificar que a melhor solução é aquela obtida a partir da interpolação por B-splines não-uniformes.

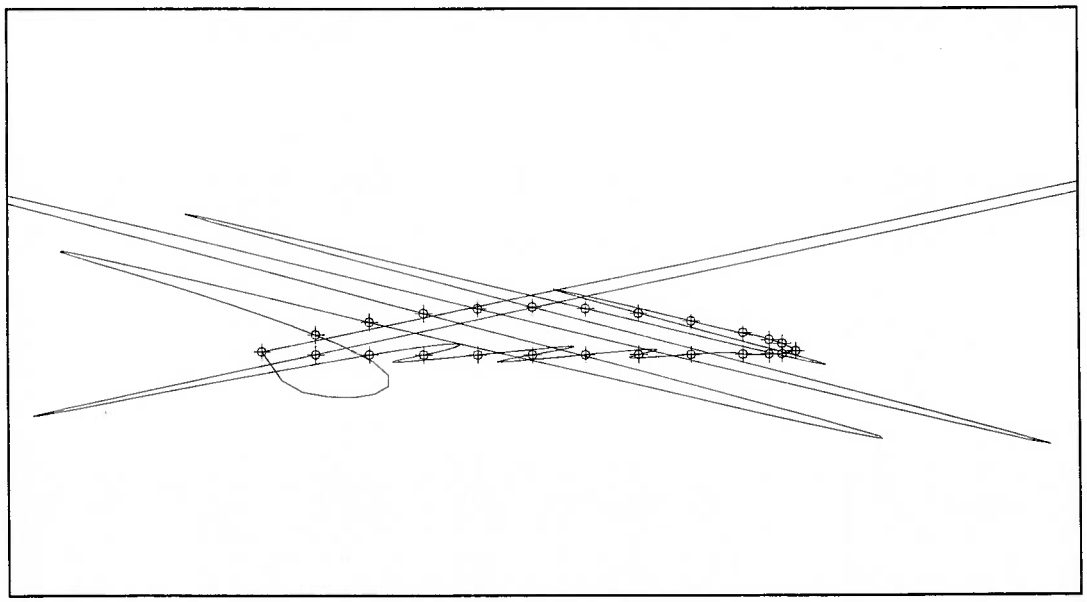

Fig. 5.1 - Interpolação por B-spline uniforme

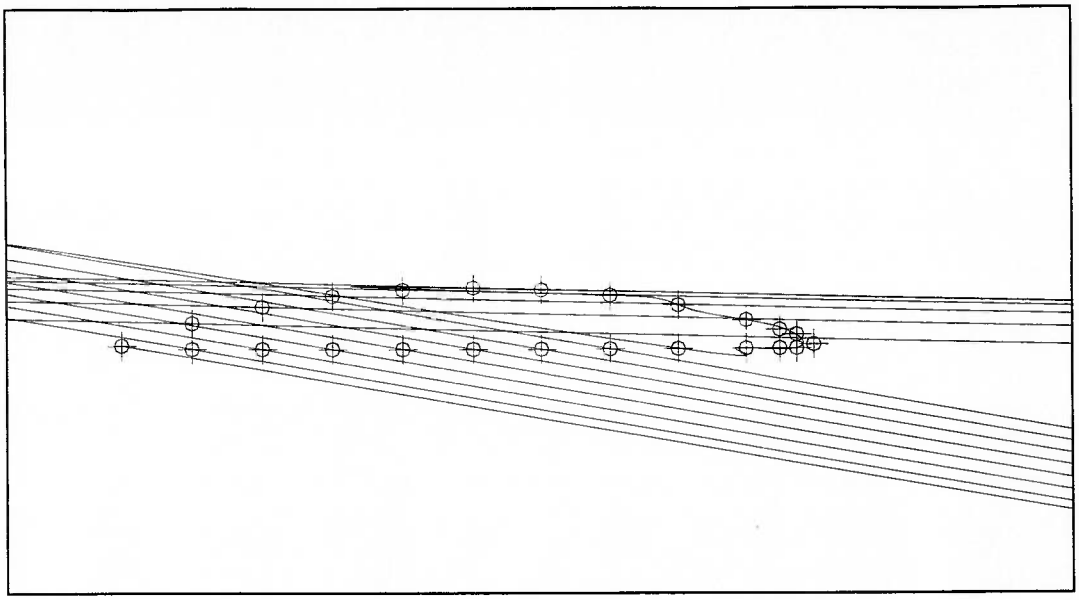

Fig. 5.2 - Aproximação por B-spline uniforme $n0$=24

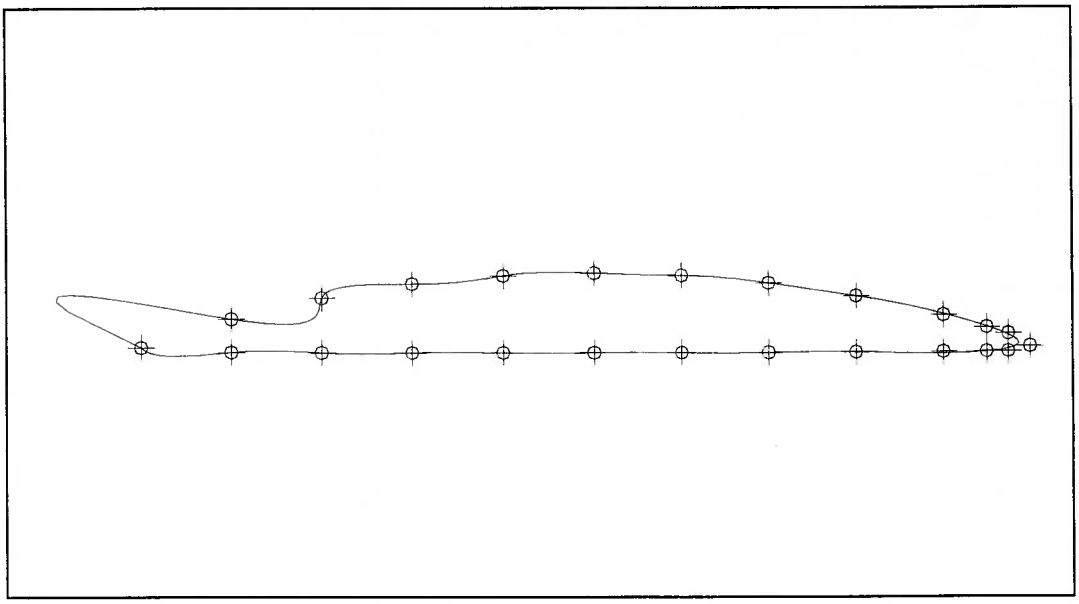

Fig. 5.3 - Aproximação por B-spline uniforme $n0$=21

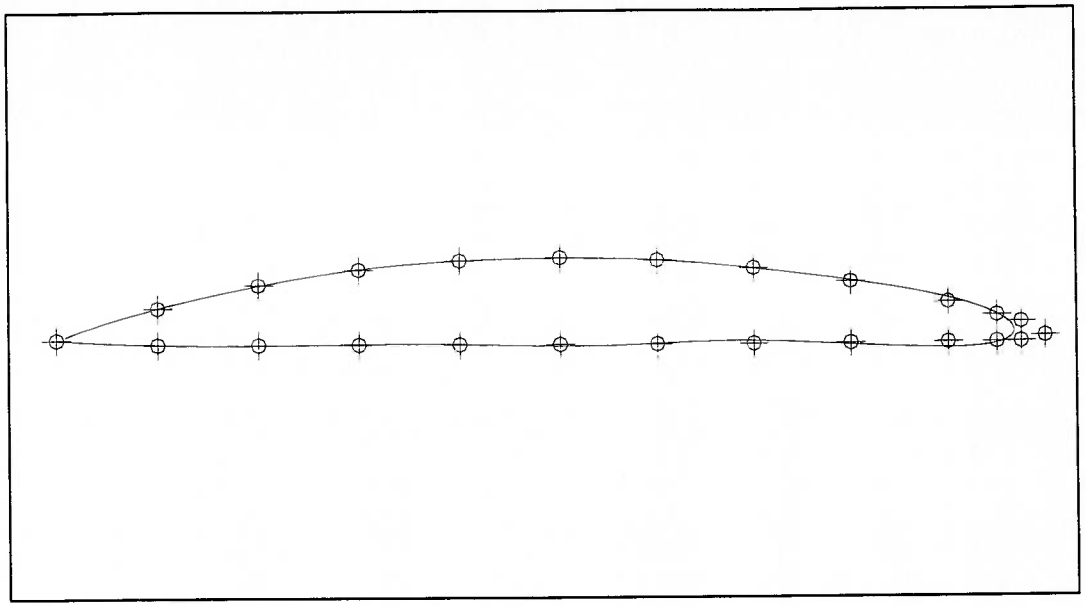

Fig. 5.4 - Aproximação por B-spline uniforme $n0$=14

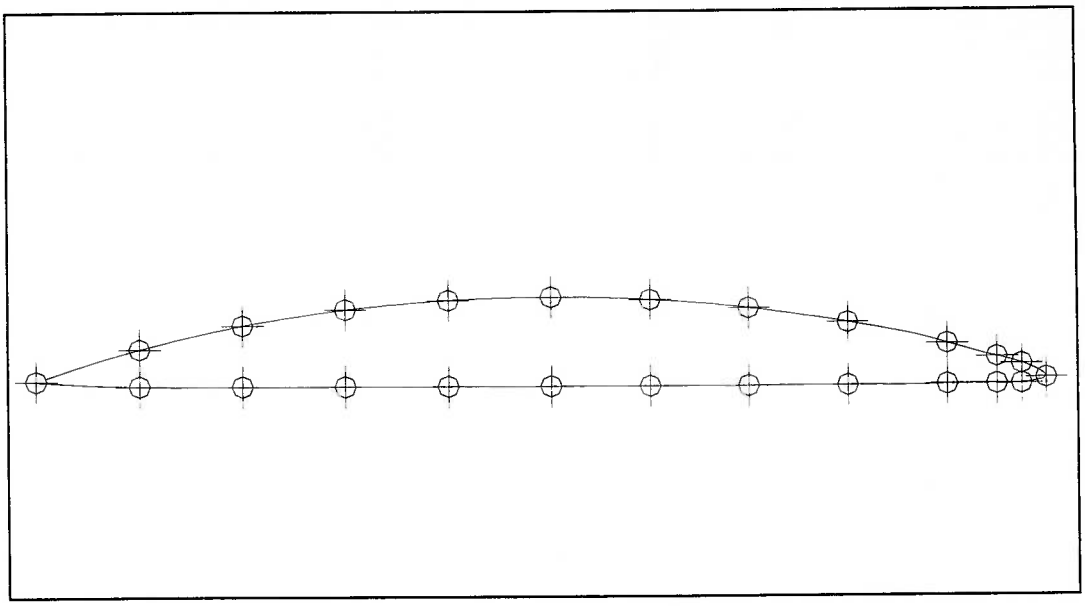

Fig. 5.5 - Interpolação por B-spline não-uniforme

A interpolação por B-spline uniforme mostrou-se instável (fig. 5.1), assim como as aproximações com grande número de pontos de controle (fig. 5.2 e fig. 5.3). Com 14 pontos de controle pode-se observar que a aproximação

deixa de ser instável (fig. 5.4), porém a curva se distancia dos pontos fornecidos, em especial, no bordo de ataque.

O resultado obtido com a interpolação por B-spline não-uniforme (fig. 5.5) produz uma curva suave passando por todos os pontos, o que torna desnecessário o uso de aproximação por esse tipo de B-spline, pois elimina-se a necessidade de manipulação adicional dos pontos de controle para fazer a curva passar pelos pontos fornecidos.

5.2 - Uniformização dos vetores nós

A forma mais simples de uniformizar os vetores nós de todas as curvas seria inserir em cada uma delas todos os nós interiores, i. e., u_k, $k0+1 \leq k \leq n0$,. Tal processo criaria vetores contendo pontos de controle em quantidade dada pela expressão:

$$n*(n0-k0)+2*k0, \qquad (5\text{-}10)$$

neste caso 9*(25-4)+2*4 = 197 nós.

E, portanto, 193 pontos de controle em cada curva. Tal alternativa exigiria espaços de memória grandes e o processo de interpolação dos pontos de controle da etapa seguinte seria lento, pois haveria 193 sistemas de equações para serem resolvidos. Assim, optou-se pela seguinte alternativa:

a) inserir n=25 nós igualmente espaçados, i. e. $u_k = k/(n+1)$;

b) remover todos os nós existentes anteriormente, que puderem ser removidos;

c) incluir nos vetores nós de cada curva todos os nós das demais que não puderam ser removidos;

Tal processo mostrou-se eficiente resultando curvas com 104 pontos de controle, ou seja, uma economia de quase 90 curvas.

5.3 - Interpolação dos pontos de controle

A interpolação dos pontos de controle foi feita empregando-se 104 B-splines não-uniformes cada uma com inicialmente 9 pontos de controle. A interpolação é feita pelos pontos de controle correspondentes obtidos um de cada B-spline que interpolou cada perfil de raio constante. A seguir, inserem-se 9 nós adicionais igualmente espaçados de 0.1. Logo, removem-se todos os nós inicialmente gerados no processo de interpolação, sempre que possível. Finalmente inserem-se em todas as demais B-splines os nós que não puderam ser removidos na etapa anterior.

As B-splines resultantes passam a ter 16 pontos de controle e vetores nós iguais entre si.

5.4 - Obtenção de uma seção plana

GRANDINE; KLEIN IV (1997) sugerem um algoritmo para obter seções planas de uma superfície paramétrica. Porém, neste caso, o problema pode ser simplificado empregando-se um algoritmo de busca linear baseado no método da dicotomia, fixando-se o parâmetro u e variando-se v até que o valor de **Z**=**Z**(u,v) da superfície esteja próximo do valor desejado para uma dada precisão. Foi possível empregar esse método porque se conhece o domínio de busca e a topologia do resultado, i. e.:

a) o valor de **Z** procurado existe na superfície;

b) Para um dado **Z** a topologia resultante da seção é dada pela fig. 5.6:

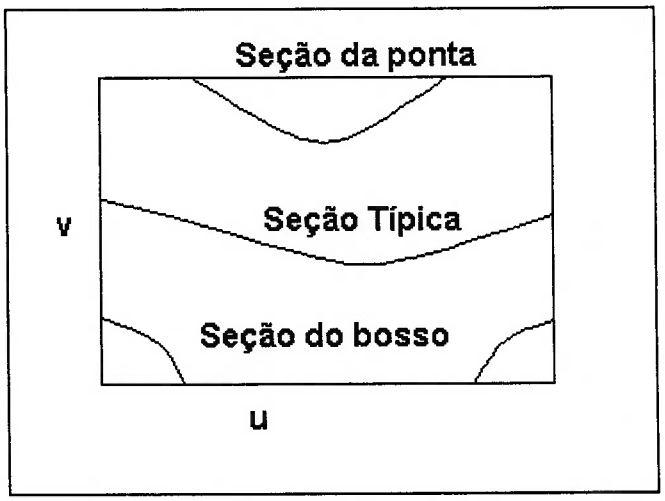

Fig. 5.6 - Topologia de uma seção plana

5.5 - Seções planas que interceptam o bosso

Uma seção plana intercepta o bosso quando o valor de Z_C fornecido é menor que $0,2R_p$. Neste caso, obtém-se o traço da superfície cilíndrica de raio $0,2R_p$ com plano $Z=Z_C$ O resultado é um retângulo definido pelos pontos (X_{min}, Y_c, Z_C) e (X_{max}, $-Y_c$, Z_C). Com auxílio da fig. 5.7 pode-se obter a equação (5-9) para o cálculo da coordenada Y_C.

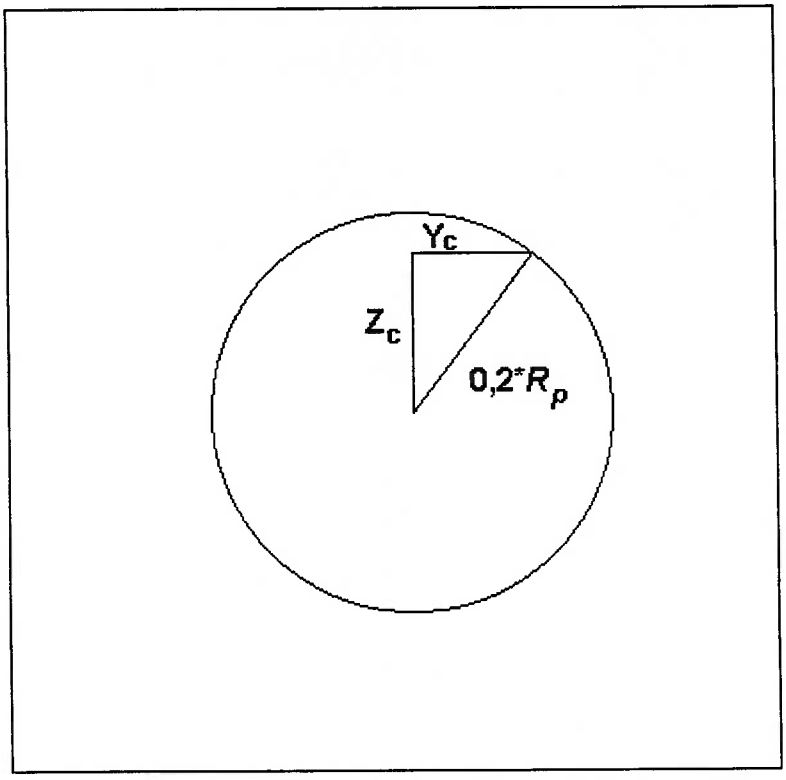

Fig. 5.7 - Cálculo da coordenada Y_C da intercecção com o bosso

$$Y_c = \sqrt{(0.2 * R_p)^2 - Z_c^2} \qquad (5\text{-}11)$$

As coordenadas $\mathbf{X}_{min}$ e $\mathbf{X}_{max}$ são calculadas obtendo-se os valores mínimo e máximo, respectivamente, no perfil 0,2*R_p, ou seja para u=0 na superfície. na fig. 5.8 é mostrado um exemplo de seção plana com intersecção no bosso.

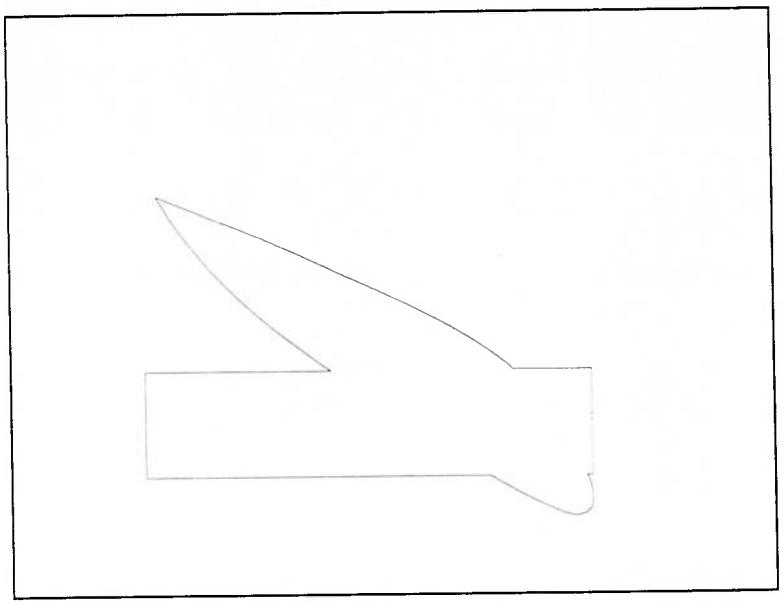

Fig. 5.8 - Exemplo de intersecção com o bosso

6 - IMPLEMENTAÇÃO COMPUTACIONAL

A fim de mostrar a aplicação do método exposto nos capítulos anteriores desenvolveu-se um aplicativo computacional em linguagem C++, compilado segundo as orientações contidas em OBJECTARX Developer's Guide... (1997), OBJECTARX Reference Manual... (1997) e ADSARX Developer's Guide... (1997) para funcionar no ambiente do Autocad R. 14.

Assim, foram criadas seis classes:

- B_Troost;
- kaplan;
- B_Spline;
- B_Spline_cl;
- B_Surf;
- B_Surf_cl.

E utilizadas seis classes existentes na biblioteca ARX:

- AcGePoint3d;
- AcGePoint3dArray;
- AcDb3dPolyline.
- AcDbDatabase
- AcDbObjectId
- AcGeVector3d

Foram empregadas as 10 funções listadas na tabela 6.1, sugeridas em OBJECTARX Developer's Guide... (1997), OBJECTARX Reference Manual... (1997) e ADSARX Developer's Guide... (1997). Algumas tiveram que ser corrigidas porque apresentavam erros.

AcDbEntity* selectEntity(AcDbObjectId& eId, AcDb::OpenMode openMode);
void getZucs(AcDbDatabase * pDb, AcGeVector3d& v);
int getPoint(AcGePoint3d& pF, char* pMessage, AcGePoint3d& pT);
int getPoint(char* pMessage, AcGePoint3d& pT);
int sa_u2w(ads_point p1, ads_point p2);
int sa_w2u(ads_point p1, ads_point p2);
void initApp(void);
void unloadApp(void);
extern "C" AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode msg, void*);
void iterate(AcDbObjectId plineId);

Tabela 6.1 - Funções sugeridas nos arquivos de ajuda da biblioteca ARX

Outras classes da biblioteca ARX foram utilizadas nas 10 funções acima, porém, limitadas ao escopo de cada função. Assim, optou-se por não descrever essas classes adicionais.

Adicionalmente foram escritas as 24 funcões listadas na tabela 6.2.

Com as 12 classes e 34 funções apresentadas, elaborou-se o aplicativo HELIXSURF, com mais de 4000 linhas de código, utilizando o compilador Visual C++ 5.0.

void HELIXSURF(void);
void KAPLAN(void);
void bsplab(void);
void bsplab_cl(void);
int bvl(AcDbObjectId plineId, AcGePoint3d vl[]);
void DrawPoly(AcGePoint3dArray ptArr, char *la);
void EdCurv(void);
int findLayer(char *la);
void addLayer(char *la, int c);
void dcp(B_Spline sp, char *la);
void dcp(B_Spline_cl sp, char *la);
void pl2sp(double r, double phi, double xp, double yp, double& xe, double& ye, double& ze);
void sp2pl(double r, double phi, double xe, double ye, double ze, double& xp, double& yp);
void dsp(B_Spline sp, char *la);
void dsp(B_Spline_cl sp, char *la);
void dsppl(B_Spline sp, char *la, double r, double phi);
void dsppl(B_Spline_cl sp, char *la, double r, double phi);
void dknv(B_Spline sp, char *la);
void dknv(B_Spline_cl sp, char *la);
void DrawSurfaceControlPoints(B_Surf bls, char *la);
void DrawSurfaceControlPoints(B_Surf_cl bls, char *la);
void DrawRadialProfile(B_Surf bs, double rr, char *la);
void DrawRadialProfile(B_Surf_cl bs, double rr, char *la);
void UBgNUB(B_Spline& ubs, B_Spline_cl nubs, int nc, int k, double eps);

Tabela 6.2 - Funções adicionais

6.1 - A classe B_Troost

A classe B_Troost foi criada para permitir a geração de pás da série sistemática B-Troost. Possui a declaração apresentada em ALG 6-1:

```
class B_Troost
{
public:
    B_Troost();
    B_Troost(double dd, double zz, double aa, double p);
    void build(double dd, double zz, double aa, double p);
    ~B_Troost();
    double d, z, aea0, pd;
    struct tprof
    {
            int np;
             double xle[11], xte[11];
            double yfle[11], yble[11], yfte[11], ybte[11];
            double xe[41], ye[41], ze[41];
            int n0, k0;
            double arr, brr, cr, sr;
            double tle, tte;
    };
    tprof bl[10];
Private:
    static const double ccr[9];
    static const double arcr[9];
    static const double brcr[9];
    static const double ar[9];
    static const double br[9];
    static const double v1[9][20];
    static const double v2[9][20];
    static const double pc[20];
    static const double rr[9];
};
```

(ALG 6-1)

A inicialização das constantes da série sistemática, obtidas em PRINCIPLES of Naval Architecture, Volume II ... (1988), foi feita com o código apresentado em ALG 6-2:

```
const double B_Troost::ccr[9]    = {1.662,1.882,2.050,2.152,2.187,2.144,1.970,1.582,0.000};
const double B_Troost::arcr[9]   = {0.617,0.613,0.601,0.586,0.561,0.524,0.463,0.351,0.000};
const double B_Troost::brcr[9]   = {0.350,0.350,0.351,0.355,0.389,0.443,0.479,0.500,0.000};
const double B_Troost::ar[9] = {0.0526,0.0464,0.0402,0.0340,0.0278,0.0216,0.0154,0.0092,0.0030};
const double B_Troost::br[9] = {0.0040,0.0035,0.0030,0.0025,0.0020,0.0015,0.0010,0.0005,0.0000};

const double B_Troost::v1[9][20]  = {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
```

```
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0.0006,0.0022,0.0067,0.0169,0.0382},
{0.0522,0.042,0.033,0.019,0.01,0.004,0.0012,0,0,0,0,0,0.0008,0.0034,0.0085,0.0211,0.0328,0.05,0
  .0778,0.1278},
{0.1467,0.12,0.0972,0.063,0.0395,0.0214,0.0116,0.0044,0,0,0,0.0033,0.009,0.0189,0.0357,0.0637,
  0.0833,0.1088,0.1467,0.2181},
{0.2306,0.2040,0.1790,0.1333,0.0943,0.0623,0.0376,0.0202,0.0033,0,0.0027,0.0148,0.03,0.0503,0
  .079,0.1191,0.1445,0.1760,0.2186,0.2923},
{0.2826,0.2630,0.24,0.1967,0.157,0.1207,0.088,0.0592,0.0172,0,0.0049,0.0304,0.052,0.0804,0.11
  8,0.1685,0.2,0.2353,0.2821,0.3560}
        };

const double B_Troost::v2[9][20] ={{0,0.0975,0.19,0.36,0.51,0.64,0.75,0.84,0.96,1,
            0.9600,0.8400,0.75,0.6400,0.51,0.3600,0.2775,0.1900,0.0975,0},
    {0,0.0975,0.19,0.36,0.51,0.64,0.75,0.84,0.96,1,
    0.9600,0.8400,0.75,0.6400,0.51,0.3600,0.2775,0.1900,0.0975,0},
    {0,0.0975,0.19,0.36,0.51,0.64,0.75,0.84,0.96,1,
    0.9635,0.8520,0.7635,0.6545,0.5265,0.3765,0.2925,0.2028,0.1050,0},
    {0,0.0975,0.19,0.36,0.51,0.64,0.75,0.84,0.96,1,
    0.9675,0.8660,0.7850,0.6840,0.5615,0.4140,0.3300,0.2337,0.1240,0},
    {0,0.0965,0.1885,0.3585,0.5110,0.6415,0.7530,0.8426,0.9613,1,
    0.9690,0.8790,0.8090,0.7200,0.6060,0.4620,0.3775,0.2720,0.1485,0},
    {0,0.0950,0.1865,0.3569,0.5140,0.6439,0.7580,0.8456,0.9639,1,
    0.9710,0.8880,0.8275,0.7478,0.6430,0.5039,0.4135,0.3056,0.1750,0},
    {0,0.0905,0.1810,0.3500,0.5040,0.6353,0.7525,0.8415,0.9645,1,
    0.9725,0.8933,0.8345,0.7593,0.6590,0.5220,0.4335,0.3235,0.1935,0},
    {0,0.0800,0.1670,0.3360,0.4885,0.6195,0.7335,0.8265,0.9583,1,
    0.9750,0.8920,0.8315,0.7520,0.6505,0.5130,0.4265,0.3197,0.1890,0},
    {0,0.0640,0.1455,0.3060,0.4335,0.5842,0.6995,0.7984,0.9446,1,
    0.9750,0.8875,0.8170,0.7277,0.6190,0.4777,0.3905,0.2840,0.1560,0}
    };

const double B_Troost::pc[20] = {-1,-0.95,-0.9,-0.8,-0.7,-0.6,-0.5,-0.4,-0.2,0,0.2,0.4,0.5,0.6,0.7,
                                 0.8,0.85,0.9,0.95, 1};
const double B_Troost::rr[9] = {0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1};
```

(ALG 6-2)

A seguir serão apresentadas as funções membros da classe B_Troost.

a) O construtor padrão: B_Troost()

O construtor padrão serve para declarar uma instância da classe e alocar memória para a sua utilização. Nenhum cálculo é efetuado. A implementação é a seguinte:

```
B_Troost::B_Troost()
{

}
```

(ALG 6-3)

b) O construtor adicional : B_Troost(double, double, double, double)

O construtor adicional possui em sua assinatura 4 parâmetros do tipo double e permite criar uma pá fornecendo-se os valores do diâmetro (d), do número de pás (zz), da razão entre área expandida e área projetada (aa) e da relação entre passo e diâmetro (p). A implementação é mostrada a seguir:

```
B_Troost::B_Troost(double dd, double zz, double aa, double p)
{
        build(dd, zz, aa, p);
}
```

(ALG 6-4)

c) A funçao build

A função build efetua os cálculos da série e calcula as coordenadas de cada perfil no plano expandido e no espaço tri-dimensional. A implementação é mostrada abaixo:

```
void B_Troost::build(double dd, double zz, double aa, double p)
{
  int i,j,k;
  double re1,re2,radius,phi, dummyreal;
  d=dd;
  z=zz;
  aea0=aa;
  pd=p;

   for (i=1 ; i<=9; i++)

   {
      bl[i].np=40;
      bl[i].cr=ccr[i-1]*d*aea0/z;
      bl[i].sr=d*(ar[i-1]-br[i-1]*z);
      bl[i].tle=0;
      bl[i].tte=0;

      bl[i].arr=arcr[i-1]*bl[i].cr;
      bl[i].brr=brcr[i-1]*bl[i].cr;

      //calculo dos xte

      re1=(double)bl[i].arr - (double)bl[i].brr;
```

```
re2=(double)bl[i].cr - (double)bl[i].brr;
for (k=1; k<=10; k++)
   bl[i].xte[k] = re1+pc[k-1]*re2;

//calculo dos xle

dummyreal=(double)bl[i].arr - (double)bl[i].brr;
for (k=1 ; k<=10; k++)
  bl[i].xle[k]= dummyreal + pc[k+10-1]*bl[i].brr;


//calculo da face


for (k=1; k<=10; k++)
{
    //- bordo de fuga
   bl[i].yfte[k] = v1[10-i-1][k-1]*(bl[i].sr-bl[i].tte);

    //- bordo de ataque
   bl[i].yfle[k] = v1[10-i-1][10+k-1]*(bl[i].sr-bl[i].tle);
}

//calculo do dorso

for (k=1; k<=10; k++)
{
     //- bordo de fuga
    bl[i].ybte[k] = ( v1[10-i-1][k-1] + v2[10-i-1][k-1] ) *
                                                ( bl[i].sr-bl[i].tte ) + bl[i].tte;

     //- bordo de ataque
    bl[i].yble[k] = (v1[10-i-1][10+k-1]+v2[10-i-1][10+k-1]) *
                                              (bl[i].sr-bl[i].tle)+bl[i].tle;

}

   //calculo das coordenadas espaciais

radius=(double)rr[i-1]*d/2;
phi=(double)atan(pd*d/2/3.141592/radius);

k=1;
   for (j=1; j<=10; j++)
   {      // bordo de fuga - dorso

      bl[i].xe[k]= bl[i].xte[j]*sin(phi)+bl[i].ybte[j]*cos(phi);
      bl[i].ye[k]= -radius*sin((bl[i].xte[j]*cos(phi)-bl[i].ybte[j]*sin(phi))/radius);
      bl[i].ze[k]=  radius*cos((bl[i].xte[j]*cos(phi)-bl[i].ybte[j]*sin(phi))/radius);
      k++;

   }
   for (j=1; j<=10; j++)
   {      //bordo de atque - dorso

      bl[i].xe[k]= bl[i].xle[j]*sin(phi)+bl[i].yble[j]*cos(phi);
      bl[i].ye[k]= -radius*sin((bl[i].xle[j]*cos(phi)-bl[i].yble[j]*sin(phi))/radius);
      bl[i].ze[k]=  radius*cos((bl[i].xle[j]*cos(phi)-bl[i].yble[j]*sin(phi))/radius);

      k++;

   }
   for (j=10; j>= 1; j--)
   {      // bordo de ataque - face

      bl[i].xe[k]= bl[i].xle[j]*sin(phi)+bl[i].yfle[j]*cos(phi);
      bl[i].ye[k]= -radius*sin((bl[i].xle[j]*cos(phi)-bl[i].yfle[j]*sin(phi))/radius);
      bl[i].ze[k]=  radius*cos((bl[i].xle[j]*cos(phi)-bl[i].yfle[j]*sin(phi))/radius);
```

```
            k++;

        }
        for (j=10; j>=1; j--)
        {       // bordo de fuga - face

            bl[i].xe[k]= bl[i].xte[j]*sin(phi)+bl[i].yfte[j]*cos(phi);
            bl[i].ye[k]= -radius*sin((bl[i].xte[j]*cos(phi)-bl[i].yfte[j]*sin(phi))/radius);
            bl[i].ze[k]=  radius*cos((bl[i].xte[j]*cos(phi)-bl[i].yfte[j]*sin(phi))/radius);

            k++;

        }

    }
}
```

(ALG 6-5)

d) O destrutor

O destrutor declarado para esta classe nada executa porque não há alocação dinâmica de memória e as instâncias desta classe declaradas no aplicativo são utilizadas até o fim da execução. O código do destrutor é:

```
B_Troost::~B_Troost()
{

}
```

(ALG 6-6)

6.2 - A classe kaplan

A classe kaplan foi criada para permitir a geração de pás da série Kaplan e possui a seguinte declaração:

```
class kaplan{
public:
                kaplan();
                kaplan(double dd, double zz, double aa, double p);
                void build(double dd, double zz, double aa, double p);

                ~kaplan();
                double d, z, aea0, pd;
                struct tprof{
                int np;
                 double xle[11],xte[11];
                double yfle[11],yble[11],yfte[11],ybte[11];
                double xe[41],ye[41],ze[41];
                int n0,k0;
                double arr,brr,cr,sr,mtd, xmt;
                double tle,tte;
                                        };

 tprof bl[10];

private:

                static const double ccr[9];
                static const double clte[9];
                static const double clle[9];

                static const double mbt[9];

                static const double dmbt[9];
                static const double face[9][12];

                static const double back[9][12];

                static const double pc[13];
                static const double rr[9];

};                                                              (ALG 6-7)
```

A inicialização das constantes da série foi feita com as seguintes linhas de código:

```
const double kaplan::ccr[9]    = {0.6715,0.7659,0.8519,0.9301,1.0000,1.0586,1.1008,1.1266,1.1288};

const double kaplan::clte[9]   = {0.3021,0.3617,0.4145,0.4599,0.4987,0.5293,0.5504,0.5633,0.5644};
const double kaplan::clle[9]   = {0.3694,0.4042,0.4374,0.4702,0.5013,0.5293,0.5504,0.5633,0.5644};

const double kaplan::mbt[9]    = {0.0400,0.0352,0.0300,0.0245,0.0190,0.0138,0.0092,0.0061,0.0050};
const double kaplan::dmbt[9]   = {0.3498,0.3976,0.4602,0.4913,0.4998,0.5000,0.5000,0.5000,0.5000};

const double kaplan::face[9][12]  =
{
                {0.2021,0.0729,0.0177,0.0010,0.0000,0.0021,0.0146,0.0437,0.1052,0.1604,0.2062,0.3333},
                {0.1385,0.0462,0.0107,0.0000,0.0000,0.0012,0.0083,0.0272,0.0615,0.0828,0.1030,0.2118},
                {0.0917,0.0236,0.0056,0.0000,0.0000,0.0000,0.0042,0.0139,0.0292,0.0389,0.0444,0.1347},
                {0.0662,0.0068,0.0017,0.0000,0.0000,0.0000,0.0017,0.0051,0.0102,0.0136,0.0153,0.0781},
                {0,0,0,0,0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0,0,0,0,0}
};

const double kaplan::back[9][12] =
{
                {0.0000,0.3823,0.6365,0.8240,0.9500,0.9792,0.9083,0.7719,0.5500,0.3875,0.2740,0.0000},
                {0.0000,0.3905,0.6663,0.8414,0.9586,0.9763,0.9006,0.7562,0.5302,0.3787,0.2757,0.0000},
                {0.0000,0.4056,0.6694,0.8569,0.9625,0.9722,0.8889,0.7361,0.5000,0.3472,0.2583,0.0000},
                {0.0000,0.4177,0.6859,0.8642,0.9660,0.9677,0.8710,0.7046,0.4584,0.3022,0.2224,0.0000},
                {0.0000,0.4358,0.6826,0.8589,0.9647,0.9647,0.8589,0.6826,0.4358,0.2859,0.2044,0.0000},
                {0.0000,0.4531,0.6924,0.8633,0.9658,0.9658,0.8633,0.6924,0.4531,0.3079,0.2288,0.0000},
                {0.0000,0.4816,0.7084,0.8704,0.9676,0.9676,0.8704,0.7084,0.4816,0.3439,0.2690,0.0000},
                {0.0000,0.5175,0.7294,0.8809,0.9717,0.9717,0.8809,0.7294,0.5175,0.3887,0.3187,0.0000},
                {0.0000,0.5200,0.7300,0.8800,0.9700,0.9700,0.8800,0.7300,0.5200,0.3925,0.3231,0.0000}
};

const double kaplan::pc[13] = {-1.0,-0.8,-0.6,-0.4,-0.2,0.0,0.2,0.4,0.6,0.8,0.9,0.95,1.0};
const double kaplan::rr[9] = {0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1};
```

(ALG 6-8)

A posição espacial dos pontos de um dado perfil é armazenada nos vetores xe[], ye[] e ze[] da estrutura tprof.

As constantes privadas da classe contém os coeficientes da série sistemática Kaplan obtidos em PRINCIPLES of Naval Architecture, Volume II ... (1988), que aliás contém erros.

No caso específico da série Kaplan há 26 pontos (dois repetidos no bordo de ataque) por perfil e um total de 9 perfis.

A seguir serão mostradas as funções membros da classe kaplan.

a) O construtor padrão: kaplan()

O construtor padrão serve para declarar uma instância da classe e alocar memória para a sua utilização. Nenhum cálculo é efetuado. A implementação é a seguinte:

```
kaplan::kaplan()
{

}
```

(ALG 6-9)

b) O construtor adicional : kaplan(double, double, double, double)

O construtor adicional possui em sua assinatura 4 parâmetros do tipo double e permite criar uma pá fornecendo-se os valores do diâmetro (dd), do número de pás (zz), da razão entre área expandida e área projetada (aa) e da relação entre passo e diâmetro (p). A implementação é mostrada a seguir:

```
kaplan::kaplan(double dd, double zz, double aa, double p)
{
        build(dd, zz, aa, p);
}
```

(ALG 6-10)

c) A funçao build

A função build efetua os cálculos da série e calcula as coordenadas de cada perfil no plano expandido e no espaço tri-dimensional. A implementação é mostrada abaixo.

```
void kaplan::build(double dd, double zz, double aa, double p)
{
  int i,j,k;
  double radius,phi,lb06r;
  d=dd;
  z=zz;
  aea0=aa;
  pd=p;
  lb06r=(double)1.969*aea0/z*d;

   for (i=1 ; i<=9; i++)

   {
                    bl[i].np=25; // numero total de pontos em cada perfil = 26-1
     bl[i].cr=ccr[i-1]*lb06r; // comprimento da corda

     bl[i].sr=d*mbt[i-1]; // espessura máxima da pá
     bl[i].tle=0;
     bl[i].tte=0;

     bl[i].arr=clle[i-1]*bl[i].cr; // distancia da linha de centro ao bordo de ataque
     bl[i].brr=clte[i-1]*bl[i].cr; // distancia da linha de centro ao bordo de fuga
                    bl[i].mtd=dmbt[i-1]*bl[i].cr; // distancia do bordo de ataque ao ponto de maxima
          espessura
                    bl[i].xmt=bl[i].arr-bl[i].mtd; // abscissa do ponto de maxima espessura


     //abscissas do lado do bordo de fuga

     for (k=1; k<=6; k++)
       bl[i].xte[k] = bl[i].xmt+pc[k-1]*(bl[i].cr-bl[i].mtd);


     // abscissas do lado do bordo de ataque

     for (k=1 ; k<=7; k++)
       bl[i].xle[k]= bl[i].xmt+pc[k+6-1]*bl[i].mtd;



     //face
     for (k=1; k<=5; k++)
     {
         //- bordo de fuga
       bl[i].yfte[k] = face[i-1][k-1]*(bl[i].sr-bl[i].tte);
                    }
                                        // - maxima espessura
                    bl[i].yfte[6] = 0;


                    for (k=1; k<=7; k++)
                    {
                    //- bordo de ataque
       bl[i].yfle[k] = face[i-1][6+k-2]*(bl[i].sr-bl[i].tle);
     }

     // dorso
     for (k=1; k<=5; k++)
```

```
    {
        //- bordo de fuga
        bl[i].ybte[k] = back[i-1][k-1]*(bl[i].sr-bl[i].tte )
                                                    + bl[i].yfte[k];
                    }
                    // maxima espessura
                    bl[i].ybte[6] = bl[i].sr;
                    for (k=1; k<=6; k++) // nao repetir ponta!! (logo sao  6, nao 7)
                    {
        //- bordo de ataque
        bl[i].yble[k] = back[i-1][6+k-2]*(bl[i].sr-bl[i].tle)
                                                    + bl[i].yfle[k];

    }

      //calculo das coordenadas 3d

    radius=(double)rr[i-1]*d/2;
    phi=(double)atan(pd*d/2/3.141592/radius);

    k=1;
       for (j=1; j<=6; j++)
       {       // bordo de fuga - dorso

          bl[i].xe[k]= bl[i].xte[j]*sin(phi)+bl[i].ybte[j]*cos(phi);
          bl[i].ye[k]= -radius*sin((bl[i].xte[j]*cos(phi)-bl[i].ybte[j]*sin(phi))/radius);
          bl[i].ze[k]=  radius*cos((bl[i].xte[j]*cos(phi)-bl[i].ybte[j]*sin(phi))/radius);
k++;

       }
       for (j=1; j<=6; j++) // no need to repeat tip!! (so stop at 6, not 7
       {       // bordo de ataque - dorso

          bl[i].xe[k]= bl[i].xle[j]*sin(phi)+bl[i].yble[j]*cos(phi);
          bl[i].ye[k]= -radius*sin((bl[i].xle[j]*cos(phi)-bl[i].yble[j]*sin(phi))/radius);
          bl[i].ze[k]=  radius*cos((bl[i].xle[j]*cos(phi)-bl[i].yble[j]*sin(phi))/radius);
k++;

       }
       for (j=7; j>= 1; j--)
       {       // bordo de ataque - face

          bl[i].xe[k]= bl[i].xle[j]*sin(phi)+bl[i].yfle[j]*cos(phi);
          bl[i].ye[k]= -radius*sin((bl[i].xle[j]*cos(phi)-bl[i].yfle[j]*sin(phi))/radius);
          bl[i].ze[k]=  radius*cos((bl[i].xle[j]*cos(phi)-bl[i].yfle[j]*sin(phi))/radius);
k++;

       }
       for (j=6; j>=1; j--)
       {       //bordo de fuga -  face

          bl[i].xe[k]= bl[i].xte[j]*sin(phi)+bl[i].yfte[j]*cos(phi);
          bl[i].ye[k]= -radius*sin((bl[i].xte[j]*cos(phi)-bl[i].yfte[j]*sin(phi))/radius);
          bl[i].ze[k]=  radius*cos((bl[i].xte[j]*cos(phi)-bl[i].yfte[j]*sin(phi))/radius);
k++;

       }

   }
}
```

(ALG 6-11

d) O destrutor

O destrutor declarado para esta classe nada executa porque não há alocação dinâmica de memória e as instâncias desta classe declaradas no aplicativo são utilizadas até o fim da execução. O código do destrutor é:

```
kaplan::~kaplan()
{

}
```

(ALG 6-12)

6.3 - A classe B_Spline

A implementação computacional para aproximar ou interpolar um conjunto de pontos por uma curva B-spline uniforme é feita com auxílio da classe B_Spline apresentada a seguir:

```
class B_Spline{
public:
        B_Spline();
        ~B_Spline();
        // pontos de controle
        double bx[maxCP], by[maxCP], bz[maxCP];

        // parametros da spline

        int n0;  //numero de pontos de controle
        int k0;  //ordem  = grau +1

        int knm; // valor maximo do vetor no
        int nn; // numero de nos -em geral,  n0+k0
        int knots[maxKnots];  // vetor no

        int knot(int v[], int n, int k);
        void saveknots();
        int FindSpan(double u);
        void BasisRow(double N[], double u);

        void bsp(double N[][10], int v[], int k, int ip, double t);
        double S(double b[], double u);

        void g(long double a[][50], int nx, int ny);
        void aprox(double x[], double y[], double z[],
                        int npp, int nn, int kk);
        void adj(double x[], double y[], double z[],
                        int npp, int kk);
        void SmoothKnot(int kn);


private:

};
```

(ALG 6-13)

Algumas de suas propriedades-membro são a seguir descritas:

n0 : número de pontos de controle

k0 : ordem do polinômio interpolante, igual ao grau + 1

knm : valor máximo do vetor nó, igual a n0-k0+1

knots[] : vetor nó;

bx[], by[], bz[] : coordenadas espaciais dos pontos de controle

a) O construtor padrão: B_Spline()

Esta classe possui apenas o construtor padrão, dado pelo código a seguir:

```
B_Spline::B_Spline()
{
}
```

(ALG 6-14)

b) O destrutor

O destrutor declarado para esta classe nada executa porque não há alocação dinâmica de memória e as instâncias desta classe declaradas no aplicativo são utilizadas até o fim da execução. O código do destrutor é:

```
B_Spline::~B_Spline()
{
}
```

(ALG 6-15)

c) A função knot

A função knot permite calcular um vetor nó uniforme a partir da quantidade de pontos de controle e da ordem da B-spline. O código desta função é:

```
int B_Spline::knot(int v[], int n, int k)
{

   int m, i;
   m=n-k+1;

   for (i=1;i<=k;i++)     v[i] = 0;
   for (i=k+1;i<=n+1;i++) v[i] = v[i-1]+1;
   for (i=n+2;i<=n+k;i++) v[i] = v[i-1];
                  return m;
}
```

(ALG 6-16)

d) Função saveknots

Esta função chama a função knot armazenando os valores dos nós na propriedade-membro knots[]. Adicionalmente calcula o valor máximo do vetor nó (knm) e a quantidade de nós (nn), armazenando-os nas propriedades-membro correspondentes. O código é o seguinte:

```
void B_Spline::saveknots()
{
                  knm = knot(knots, n0, k0);
                  nn = n0+k0;

}
```

(ALG 6-17)

e) A função Findspan

Esta função é sugerida por PIEGL; TYLER (1997) e permite obter o índice do vetor nó cujo valor do parâmetro é o máximo menor que um parâmetro dado. O código é:

```
int B_Spline::FindSpan(double u)
{
        int low, high, mid;
        if (u==knots[n0]) return(n0-1);
        low=k0-1;
        high=n0+1;
        mid=(low+high)/2;
        while (u<knots[mid] || u>=knots[mid+1])
        {
            if (u<knots[mid])   high=mid;
            else                low=mid;
            mid = (low+high)/2;
        }
        return(mid);
}
```

(ALG 6-18)

f) A função BasisRow

Esta função, também sugerida por PIEGL; TYLER (1997), retorna um vetor contendo as funções base não-nulas para um dado parâmetro. O código é:

```
void B_Spline::BasisRow(double N[], double u)
{
    double M[10][10];
    int i, ip;

    ip=k0+1;

    while (u>knots[ip]) ip++;
    ip--;
    bsp(M, knots, k0, ip, u);
    for (i=1; i<=n0; i++) N[i]=0;
    for (i=1; i<=k0; i++) N[ip-k0+i]=M[i][k0];

}
```

(ALG 6-19)

g) A função bsp

Esta função, efetivamente utilizada, no aplicativo é semelhante à função BasisRow. O código é:

```
void B_Spline::bsp(double N[][10], int v[], int k, int ip, double t)
{

 int i,j;
 double f;
 double dp[60], dm[60];

 N[1][1]=1;

 for (i=1; i<=k-1; i++)
 {
                dp[i]=v[ip+i]-t;
                dm[i]=t-v[ip+1-i];
                N[1][i+1]=0;
                for (j=1 ; j<=i; j++)
                 {
                      f          = N[j][i]/(dp[j] + dm[i+1-j]);
                      N[j][i+1]  = N[j][i+1] + dp[j]*f;
                      N[j+1][i+1] = dm[i+1-j]*f;
                 }
 }
}
```

(ALG 6-20)

h) A função S

A função-membro S() permite obter as coordenadas x, y, z de um ponto na curva aproximada a partir do parâmetro u, da seguinte forma:

x = S(bx[], u)

y = S(by[], u)

z = S(bz[], u)

O código é:

```
double B_Spline::S(double b[], double u)
{
        double  sum;
        int    i, k;
        double N[10][10];

        k=k0+1;
        while (u> knots[k]) k++;
        k--;
        bsp(N,knots,k0,k,u);
        sum=0;
        for (i=1 ;i<=k0; i++) sum=sum+b[k-k0+i]*N[i][k0];

        return sum;

}
```

(ALG 6-21)

i) A função g

Esta função implementa o algoritmo de eliminação de Gauss baseado no algoritmo sugerido em ENGELN-MÜLLGES; UHLIG (1996). O código é:

```
void B_Spline::g(long double a[][50], int nx, int ny)
{

int   i, j, l, k, jx;
long double r, pivot, den;

for (l=1 ;l<= nx-1 ; l++)
   {
    for (i=l+1; i<=nx; i++ )
     {
      if (fabs(a[l][l]) < fabs(a[i][l] ))
       for (jx=l ; jx<=ny; jx++ )
        {
                r=a[i][jx];
                a[l][jx]=a[i][jx];
                a[i][jx]=r;
        }
      pivot=a[l][l];
     }

    for (i=l+1 ; i<= nx ; i++)
     {
      den=a[i][l];
                        for (j=1; j<=l; j++) a[i][j]=0;
      for (j=l+1 ; j<= ny; j++ )
         a[i][j] = a[i][j]-den/pivot*a[l][j];
      }
   }
```

```
// SUBSTITUICAO

 pivot=a[nx][nx];
 for (jx=nx+1; jx<= ny; jx++ )
                a[nx][jx]=a[nx][jx]/pivot;

 for (i=nx-1; i>=1; i--)
 {
                pivot=a[i][i];
                for (k=nx+1; k<=ny; k++)
                {

                    den=a[i][k];
                   for (j=i+1; j<=nx; j++)
                    den = den - a[i][j]*a[j][k];
                    a[i][k]=den/pivot;
                }
 }
}
```

(ALG 6-22)

j) A função aprox

Esta função permite aproximar um conjunto de npp pontos (x[], y[], z[]), por uma B-spline uniforme de ordem kk, com nn (nn ≤ npp) pontos de controle. A função efetua as seguintes operações:

- armazena n0, k0;
- calcula o vetor dos nós (knots[])e o valor do nó máximo (knm) chamando a função knot();
- monta o sistema de equações na matriz c[][] utilizando um parâmetro u calculado a partir da distância entre os pontos a serem ajustados e as bases B-splines calculadas segundo o algoritmo dede Boor implementado na função bsp()
- resolve o sistema chamando a função g() que utiliza-se o método da eliminação de Gauss

- armazena as coordenadas dos pontos de controle nos vetores bx[], by[], bz[]

```
void B_Spline::aprox(double x[], double y[], double z[],
                                        int np, int nn, int kk)
{
                int i,j, k ,ip, ii;
double u[100];
long double c[50][50], a[50][50], at[50][50];
double N[10][10];

                n0=nn;
                k0=kk;
knm=knot(knots, n0, k0);

 u[1]=0;
 for (i=2; i<=np; i++)
                u[i]=u[i-1] + sqrt((x[i]-x[i-1])*(x[i]-x[i-1]) +
                (y[i]-y[i-1])*(y[i]-y[i-1]) +
                (z[i]-z[i-1])*(z[i]-z[i-1]));

 for (i=1; i<=np; i++) u[i]=u[i]*knm/u[np];

 for (i=1; i<= 49; i++)
   for (j=1; j<=49; j++) c[i][j]=a[i][j]=at[i][j]=0;

 for (i=1; i<=np; i++)
 {
   ip=k0+1;

   while (u[i]>knots[ip]) ip++;

   ip--;
bsp(N,knots,k0,ip,u[i]);

   for (ii=1; ii<=k0; ii++) a[i][ip-k0+ii]=N[ii][k0];
 }

 for (i=1; i<=np;i++)
for (j=1; j<=n0; j++) at[j][i]=a[i][j];

for (i=1; i<=n0; i++)
    for (j=1; j<=n0; j++)
    {
       c[i][j]=0.;
       for (k=1; k<=np; k++)
                                        c[i][j]=c[i][j]+at[i][k]*a[k][j];
    }

for (i=1; i<=n0; i++)
 {
    c[i][n0+1]=0;
   c[i][n0+2]=0;
   c[i][n0+3]=0.;

    for (k=1; k<= np; k++)
   {

        c[i][n0+1]=c[i][n0+1]+at[i][k]*x[k];
        c[i][n0+2]=c[i][n0+2]+at[i][k]*y[k];
        c[i][n0+3]=c[i][n0+3]+at[i][k]*z[k];
    }
 }
```

```
g(c, n0, n0+3);
for( i=1; i<= n0; i++)
{
                bx[i]=c[i][n0+1];
                 by[i]=c[i][n0+2];
                 bz[i]=c[i][n0+3];
}

bx[1]=x[1];
by[1]=y[1];
bz[1]=z[1];

bx[n0]=x[np];
by[n0]=y[np];
bz[n0]=z[np];

}
```

(ALG 6-23)

l) A função adj

Esta função interpola uma curva B-spline uniforme de grau kk, que passa por um conjunto de npp pontos no espaço. O código é:

```
void B_Spline::adj(double x[], double y[], double z[], int npp, int kk)
{
                int i,j,ip, ii;
                double u[100];
                long double c[50][50];
                double N[10][10];

                n0=npp;
                k0=kk;
                saveknots();
                u[1]=0.0;
                for (i=2; i<=npp; i++)
                        u[i] = u[i-1] + sqrt((x[i]-x[i-1])*(x[i]-x[i-1]) +
                                            (y[i]-y[i-1])*(y[i]-y[i-1]) +
                                            (z[i]-z[i-1])*(z[i]-z[i-1]));

 for (i=1; i<=npp; i++) u[i]=u[i]*(double)knm/u[npp];

 for (i=1; i<=49; l++)
for (j=1; j<=49; j++) c[i][j]=(long double)0.0;

 for (i=1; i<=n0; i++)
 {
   ip=k0+1;

   while (u[i]>knots[ip]) ip++;
   ip--;

   bsp(N, knots, k0, ip, u[i]);

   for (ii=1; ii<=k0; ii++) c[i][ip-k0+ii]=(long double)N[ii][k0];
 }
```

```
for (i=1; i<=n0; i++)
{
    c[i][n0+1]=x[i];
    c[i][n0+2]=y[i];
    c[i][n0+3]=z[i];
}

FILE *f;
 f = fopen( "Nij.txt", "a+" );
 if( f == NULL ) printf("\nNao foi possivel abrir Nij.txt");
 else
 {              fprintf(f, "\n\n\n\n");
                  for (i=1; i<=n0; i++)
                      {          for (j=1; j<=n0+3; j++)
                                          fprintf(f, "%f ", c[i][j]);
                                 fprintf(f, "\n");
                      }
 }
 fclose(f);

 g(c,n0,n0+3);
 for( i=2; i< n0; i++)
 {
   bx[i]=c[i][n0+1];
                 by[i]=c[i][n0+2];
                 bz[i]=c[i][n0+3];
 }

 bx[1]=x[1];
 by[1]=y[1];
 bz[1]=z[1];

 bx[n0]=x[npp];
 by[n0]=y[npp];
 bz[n0]=z[npp];
}
```

(ALG 6-24)

m) A função SmoothKnot

Esta função, baseada na proposta de FARIN (1996) de suavização de curvas, permite suavizar a curva B-spline na vizinhança do ponto de controle cujo índice é dado por kn. Essa proposta pode ser visualizada com auxílio da fig. 6.1 e das expressões (6-1), (6-2) e (6-3).

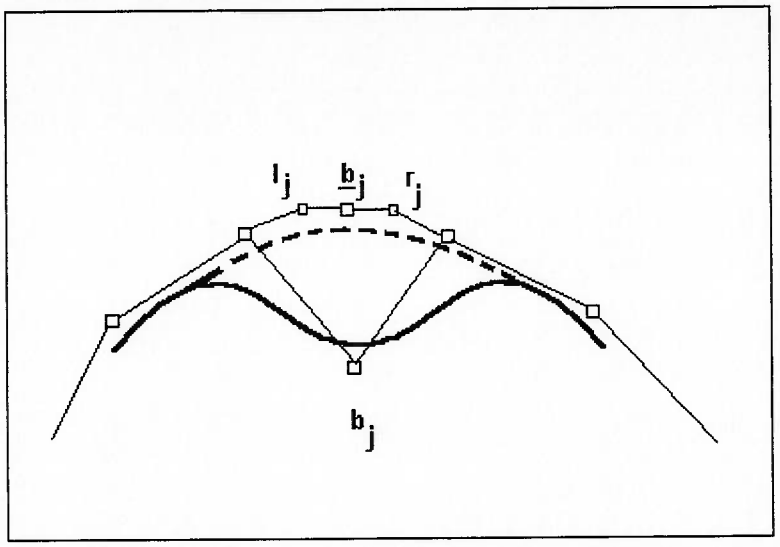

Figura 6.1 - Visualização geométrica do processo de suavização sugerido por FARIN (1996).

$$\underline{\mathbf{b}}_j = \frac{(u_{j+2} - u_j)\mathbf{l}_j + (u_j - u_{j-2})\mathbf{r}_j}{u_{j+2} - u_{j-2}} \qquad (6\text{-}1)$$

$$\mathbf{l}_j = \frac{(u_{j+1} - u_{j-3})\mathbf{d}_{j-1} - (u_{j+1} - u_j)\mathbf{d}_{j-2}}{u_j - u_{j-3}} \qquad (6\text{-}2)$$

$$\mathbf{r}_j = \frac{(u_{j+3} - u_{j-1})\mathbf{d}_{j+1} - (u_j - u_{j-1})\mathbf{d}_{j+2}}{u_{j+3} - u_j} \qquad (6\text{-}3)$$

A implementação computacional é:

```
void B_Spline::SmoothKnot(int kn)
{
        double x, y, z, xr, xl, yr, yl, zr, zl, xnew, ynew, znew;
        int i;
        if ((kn>2) && (kn<n0-1))
        {
```

```
        x=bx[kn];
        y=by[kn];
        z=bz[kn];

        i=kn+k0;

        xl=((knots[i+1]-knots[i-3])*bx[kn-1]-(knots[i+1]-knots[i])*bx[kn-2])/(knots[i]-knots[i-
3]);
        yl=((knots[i+1]-knots[i-3])*by[kn-1]-(knots[i+1]-knots[i])*by[kn-2])/(knots[i]-knots[i-
3]);
        zl=((knots[i+1]-knots[i-3])*bz[kn-1]-(knots[i+1]-knots[i])*bz[kn-2])/(knots[i]-knots[i-
3]);

        xr=((knots[i+3]-knots[i-1])*bx[kn+1]-(knots[i]-knots[i-1])*bx[kn+2])/(knots[i+3]-
knots[i]);
        yr=((knots[i+3]-knots[i-1])*by[kn+1]-(knots[i]-knots[i-1])*by[kn+2])/(knots[i+3]-
knots[i]);
        zr=((knots[i+3]-knots[i-1])*bz[kn+1]-(knots[i]-knots[i-1])*bz[kn+2])/(knots[i+3]-
knots[i]);

        xnew=((knots[i+2]-knots[i])*xl+(knots[i]-knots[i-2])*xr)/(knots[i+2]-knots[i-2]);
        ynew=((knots[i+2]-knots[i])*yl+(knots[i]-knots[i-2])*yr)/(knots[i+2]-knots[i-2]);
        znew=((knots[i+2]-knots[i])*zl+(knots[i]-knots[i-2])*zr)/(knots[i+2]-knots[i-2]);

        bx[kn]=xnew;
        by[kn]=ynew;
        bz[kn]=znew;
    }
}
```

(ALG 6-25)

6.4 - A classe B_spline_cl

A implementação computacional para aproximar ou interpolar um conjunto de pontos por uma curva B-spline não-uniforme é feita com auxílio da classe B_Spline_cl (o sufixo cl refere-se a "chord length" - significando que a parametrização é feita em função da distância entre os pontos da poligonal que será aproximada ou interpolada) apresentada a seguir:

```
class B_Spline_cl{
public:
        B_Spline_cl();
        ~B_Spline_cl();
        // pontos de controle
        double bx[maxCP], by[maxCP], bz[maxCP];
        // parametros
        int n0;  //numero de pontos de controle
        int k0;  //ordem  = grau +1
        double knm; // valor maximo do vetor no
        int nn; // numero de nos - em geral  n0+k0
        double knots[maxKnots];  // vetor no
        double knot(int np, double x[], double y[], double z[],
                    double v[], int n, int k);
        void saveknots(int np, double x[], double y[], double z[]);
        int FindSpan(double u);
        void BasisRow(double N[], double u);
        void bsp(double N[][maxBase], double v[], int k, int ip,
              double u);
        double S(double b[], double u);
        void g(long double a[][maxG], int nx, int ny);
        void aprox(double x[], double y[], double z[],
                    int npp, int nn, int kk);
        void adj(double x[], double y[], double z[],
                   int npp, int kk);
        void SmoothKnot(int kn);
        double length(double u, double eps);
        double ugl(double L, double eps);
        int InsertKnot(double u);
        int RemoveKnot(int r);
        int UpgradeKnots(int q);
        int KnotExists(double u);

private:

};
```

(ALG 6-26)

Algumas das propriedades-membro são descritas a seguir:

n0 : número de pontos de controle

k0 : ordem do polinômio interpolante, igual ao grau + 1

knm : valor máximo do vetor nó, igual a 1.0

knots[] : vetor nó;

bx[], by[], bz[] : coordenadas espaciais dos pontos de controle

a) O construtor padrão: B_Spline_cl()

Esta classe possui apenas o construtor padrão, dado pelo código a seguir:

```
B_Spline_cl::B_Spline_cl()
{
}
```

(ALG 6-27)

b) O destrutor

O destrutor declarado para esta classe nada executa porque não há alocação dinâmica de memória e as instâncias desta classe declaradas no aplicativo são utilizadas até o fim da execução. O código do destrutor é:

```
B_Spline::~B_Spline_cl()
{
}
```

(ALG 6-28)

c) A função knot

A função knot permite calcular um vetor nó não-uniforme a partir da poligonal dos pontos que serão aproximados ou interpolados, da quantidade de pontos de controle e da ordem da B-spline. O código desta função é:

```
double B_Spline_cl::knot(int np, double x[], double y[], double z[], double v[], int n, int k)
{
        double sum, u[maxKnots];
        int i, j;

        u[1]=(double)0;
        for (i=2; i<=np; i++)
        u[i]=u[i-1] + sqrt((x[i]-x[i-1])*(x[i]-x[i-1]) +
                                        (y[i]-y[i-1])*(y[i]-y[i-1]) +
                                        (z[i]-z[i-1])*(z[i]-z[i-1]) );
                                    for (i=1; i<np; i++) u[i]=u[i]/u[np];
        u[np]=(double)1;
for (i=1;i<=k;i++)    v[i] =(double) 0; //k nos iniciais

  for (i=k+1;i<=n;i++) //n-k nos internos
        {
            sum=(double)0;
            for (j=i-k+1; j<=i-1; j++) sum=sum+u[j];

            v[i] = sum/(double)(k-1);
        }
  for (i=n+1; i<=n+k; i++) v[i] = (double)1;   // k nos finais
        return v[n+k];
}
```

(ALG 6-29)

d) Função saveknots

Esta função chama a função knot armazenando os valores dos nós na propriedade-membro knots[]. Adicionalmente guarda o valor máximo do vetor nó (knm=1.0) e a quantidade de nós (nn), armazenando-os nas propriedades-membro correspondentes.O código é o seguinte:

```
void B_Spline_cl::saveknots(int np, double x[], double y[], double z[])
{
        knm = knot(np, x, y, z, knots, n0, k0);
        nn = n0+k0;
}
```

(ALG 6-30)

e) A função Findspan

Esta função permite obter o índice do vetor nó cujo valor do parâmetro é máximo, porém menor que o parâmetro u fornecido. O código é:

```
int B_Spline_cl::FindSpan(double u)
{
        int low;
        // high, mid;
        if (u==knots[n0+1]) return n0;
        low=k0+1;
        while(u>knots[low])low++;
        low--;
        return low;
}
```

(ALG 6-31)

f) A função BasisRow

Esta função, também sugerida por PIEGL; TYLER (1997), retorna um vetor contendo as funções base não-nulas para um dado parâmetro. O código é:

```
void B_Spline_cl::BasisRow(double N[], double u)
{
        double M[maxBase][maxBase];
        int i, ip;

         ip=FindSpan(u);
        bsp(M, knots, k0, ip, u);
        for (i=1; i<=n0; i++) N[i]=0;
        for (i=1; i<=k0; i++) N[ip-k0+i]=M[i][k0];

}
```

(ALG 6-32)

g) A função bsp

Esta função, efetivamente utilizada no aplicativo, é semelhante à função BasisRow. O código é:

```
void B_Spline_cl::bsp(double N[][maxBase], double v[], int k, int ip, double u)
{
 int i,j;
 double M[maxBase], left[maxBase], right[maxBase], saved, temp;
 M[0]=1;
 for (j=1; j<=k-1; j++)
 {
                    left[j]=u-v[ip+1-j];
                    right[j]=v[ip+j]-u;
                    saved=0;
                    for(i=0; i<j; i++)
                    {
                              temp=M[i]/(right[i+1]+left[j-i]);
                              M[i]=saved+right[i+1]*temp;
                              saved=left[j-i]*temp;
                    }
                    M[j]=saved;
 }
 for(j=1; j<=k; j++)N[j][k]=M[j-1];
}
```

(ALG 6-33)

h) A função S

Esta função retorna as coordenadas (x, y, z) de um ponto sobre a curva a partir dos pontos de controle e do parâmetro fornecido, da seguinte forma:

x = S(bx[], u)

y = S(by[], u)

z = S(bz[], u)

O código é:

```
double B_Spline_cl::S(double b[], double u)
{
        double  sum;
        int    i, k;
        double N[maxBase][maxBase];
        k=FindSpan(u);
        bsp(N,knots,k0,k,u);
        sum=(double)0;
        for (i=1 ;i<=k0; i++) sum=sum+b[k-k0+i]*N[i][k0];
        return sum;
}
```

(ALG 6-34)

i) A função g

Esta função implementa o algoritmo de eliminação de Gauss de forma idêntica à classe B_Spline. O código é:

```
void B_Spline_cl::g(long double a[][maxG], int nx, int ny)
{

int   i, j, l, k, jx;
long double r, pivot, den;

for (l=1 ;l<= nx-1 ; l++)
   {
    for (i=l+1; i<=nx; i++ )
     {
      if (fabs(a[l][l]) < fabs(a[i][l] ))
        for (jx=l ; jx<=ny; jx++ )
          {
            r=a[l][jx];
                                          a[l][jx]=a[i][jx];
                                          a[i][jx]=r;
          }
       pivot=a[l][l];
     }

    for (i=l+1 ; i<= nx ; i++)
     {
       den=a[i][l];
                        for (j=1; j<=l; j++) a[i][j]=0;
       for (j=l+1 ; j<= ny; j++ )
          a[i][j] = a[i][j]-den/pivot*a[l][j];
      }
   }

 // SUBSTITUICAO

  pivot=a[nx][nx];
  for (jx=nx+1; jx<= ny; jx++ )
                  a[nx][jx]=a[nx][jx]/pivot;
```

```
for (i=nx-1; i>=1; i--)
{
        pivot=a[i][i];
        for (k=nx+1; k<=ny; k++)
        {
            den=a[i][k];
    for (j=i+1; j<=nx; j++)
                den = den - a[i][j]*a[j][k];
            a[i][k]=den/pivot;
        }
 }
}
```

(ALG 6-35)

j) A função aprox

Esta função permite aproximar um conjunto de npp pontos (x[], y[], z[]), por uma B-spline não-uniforme de ordem kk, com nn (nn ≤ npp) pontos de controle. A função efetua as seguintes operações:

- armazena n0, k0;
- calcula o vetor dos nós (knots[])e o valor do nó máximo (knm) chamando a função knot();
- monta o sistema de equações na matriz c[][] utilizando um parâmetro u calculado a partir da distância entre os pontos a serem ajustados e as bases B-splines calculadas segundo o algoritmo dede Boor implementado na função bsp()
- resolve o sistema chamando a função g() que utiliza-se o método da eliminação de Gauss
- armazena as coordenadas dos pontos de controle nos vetores bx[], by[], bz[]

O código é o seguinte:

```
void B_Spline_cl::aprox(double x[], double y[], double z[],
                                        int np, int nn, int kk)
{
                int i,j, k ,ip, ii;
                double u[maxKnots];
                long double c[maxG][maxG], a[maxG][maxG], at[maxG][maxG];
                double N[maxBase][maxBase];

                n0=nn;
                k0=kk;
                saveknots(np, x, y, z);

 u[1]=(double)0;
 for (i=2; i<=np; i++)
                u[i]=u[i-1] + sqrt((x[i]-x[i-1])*(x[i]-x[i-1]) +
                                                        (y[i]-y[i-1])*(y[i]-y[i-1]) +
                                                        (z[i]-z[i-1])*(z[i]-z[i-1]));

 for (i=1; i<np; i++) u[i]=u[i]/u[np];
 u[np]=(double)1;

 for (i=1; i<= maxG-1; i++)
   for (j=1; j<=maxG-1; j++) c[i][j]=a[i][j]=at[i][j]=0;

 for (i=1; i<=np; i++)
 {
                ip=FindSpan(u[i]);
   bsp(N, knots, k0, ip, u[i]);

   for (ii=1; ii<=k0; ii++) a[i][ip-k0+ii]=N[ii][k0];
 }

 for (i=1; i<=np;i++)
for (j=1; j<=n0; j++) at[j][i]=a[i][j];

for (i=1; i<=n0; i++)
   for (j=1; j<=n0; j++)
   {
     c[i][j]=0.;
     for (k=1; k<=np; k++)
                                c[i][j]=c[i][j]+at[i][k]*a[k][j];
   }
for (i=1; i<=n0; i++)
 {
                c[i][n0+1]=0;
                c[i][n0+2]=0;
                c[i][n0+3]=0;

   for (k=1; k<= np; k++)
                {

                    c[i][n0+1]=c[i][n0+1]+at[i][k]*x[k];
                    c[i][n0+2]=c[i][n0+2]+at[i][k]*y[k];
                    c[i][n0+3]=c[i][n0+3]+at[i][k]*z[k];
                }
 }

 g(c, n0, n0+3);
 for( i=1; i<= n0; i++)
 {

                bx[i]=c[i][n0+1];
                 by[i]=c[i][n0+2];
                 bz[i]=c[i][n0+3];
```

```
}

bx[1]=x[1];
by[1]=y[1];
bz[1]=z[1];

bx[n0]=x[np];
by[n0]=y[np];
bz[n0]=z[np];

}
```

(ALG 6-36)

l) A função adj

Esta função interpola uma curva B-spline não-uniforme de grau kk, que passa por um conjunto de npp pontos no espaço. O código é:

```
void B_Spline_cl::adj(double x[], double y[], double z[],
                                        int npp, int kk)
{
        int i,j,ip, ii;
        double u[maxG];
        long double c[maxG][maxG];
        double N[maxBase][maxBase];

        n0=npp;
        k0=kk;
        saveknots(npp, x, y, z);
        u[1]=(double)0.0;
        for (i=2; i<=npp; i++)
            u[i] = u[i-1] + sqrt((x[i]-x[i-1])*(x[i]-x[i-1]) +
                                                    (y[i]-y[i-1])*(y[i]-y[i-1]) +
                                                    (z[i]-z[i-1])*(z[i]-z[i-1]));

 for (i=1; i<npp; i++) u[i]=u[i]/u[npp];
 u[npp]=(double)1.0;

for (i=1; i<=maxG-1; i++) //no! not 60!!!
   for (j=1; j<=maxG-1; j++) c[i][j]=0;

 for (i=1; i<=n0; i++)
 {
    ip=FindSpan(u[i]);
   bsp(N,knots,k0,ip,u[i]);
for (ii=1; ii<=k0; ii++) c[i][ip-k0+ii]=N[ii][k0];
 }

 for (i=1; i<=n0; i++)
 {
    c[i][n0+1]=x[i];
    c[i][n0+2]=y[i];
    c[i][n0+3]=z[i];
 }
FILE *f;
  f = fopen( "Nij.txt", "a+" );
  if( f == NULL ) printf("\nCould not open Nij.txt");
  else
  {             fprintf(f, "\n\n\n");
```

```
                for (i=1; i<=n0; i++)
                    {           for (j=1; j<=n0+3; j++)
                                        fprintf(f, "%f ", c[i][j]);
                                fprintf(f, "\n");
                    }
    }
    fclose(f);

    g(c,n0,n0+3);
    for( i=1; i<= n0; i++)
    {
                    bx[i] = c[i][n0+1];
                    by[i] = c[i][n0+2];
                    bz[i] = c[i][n0+3];
    }

    bx[1] = x[1];
    by[1] = y[1];
    bz[1] = z[1];

    bx[n0] = x[npp];
    by[n0] = y[npp];
    bz[n0] = z[npp];
}
```

(ALG 6-37)

m) A função SmoothKnot

Esta função, baseada na proposta de FARIN (1996) para suavização de curvas, permite suavizar a curva B-spline na vizinhança do ponto de controle cujo índice é dado por kn. O código é:

```
void B_Spline_cl::SmoothKnot(int kn)
{
                double x, y, z, xr, xl, yr, yl, zr, zl, xnew, ynew, znew;
                int i;
                if ((kn>2) && (kn<n0-1))
                {
                        x=bx[kn];
                        y=by[kn];
                        z=bz[kn];

                        i=kn+k0;

                        xl=((knots[i+1]-knots[i-3])*bx[kn-1]-(knots[i+1]-knots[i])*bx[kn-2])/(knots[i]-knots[i-
3]);
                        yl=((knots[i+1]-knots[i-3])*by[kn-1]-(knots[i+1]-knots[i])*by[kn-2])/(knots[i]-knots[i-
3]);
                        zl=((knots[i+1]-knots[i-3])*bz[kn-1]-(knots[i+1]-knots[i])*bz[kn-2])/(knots[i]-knots[i-
3]);

                        xr=((knots[i+3]-knots[i-1])*bx[kn+1]-(knots[i]-knots[i-1])*bx[kn+2])/(knots[i+3]-
knots[i]);
                        yr=((knots[i+3]-knots[i-1])*by[kn+1]-(knots[i]-knots[i-1])*by[kn+2])/(knots[i+3]-
knots[i]);
```

```
                zr=((knots[i+3]-knots[i-1])*bz[kn+1]-(knots[i]-knots[i-1])*bz[kn+2])/(knots[i+3]-
            knots[i]);

                xnew=((knots[i+2]-knots[i])*xl+(knots[i]-knots[i-2])*xr)/(knots[i+2]-knots[i-2]);
                ynew=((knots[i+2]-knots[i])*yl+(knots[i]-knots[i-2])*yr)/(knots[i+2]-knots[i-2]);
                znew=((knots[i+2]-knots[i])*zl+(knots[i]-knots[i-2])*zr)/(knots[i+2]-knots[i-2]);

                bx[kn]=xnew;
                by[kn]=ynew;
                bz[kn]=znew;
            }
}
```

(ALG 6-38)

n) A função length

Esta função calcula o comprimento sobre a curva para um dado parâmetro u e uma precisão eps. O código é:

```
double B_Spline_cl::length(double u, double eps)
{
        double L0, L1, du, x1, x2, y1, y2, z1, z2, sum, uu;
        du=u/100;
        x1=S(bx, (double)0);
        y1=S(by, (double)0);
        z1=S(bz, (double)0);

        x2=S(bx, u/(double)2.0);
        y2=S(by, u/(double)2.0);
        z2=S(bz, u/(double)2.0);

        L1=(double)0;

        do
        {
            L0=L1;
            sum=(double)0;
            uu=(double)0;
            while(uu+du<u)
            {
                    x1=S(bx, uu);
                    y1=S(by, uu);
                    z1=S(bz, uu);

                    x2=S(bx, uu+du);
                    y2=S(by, uu+du);
                    z2=S(bz, uu+du);

                    sum=sum+sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+(z2-z1)*(z2-z1));
                    uu+=du;
            }
            L1=sum;
            du/=(double)2.0;
        }while(fabs(L1-L0)>eps);
        return L1;
}
```

(ALG 6-39)

o) A função ugl

Esta função fornece o valor do parâmetro u para um dado comprimento sobre a curva e uma dada precisão eps. O código é:

```
double B_Spline_cl::ugl(double L, double eps)
{
        double low=(double)0.0, high=(double)1.0, mid, LL;
        do
        {
            mid=(low+high)/(double)2.0;
            LL=length(mid, eps);
            if(LL<L)low=mid;
            else    high=mid;
        }while(fabs(L-LL)>eps);
        return mid;
}
```

(ALG 6-40)

p) A função InsertKnot

Esta função, proposta por PIEGL; TYLER (1997), insere um nó u no vetor nó e recalcula os novos pontos de controle de forma que a curva mantenha-se inalterada. O código é:

```
int B_Spline_cl::InsertKnot(double u)
{
        int i, j;
        double lambda, x[maxKnots], y[maxKnots], z[maxKnots];

        i=FindSpan(u);
        nn++;
        n0++;

        for(j=1; j<=i-k0+1; j++)
        {
            x[j]=bx[j];
            y[j]=by[j];
            z[j]=bz[j];
        }

        for (j=i-k0+2; j<=i; j++)
        {
            lambda=(u-knots[j])/(knots[j+k0-1]-knots[j]);

            x[j]=lambda*bx[j]+(1-lambda)*bx[j-1];
            y[j]=lambda*by[j]+(1-lambda)*by[j-1];
```

```
            z[j]=lambda*bz[j]+(1-lambda)*bz[j-1];

        }
        for (j=i+1; j<=n0; j++)
        {
            x[j]=bx[j-1];
            y[j]=by[j-1];
            z[j]=bz[j-1];
        }
        for (j=i-k0+2; j<=n0; j++)
        {
            bx[j]=x[j];
            by[j]=y[j];
            bz[j]=z[j];
        }
        for (j=nn; j>i+1; j--) knots[j]=knots[j-1];
        knots[i+1]=u;

        return (i+1);
}
```

(ALG 6-41)

q) A função RemoveKnot

Esta função, também proposta por PIEGL; TYLER (1997), verifica se é possível remover o nó u e, se possível, remove-o e recalcula os pontos de controle de forma que a curva se mantenha inalterada. O código é:

```
int B_Spline_cl::RemoveKnot(int r)
{
        int fout,
            off,
            last,
            first,
            k,
            i,
            ii,
            j,
            jj,
            remflag;
        double tempx[maxKnots],
               tempy[maxKnots],
               tempz[maxKnots],
               alfi,
               alfj;

        fout=(2*r-1-k0+1)/2;
        last=r-1;
        first=r-k0+1;

        //
        off=first-1;
        tempx[0]=bx[off];
        tempy[0]=by[off];
        tempz[0]=bz[off];

        tempx[last+1-off]=bx[last+1];
```

```
                for(k=r+1; k<=nn; k++) knots[k-1]=knots[k];
                j=fout;
                i=j;
                for (k=i+1; k<=n0; k++)
                {
                        bx[j]=bx[k];
                        by[j]=by[k];
                        bz[j]=bz[k];
                        j++;
                }
                n0--;
                nn--;
                return 1;
}
```

(ALG 6-42)

r) A função UpgradeKnots

Esta função insere q nós igualmente espaçados no vetor nó, e, em seguida, remove os nós pré-existentes que puderem ser removidos. O código é:

```
int B_Spline_cl::UpgradeKnots(int q)
{
                int i, j, nn0, nnn;
                double kn[maxKnots],
                        bbx[maxCP],
                        bby[maxCP],
                        bbz[maxCP],
                        step, u;
// salva nos existentes
                nn0=n0;
                nnn=nn;
                for (i=1; i<=nn; i++) kn[i]=knots[i];
// salva pontos de controle
                for (i=1; i<=n0; i++)
                {
                        bbx[i]=bx[i];
                        bby[i]=by[i];
                        bbz[i]=bz[i];
                }
// insere nos
                step=(double)1.0/(double)(q+1);
                u=step;
                while(u<1.0)
                {
                        InsertKnot(u);
                        u+=step;
                }

// remove nos existentes anteriormente

                for (i=k0+1; i<=nn0; i++)
                {
                        j=RemoveKnot(FindSpan(kn[i])+1);
                        if (!j)
                        {
```

```
                    ads_printf("\nnao foi possivel remover no: %d: %f\n", i, kn[i]);
                    //restore knots + CP
                }
            }
            return q;
}
```

(ALG 6-43)

s) A função KnotExists

Esta função verifica se o nó u pertence ao vetor nó. Caso exista retorna o índice no vetor nó, caso contrário, retorna zero. O código é:

```
int B_Spline_cl::KnotExists(double u)
{
            int p=0, k;
            for (k=1; k<=nn; k++) if (knots[k]==u) p=k;
            return p;
}
```

(ALG 6-40)

6.5 - A classe B_Surf

A implementação computacional para gerar uma superfície B-spline uniforme na direção v a partir de um conjunto de curvas B-spline uniformes ou não-uniformes é feita com auxílio da classe B_Surf apresentada a seguir:

```
class B_Surf{
public:
        double bxx[maxCP][11], byy[maxCP][11], bzz[maxCP][11];
        int n0, m0, k0, l0;
        int knotsu[maxKnots], knotsv[maxKnots];
        int knum, knvm , nnu, nnv;
        B_Surf();
        B_Surf(int n, int l, B_Spline s[]);
        B_Surf(int n, int l, B_Spline_cl s[]);
        void build(int n, int l, B_Spline s[]);
        void build(int n, int l, B_Spline_cl s[]);

        ~B_Surf();

        double SS(double b[][11], double u, double v);
        int knot(int v[], int n, int k);
        void bsp(double N[][10], int v[], int k, int ip, double t);
        double V(double z, double u);
        double Vr(double rr, double u);

private:
        static const double eps;
        double fi(double z, double u, double v);
        double fir(double rr, double u, double v);
        B_Spline spl;
};
```

(ALG 6-45)

As propriedades-membro principais desta classe são:

n0 - número de pontos de controle na direção u

m0 - número de pontos de controle na direção v

k0 - ordem na direção u

l0 - ordem na direção v

knotsu[] - vetor nó na direção u

knotsv[] - vetor nós na direção v

knum - valor máximo do vetor nó na direção u

knvm - valor máximo do vetor nó na direção v

nnu - número de nós na direção u

nnv - número de nós na direção v

bxx[], byy[], bzz[] - pontos de controle

a) O construtor padrão: B_Surf()

O construtor padrão serve para declarar uma instância da classe e alocar memória para a sua utilização. Nenhum cálculo é efetuado. O código é:

```
B_Surf::B_Surf()
{
}
```

(ALG 6-46)

b) O construtor com curvas uniformes: B_Surf(int, int, B_Spline)

Este construtor possui uma assinatura com três parâmetros e permite criar uma superfície B-spline uniforme de grau l a partir de n B-splines uniformes s[]. O código é:

```
B_Surf::B_Surf(int n, int l, B_Spline s[])
{
        build(n, l, s);
}
```

(ALG 6-47)

c) construtor com curvas não-uniformes: B_Surf(int, int, B_Spline_cl)

Este construtor possui uma assinatura com três parâmetros e permite criar uma superfície B-spline uniforme de grau l na direção v, a partir de n B-splines não-uniformes s[], cujos vetores nós tenham sido uniformizados. O código é:

```
B_Surf::B_Surf(int n, int l, B_Spline_cl s[])
{
        build(n, l, s);
}
```

(ALG 6-48)

d) O destrutor

O destrutor declarado para esta classe nada executa porque não há alocação dinâmica de memória e as instâncias desta classe declaradas no aplicativo são utilizadas até o fim da execução. O código do destrutor é:

```
B_Surf::~B_Surf()
{
}
```

(ALG 6-49)

e) A função build com B-splines uniformes

Esta função calcula os pontos de controle e vetor nó de uma superfície B-spline uniforme de grau l na direção v, a partir de n B-splines uniformes s[]. O código é:

```
void B_Surf::build(int n, int l, B_Spline s[])
{
        int i, j ;
        m0=n;
        l0=l;
        double x[12], y[12], z[12];

        k0=s[1].k0;
        n0=s[1].n0;

        knum=knot(knotsu, n0, k0);
        nnu=n0+k0;
        knvm=knot(knotsv, m0, l0);
        nnv=m0+l0;

        for(i=1; i<=n0;i++)
        {
            for (j=1;j<=m0;j++)
            {
                x[j]=s[j].bx[i];
                y[j]=s[j].by[i];
                z[j]=s[j].bz[i];
            }
            spl.adj(x,y,z,m0,l0);
            for(j=1;j<=m0;j++)
            {
                bxx[i][j]=spl.bx[j];
                byy[i][j]=spl.by[j];
                bzz[i][j]=spl.bz[j];
            }
        }
}
```

(ALG 6-50)

f) A função build com B-splines não-uniformes

Esta função calcula os pontos de controle e vetor nó de uma superfície B-spline uniforme de grau l na direção v, a partir de n B-splines não-uniformes s[], cujos vetores nó tenham sido uniformizados. O código é:

```
void B_Surf::build(int n, int l, B_Spline_cl s[])
{
        int i, j ;
        m0=n;
        l0=l;
        double x[12], y[12], z[12];

        k0=s[1].k0;
        n0=s[1].n0;

        knum=knot(knotsu, n0, k0);
        nnu=n0+k0;
        knvm=knot(knotsv, m0, l0);
        nnv=m0+l0;
```

```
        for(i=1; i<=n0;i++)
        {
                for (j=1;j<=m0;j++)
                {
                        x[j]=s[j].bx[i];
                        y[j]=s[j].by[i];
                        z[j]=s[j].bz[i];
                }
                spl.adj(x,y,z,m0,l0);
                for(j=1;j<=m0;j++)
                {
                        bxx[i][j]=spl.bx[j];
                        byy[i][j]=spl.by[j];
                        bzz[i][j]=spl.bz[j];
                }
        }
}
```

(ALG 6-51)

g) A função SS

Esta função calcula as coordenadas cartesianas de ponto no espaço dado pelos parâmetros u e v. O código é:

```
double B_Surf::SS(double b[][11], double u, double v)
{
        double s, N[10][10], M[10][10];
        int k, l, i, j;

 k=k0+1;
 l=l0+1;
 while (u> knotsu[k] ) k++;
 k--;
 while (v> knotsv[l] ) l++;
 l--;

 bsp(N,knotsu,k0,k,u);
 bsp(M,knotsv,l0,l,v);
 s=0;
 for (i=1; i<=k0; i++)
   for (j=1; j<=l0; j++)
                s=s+b[k-k0+i][l-l0+j]*M[j][l0]*N[i][k0];

 return s;

}
```

(ALG 6-52)

h) A função knot

Esta função calcula um vetor nó uniforme em função da quantidade de pontos de controle e da ordem. O código é:

```
int B_Surf::knot(int v[], int n, int k)
{
                int m, i;
                m=n-k+1;

  for (i= 1;i<=k;i++)    v[i] = 0;
  for (i=k+1;i<=n+1;i++) v[i] = v[i-1]+1;
  for (i=n+2;i<=n+k;i++) v[i] = v[i-1];
                return m;
}
```

(ALG 6-53)

i) A função bsp

Esta função retorna um vetor contendo as funções base não-nulas para um dado parâmetro u ou v. O código é:

```
void B_Surf::bsp(double N[][10], int v[], int k, int ip, double t)
{

 int i,j;
 double f;
 double dp[110], dm[110];


 N[1][1]=1;

 for (i=1; i<=k-1; i++)
 {

  dp[i]=v[ip+i]-t;
                dm[i]=t-v[ip+1-i];
                N[1][i+1]=0;
   for (j=1 ; j<=i; j++)
   {

                f           = N[j][i]/(dp[j] + dm[i+1-j]);
                N[j][i+1]   = N[j][i+1] + dp[j]*f;
                N[j+1][i+1] = dm[i+1-j]*f;
   }
 }
}
```

(ALG 6-54)

j) A função V

Esta função obtém o valor do parâmetro v da superfície para um dado valor de z e parâmetro u. O código é:

```
double B_Surf::V(double z, double u)
{
        double v1=0, v2=(double)knvm, r=1, vv;
        if(fi(z, u, v1)*fi(z, u, v2)>=0) return -1;
        else while(fabs(r)>eps)
        {
                vv=(v1+v2)/2;
                r=fi(z, u, vv);
                if (r*fi(z, u, v2)<0) v1=vv;
                else          v2=vv;
        }
        return vv;
}
```

(ALG 6-55)

l) A função Vr

Esta função calcula o valor do parâmetro v da superfície para um dado valor do raio e parâmetro u. O código é:

```
double B_Surf::Vr(double rr, double u)
{
        double v1=0, v2=(double)knvm, r=1, vv;
        if(fir(rr, u, v1)*fir(rr, u, v2)>=0) return -1;
        else while(fabs(r)>eps)
        {
                vv=(v1+v2)/2;
                r=fir(rr, u, vv);
                if (r*fir(rr, u, v2)<0) v1=vv;
                else                v2=vv;
        }
        return vv;
}
```

(ALG 6-56)

m) A função privada fi

Esta função retorna a diferença entre um valor de z fornecido e o calculado na superfície pelos parâmetros u e v dados. O código é:

```
double B_Surf::fi(double z, double u, double v)
{
        double r;
        r=z-SS(bzz, u, v);
        return r;
}
```

(ALG 6-57)

n) A função privada fir

Esta função retorna a diferença entre um valor de raio rr fornecido e o calculado na superfície pelos parâmetros u e v dados. O código é:

```
double B_Surf::fir(double rr, double u, double v)
{
        double r, y, z;
        y=SS(bzz, u, v);
        z=SS(byy, u, v);
        r=rr-sqrt(y*y + z*z);
        return r;
}
```

(ALG 6-58)

6.6 - A classe B_Surf_cl

A implementação computacional para gerar uma superfície B-spline não-uniforme na direção v a partir de um conjunto de curvas B-spline não-uniformes é feita com auxílio da classe B_Surf_cl apresentada a seguir:

```
class B_Surf_cl{
public:
        double bxx[maxCP][30], byy[maxCP][30], bzz[maxCP][30];
        int n0, m0, k0, l0;
        double knotsu[maxKnots], knotsv[30];
        double knum, knvm;
        int nnu, nnv;
        B_Surf_cl(); // must use parentheses
        B_Surf_cl(int n, int l, B_Spline_cl s[]);
        void build(int n, int l, B_Spline_cl s[]);
        void build(int ni, int n, int l, B_Spline_cl s[]);
        ~B_Surf_cl();

        double SS(double b[][30], double u, double v);

        void bsp(double N[][10], double v[], int k, int ip, double t);
        double V(double z, double u);
        double Vr(double rr, double u);

private:
        static const double eps;
        double fi(double z, double u, double v);
        double fir(double rr, double u, double v);
        B_Spline_cl spl, spla[maxKnots];
};
```

(ALG 6-59)

As propriedades-membro principais desta classe são:

n0 - número de pontos de controle na direção u

m0 - número de pontos de controle na direção v

k0 - ordem na direção u

l0 - ordem na direção v

knotsu[] - vetor nó na direção u

knotsv[] - vetor nós na direção v

knum - valor máximo do vetor nó na direção u

knvm - valor máximo do vetor nó na direção v

nnu - número de nós na direção u

nnv - número de nós na direção v

bxx[], byy[], bzz[] - pontos de controle

a) O construtor padrão: B_Surf_cl()

O construtor padrão serve para declarar uma instância da classe e alocar memória para a sua utilização. Nenhum cálculo é efetuado. O código é:

```
B_Surf_cl::B_Surf_cl()
{

}
```

(ALG 6-60)

b) O construtor adicional: B_Surf_cl(int, int, B_Spline_cl)

Este construtor possui uma assinatura com três parâmetros e permite criar uma superfície B-spline não-uniforme de grau l a partir de n B-splines não-uniformes s[]. O código é:

```
B_Surf_cl::B_Surf_cl(int n, int l, B_Spline_cl s[])
{
        build(n, l, s);
}
```

(ALG 6-61)

c) O construtor com curvas não-uniformes: B_Surf(int, int, B_Spline_cl)

Este construtor possui uma assinatura com três parâmetros e permite criar uma superfície B-spline uniforme de grau l na direção v, a partir de n B-splines não-uniformes s[], cujos vetores nós tenham sido uniformizados. O código é:

```
B_Surf::B_Surf(int n, int l, B_Spline_cl s[])
{
        build(n, l, s);
}
```

(ALG 6-62)

d) O destrutor

O destrutor declarado para esta classe nada executa porque não há alocação dinâmica de memória e as instâncias desta classe declaradas no aplicativo são utilizadas até o fim da execução. O código do destrutor é:

```
B_Surf_cl::~B_Surf_cl()
{
}
```

(ALG 6-63)

e) A função build sem uniformização de nós

Esta função calcula os pontos de controle e vetor nó de uma superfície B-spline não-uniforme de grau l na direção v, a partir de n B-splines não-uniformes s[]. O código é:

```
void B_Surf_cl::build(int n, int l, B_Spline_cl s[])
{
        int i, j ;
        m0=n;
        l0=l;
        double x[12], y[12], z[12];

        k0=s[1].k0;
        n0=s[1].n0;
        nnu=s[1].nn;

        knum=1.0;
        for(i=1; i<=nnu ; i++)knotsu[i]=s[1].knots[i];

        knvm=knot(knotsv, m0, l0);
        nnv=m0+l0;

        for(i=1; i<=n0; i++)
        {
            for (j=1;j<=m0;j++)
            {
                x[j]=s[j].bx[i];
                y[j]=s[j].by[i];
                z[j]=s[j].bz[i];
            }
            spl.adj(x, y, z, m0, l0);
            for(j=1;j<=m0;j++)
            {
                bxx[i][j]=spl.bx[j];
                byy[i][j]=spl.by[j];
                bzz[i][j]=spl.bz[j];
            }
        }
}
```

(ALG 6-64)

f) A função build com uniformização de nós

Esta função calcula os pontos de controle e vetor nó de uma superfície B-spline não-uniforme de grau l na direção v, a partir de n B-splines não-uniformes s[] efetuando a uniformização dos nós na direção v. O código é:

```
void B_Surf_cl::build(int ni, int n, int l, B_Spline_cl s[])
{
        int i, j, k ;
        m0=n;
        l0=l;
        double x[12], y[12], z[12];

        k0=s[1].k0;
        n0=s[1].n0;
        nnu=s[1].nn;

        knum=1.0;
```

```
                for(i=1; i<=nnu ; i++) knotsu[i]=s[1].knots[i];

                ads_printf("\nConstrucao do vetor no V uniforme");
                ads_printf("\n=====> Atualizando nos para a spline: ");
                for(i=1; i<=n0; i++)
                {
                        for (j=1; j<=m0; j++)
                        {
                                x[j]=s[j].bx[i];
                                y[j]=s[j].by[i];
                                z[j]=s[j].bz[i];
                        }
                        spla[i].adj(x, y, z, m0, l0);
                        ads_printf(" %d", i);
                        spla[i].UpgradeKnots(ni);
                }

// Incluir nos que nao puderam ser removidos de cada spline
// em todas as outras splines
for (i=1; i<=n0; i++)
                for (j=1; j<=n0; j++)
                        if (i != j)
                                for (k=1; k<=spla[j].nn; k++)
                                        if (!spla[i].KnotExists(spla[j].knots[k]))
                                                spla[i].InsertKnot(spla[j].knots[k]);
// salvar pontos de controle
                m0=spla[1].n0;
                for (i=1; i<=n0; i++)
                        for(j=1; j<=m0; j++)
                        {
                                bxx[i][j]=spla[i].bx[j];
                                byy[i][j]=spla[i].by[j];
                                bzz[i][j]=spla[i].bz[j];
                        }

// salvar vetor nor
                nnv=spla[1].nn;
                knvm=1.0;
                for (i=1; i<=nnv; i++) knotsv[i]=spla[1].knots[i];
}
```

(ALG 6-65)

g) A função SS

Esta função calcula as coordenadas cartesianas de ponto no espaço dado pelos parâmetros u e v. O código é:

```
double B_Surf_cl::SS(double b[][30], double u, double v)
{
                double s, N[10][10], M[10][10];
                int k, l, i, j;

 k=k0+1;
 l=l0+1;
 while (u> knotsu[k] ) k++;
 k--;
 while (v> knotsv[l] ) l++;
 l--;
```

```
bsp(N,knotsu,k0,k,u);
bsp(M,knotsv,l0,l,v);
s=0;
for (i=1; i<=k0; i++)
  for (j=1; j<=l0; j++)
                    s=s+b[k-k0+i][l-l0+j]*M[j][l0]*N[i][k0];

return s;

}
```

(ALG 6-66)

h) A função bsp

Esta função retorna um vetor contendo as funções base não-nulas para um dado parâmetro u ou v. O código é:

```
void B_Surf_cl::bsp(double N[][10], double v[], int k, int ip, double t)
{

 int i,j;
 double f;
 double dp[110], dm[110];


 N[1][1]=1;

 for (i=1; i<=k-1; i++)
 {

   dp[i]=v[ip+i]-t;
                dm[i]=t-v[ip+1-i];
                N[1][i+1]=0;
    for (j=1 ; j<=i; j++)
    {

                f           = N[j][i]/(dp[j] + dm[i+1-j]);
                N[j][i+1]   = N[j][i+1] + dp[j]*f;
                N[j+1][i+1] = dm[i+1-j]*f;
    }
 }
}
```

(ALG 6-67)

i) A função V

Esta função obtém o valor do parâmetro v da superfície para um dado valor de z e parâmetro u. O código é:

```
double B_Surf_cl::V(double z, double u)
{
        double v1=0, v2=(double)knvm, r=1, vv;
        if(fi(z, u, v1)*fi(z, u, v2)>=0) return -1;
        else while(fabs(r)>eps)
        {
            vv=(v1+v2)/2;
            r=fi(z, u, vv);
            if (r*fi(z, u, v2)<0) v1=vv;
            else          v2=vv;
        }
        return vv;
}
```

(ALG 6-68)

j) A função Vr

Esta função calcula o valor do parâmetro v da superfície para um dado valor do raio e parâmetro u. O código é:

```
double B_Surf_cl::Vr(double rr, double u)
{
        double v1=0, v2=(double)knvm, r=1, vv;
        if(fir(rr, u, v1)*fir(rr, u, v2)>=0) return -1;
        else while(fabs(r)>eps)
        {
            vv=(v1+v2)/2;
            r=fir(rr, u, vv);
            if (r*fir(rr, u, v2)<0) v1=vv;
            else              v2=vv;
        }
        return vv;
}
```

(ALG 6-69)

l) A função privada fi

Esta função retorna a diferença entre um valor de z fornecido e o calculado na superfície pelos parâmetros u e v dados. O código é:

```
double B_Surf_cl::fi(double z, double u, double v)
{
        double r;
        r=z-SS(bzz, u, v);
        return r;
}
```

(ALG 6-70)

m) A função privada fir

Esta função retorna a diferença entre um valor de raio rr fornecido e o calculado na superfície pelos parâmetros u e v dados. O código é:

```
double B_Surf_cl::fir(double rr, double u, double v)
{
        double r, y, z;
        y=SS(bzz, u, v);
        z=SS(byy, u, v);
        r=rr-sqrt(y*y + z*z);
        return r;
}
```

(ALG 6-71)

6.7 - A classe ARX: AcGePoint3d

Permite definir uma instância da entidade ponto definida através de três coordenadas (x,y,z).

Possui dois construtores:

- AcGePoint3d::AcGePoint3d() - Cria um ponto na origem (0,0,0).
- AcGePoint3d(double x, double y, double z) - cria um ponto com coordenadas (x, y, z)

Possui, ainda, o operador "[]" que permite acessar as coordenadas x, y ou z individualmente:

- double operator[](unsigned int i) const

O valor do parâmetro i acessa as coordenadas da seguinte forma:

- i=0 acessa x;
- i=1 acessa y;
- i=2 acessa z.

6.8 - A classe ARX: AcGePoint3dArray

Esta classe permite criar um vetor dinâmico de pontos acGePoint3d.

Possui um construtor que aloca um comprimento inicial igual a zero, dado pela seguinte declaração:

- AcGePoint3dArray(int initialPhysicalLength = 0, int initialGrowLength = 8)

É possível especificar o comprimento do vetor durante a execução com auxílio da função-membro setLogicalLength:

- AcGePoint3dArray& setLogicalLength(int newLogicalLength)

A função-membro SetAt permite armazenar um ponto na posição index:

- AcGePoint3dArray& setAt(int index, const AcGePoint3d& value)

A função-membro length permite obter o comprimento do vetor num dado momento:

- AcGePoint3dArray::length().

Adicionalmente é possível acessar as coordenadas (x, y, z) do ponto na posição index com auxílio do operador "[]"

- const AcGePoint3d operator[](int index) const

6.9 - A classe ARX: AcDb3dPolyline

Esta classe permite criar no banco de dados do desenho uma entidade do tipo 3DPOLY, i. e. , uma polilinha no espaço. Possui um construtor, dado pela seguinte delaração:

- AcDb3dPolyline::AcDb3dPolyline()

6.10 - A classe ARX: AcDbDatabase

Esta classe representa o banco de dados completo de um desenho. Possui o seguinte construtor padrão:

- AcDbDatabase ::AcDbDatabase(Adesk::Boolean buildDefaultDrawing = Adesk::kTrue)

6.11 - A classe ARX: AcDbObjectId

Esta classe permite armazenar a identidade de uma entidade no banco de dados do desenho. O construtor padrão e:

- AcDbObjectId::AcDbObjectId()

6.12 - A classe ARX: AcGeVector3d

Esta classe permite representar um vetor no espaço tridimensional. O construtor abaixo define um vetor a partir das coordenadas x, y e z:

- AcGeVector3d ::AcGeVector3d(double x, double y, double z)

6.13 - As 10 funções sugeridas na biblioteca ARX

As funções que se seguem foram obtidas em OBJECTARX Developer's Guide... (1997), OBJECTARX Reference Manual... (1997) e ADSARX Developer's Guide... (1997), algumas delas adaptadas com algumas correções e modificações.

a) A função selectEntity

Permite selecionar uma entidade no editor gráfico do AutoCAD, retornando um pointer para ela e o respectivo número de identificação. O código é:

```
// selectEntity - utility function to select entity
//
// Select entity, retrieve object Id and open it,
// creating and returning a valid object pointer.
//
AcDbEntity* selectEntity(AcDbObjectId& eId, AcDb::OpenMode openMode)
{

   ads_name en;
   ads_point pt;
   int rc = ads_entsel("\nSelect an entity: ", en, pt);
   if (rc != RTNORM) {
      ads_printf("Nothing selected.\n", rc);
      return NULL;
   }
     Acad::ErrorStatus es = acdbGetObjectId(eId, en);
   if (es != Acad::eOk) {
      ads_printf("Either acdbGetObjectId or ads_entsel failed: "
         "Entity name <%lx,%lx>, error %d.\n", en[0], en[1], es);
      return NULL;
   }
   AcDbEntity* entObj;
   es = acdbOpenObject(entObj, eId, openMode);
   if (es != Acad::eOk) {
      ads_printf("acdbOpenObject failed with error %d.\n", es);
   return NULL;
   }
   return entObj;
}
```

(ALG 6-72)

b) A função getZucs

Obtém a normal do sistema de referência. O código é:

```
// getZucs - retrieve UCS Z coordinate:
//
void getZucs(AcDbDatabase * pDb, AcGeVector3d& v)
{
   AcGeVector3d x = pDb->pucsxdir();
   AcGeVector3d y = pDb->pucsydir();
   v = x.crossProduct(y);
   v.normalize();
}
```

(ALG 6-73)

c) A função getPoint(char*, AcGePoint3d&)

Lê um ponto fornecido pelo usuário através do editor gráfico. O código é:

```
int getPoint(char* pMessage, AcGePoint3d& pT)
{
   int rc;
   ads_point pTo;

   if ((rc = ads_getpoint(NULL, pMessage, pTo)) == RTNORM) {
      rc = sa_u2w(pTo, pTo);
      pT[X] = pTo[X];
      pT[Y] = pTo[Y];
      pT[Z] = pTo[Z];
   }
   return rc;
}
```

(ALG 6-74)

d) A função getPoint(AcGePoint3d&, char*, AcGePoint3d&)

Esta função é uma variação polimórfica da anterior e permite ler um ponto fornecido pelo usuário no editor gráfico a partir de um ponto existente. O código é:

```
// getPoint - get WCS point from user:
//
int getPoint(AcGePoint3d& pF, char* pMessage, AcGePoint3d& pT)
{
   int rc;
   ads_point pFrom = {pF.x, pF.y, pF.z},
         pTo;

   if ((rc = sa_w2u(pFrom, pFrom)) == RTNORM
      && (rc = ads_getpoint(pFrom, pMessage, pTo)) == RTNORM)
   {
      rc = sa_u2w(pTo, pTo);
      pT.x = pTo[X];
      pT.y = pTo[Y];
      pT.z = pTo[Z];
   }
   return rc;
}
```

(ALG 6-75)

e) A função sa_u2w

Converte coordenadas locais para globais. O código é:

```
int sa_u2w(ads_point p1, ads_point p2)
{
   struct resbuf wcs,ucs;

   wcs.restype    = RTSHORT;
   wcs.resval.rint = 0;
   ucs.restype    = RTSHORT;
   ucs.resval.rint = 1;

   return ads_trans(p1, &ucs, &wcs, 0, p2);
}
```

(ALG 6-76)

f) A função sa_w2u

Converte coordenadas globais para locais. O código é:

```
int sa_w2u(ads_point p1, ads_point p2)
{
   struct resbuf wcs,ucs;

   wcs.restype    = RTSHORT;
   wcs.resval.rint = 0;
   ucs.restype    = RTSHORT;
   ucs.resval.rint = 1;

   return ads_trans(p1, &wcs, &ucs, 0, p2);
}
```

(ALG 6-77)

g) A função initApp

Inicializa o aplicativo, declarando os nomes das funções que poderão ser acesadas pelo usuário, assim como os seus apelidos. Também permite a criação inicial dos layers principais.

```
void initApp()
{

   acedRegCmds->addCommand("X_CMDS", "HELIXSURF", "HS",
      ACRX_CMD_MODAL, HELIXSURF);
   acedRegCmds->addCommand("X_CMDS", "EdCurv", "Ed",
      ACRX_CMD_MODAL, EdCurv);
            acedRegCmds->addCommand("X_CMDS", "bsplab", "bs",
      ACRX_CMD_MODAL, bsplab);
            acedRegCmds->addCommand("X_CMDS", "KAPLAN", "KA",
      ACRX_CMD_MODAL, KAPLAN);
            acedRegCmds->addCommand("X_CMDS", "bsplab_cl", "bc",
      ACRX_CMD_MODAL, bsplab_cl);


            char la[20], st[20];

// criar todos os layers
            for (int i=1; i<=9; i++)
            {
                  _itoa(i, st, 10);
// das  splines
                  strcpy(la, st);
                  strcat(la, "bsp");
                  addLayer(la, 2);
// das splines nao-uniformes
```

```
                    strcpy(la, st);
                    strcat(la, "nubsp");
                    addLayer(la, 3);
//dos pontos de controle
                    strcpy(la, st);
                    strcat(la, "control");
                    addLayer(la, 4);
// dos pontos de controle nao-uniformes
                    strcpy(la, st);
                    strcat(la, "nucontrol");
                    addLayer(la, 6);
// das vistas planas
                    strcpy(la, st);
                    strcat(la, "plan");
                    addLayer(la, 7);
//dos perfis no espaco
                    strcpy(la, st);
                    strcat(la, "prof");
                    addLayer(la, 4);
                }
//para verificacao dos perfis radiais
                addLayer("rad", 2);

//para os pontos de controle da superficie
                addLayer("long", 6);

//para a base dos moldes
                addLayer("base", 7);
}
```

(ALG 6-78)

h) A função unloadApp

Descarrega o aplicativo da memória finalizando a sua execução.

```
void unloadApp()
{
  acedRegCmds->removeGroup("X_CMDS");
}
```

(ALG 6-79)

i) A função acrxEntryPoint

É o ponto de entrada da DLL ("dynamic link library"), através do qual o AutoCad interage com o aplicativo.

```
extern "C" AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
   switch (msg) {
   case AcRx::kInitAppMsg:
      acrxDynamicLinker->unlockApplication(pkt);
      initApp();
      break;
   case AcRx::kUnloadAppMsg:
      unloadApp();
   }
   return AcRx::kRetOK;
}
```

(ALG 6-80)

j) A função iterate

Esta função percorre os vértices de uma polilinha ("polyline") e os imprime na tela. É a base da construção da função bvl mostrada mais adiante. O código é:

```
// Accepts the objectId of an AcDb2dPolyline, opens it, gets
// a vertex iterator, and then iterates through the vertices
// printing out the vertex location
void iterate(AcDbObjectId plineId)
{
   AcDb3dPolyline *pPline;
   acdbOpenObject(pPline, plineId, AcDb::kForRead);

   AcDbObjectIterator *pVertIter= pPline->vertexIterator();
   pPline->close();  // Finished with the pline header.

   AcDb3dPolylineVertex *pVertex;
   AcGePoint3d location;

   AcDbObjectId vertexObjId;
   for (int vertexNumber = 0; !pVertIter->done();
      vertexNumber++, pVertIter->step())
   {
      vertexObjId = pVertIter->objectId();
      acdbOpenObject(pVertex, vertexObjId,
         AcDb::kForRead);
      location = pVertex->position();
      pVertex->close();

      ads_printf("\nVertex #%d's location is"
         " : %0.3f, %0.3f, %0.3f", vertexNumber,
         location[X], location[Y], location[Z]);

   }
   delete pVertIter;
}
```

(ALG 6-81)

6.14 - As 24 funções desenvolvidas no aplicativo

As funções descritas a seguir permitem utilizar de forma efetiva as classes e funções descritas acima, porque garantem a inicialização correta das mesmas, assim como, a seqüência correta de chamada de cada uma delas.

a) A função HELIXSURF

Esta função prepara o ambiente para a criação de uma pá, abrindo 4 janelas: uma para cada eixo principal do sistema global de referência e uma para uma vista isométrica. Em seguida, chama a rotina de criação da pá a partir da série sistemática. O código é:

```
void HELIXSURF(void)
{
        int c;
        char done=0;
        ads_command(RTSTR, "_vports", RTSTR, "4", RTNONE);

        ads_command(RTSTR, "_setvar", RTSTR, "CVPORT", RTSTR, "2", RTNONE);
        ads_command(RTSTR, "_vpoint", RTSTR, "0,0,1", RTNONE);

        ads_command(RTSTR, "_setvar", RTSTR, "CVPORT", RTSTR, "3", RTNONE);
        ads_command(RTSTR, "_vpoint", RTSTR, "-1,0,0", RTNONE);

        ads_command(RTSTR, "_setvar", RTSTR, "CVPORT", RTSTR, "4", RTNONE);
        ads_command(RTSTR, "_vpoint", RTSTR, "-1,-1,1", RTNONE);

        ads_command(RTSTR, "_setvar", RTSTR, "CVPORT", RTSTR, "5", RTNONE);
        ads_command(RTSTR, "_vpoint", RTSTR, "0,-1,0", RTNONE);


while (!done)
{
        ads_printf("\nEscolha a serie:: <1-Kaplan 2-B-Troost 3-Sair>: ");
        ads_getint("",&c);
c=1;
        switch (c)
        {
        case 1:
                KAPLAN();
                BLADE=1; //sets blade flag
```

```
                done=1;
            break;

            case 2:
                B_TROOST();
                BLADE=1;
                done=1;
            break;

            case 3:
                done=1;
            break;

            default:
                ads_printf("\nEscolha invalida Selecione: 1, 2 ou 3!");

} //switch
} //while
}
```

(ALG 6-82)

b) A função KAPLAN

Esta função solicita os dados do hélice, tais como: diâmetro, número de pás, razão entre área expandida e área projetada, razão entre passo e diâmetro e precisão de cálculo. Em seguida, efetua os cálculos da série sistemática, desenha os perfis no espaço e no plano expandido, aproxima B-splines não uniformes nos perfis, uniformiza os vetores nó e constrói uma superfície B-spline não uniforme. Por último, percorre cada janela ajustando o zoom de forma que toda a extensão do desenho seja visível. O código é:

```
void KAPLAN(void)
{
            int i, j, k, /* nc, */ done, np, c ;
            char la[20], st[20];
            AcGePoint3d pt;
            AcGePoint3d p[maxCP];
  AcGePoint3dArray pta;
  AcDbObjectId eId;

  ads_initget(0, "KAPLAN");
  AcDbDatabase *pDb = acdbCurDwg();
  AcGeVector3d normal;
  getZucs(pDb, normal);

// precisao
            double eps;
            double dd, zz, aa, pd;
            done=0;
```

```
                do{
                  ads_getstring(0, "\nDiametro: <500>", st);
                  if (strlen(st)==0) strcpy(st, "500");
                  dd = atof(st);
    if (dd > 0) done=1;
                  else ads_printf("\nValor invalido!");
                }while (!done);

                done=0;
                do{
                  ads_getstring(0,"\nNumero de pas: <4>", st);
                  if (strlen(st)==0) strcpy(st, "4");
                  zz = atof(st);
    if (zz>0) done=1;
                  else ads_printf("\nValor invalido!");
                }while (!done);

                done=0;
                do{
                  ads_getstring(0,"\nAe/A0: <0.6>", st);
                  if (strlen(st)==0) strcpy(st, "0.6");
                  aa=atof(st);
    if (aa>0) done=1;
                  else ads_printf("\nValor invalido!");
                }while (!done);

                done=0;
                do{
                  ads_getstring(0,"\nP/D: <0.9>", st);
                  if (strlen(st)==0) strcpy(st, "0.9");
                  pd=atof(st);
    if (pd>0) done=1;
                  else ads_printf("\nValor invalido!");
                }while (!done);

                done=0;
                do{
                  ads_getstring(0,"\nPrecisao: <0.01>", st);
                  if (strlen(st)==0) strcpy(st,"0.01");
                  eps=atof(st);
    if (eps>0) done=1;
                  else ads_printf("\nValor invalido!");
                }while (!done);

                ka.build((double)dd, (double)zz , (double)aa, (double)pd);

                ads_printf("\nPa KAPLAN construida:\nd= %f z= %f aea0= %f p/d= %f" ,
                        ka.d, ka.z, ka.aea0, ka.pd);

 pta.setLogicalLength(ka.bl[1].np);


for (k=1; k<=9; k++)
{
                for (i=0; i<=5 ; i++)
                {
                        p[i][0]=ka.bl[k].xte[i+1];
                        p[i][1]=ka.bl[k].yfte[i+1];
                }
                for (i=0; i<=6 ; i++)
                {
                        p[i+6][0]=ka.bl[k].xle[i+1];
                        p[i+6][1]=ka.bl[k].yfle[i+1];
                }
                for (i=5; i>=0 ; i--)
                {
                        p[18-i][0]=ka.bl[k].xle[i+1];
                        p[18-i][1]=ka.bl[k].yble[i+1];
                }
                for (i=5; i>=0 ; i--)
```

```
                {
                        p[24-i][0]=ka.bl[k].xte[i+1];
                        p[24-i][1]=ka.bl[k].ybte[i+1];
                }

                for (i=0; i<ka.bl[1].np;  i++) pta.setAt(i,p[i]);
// vista plana
                _itoa(k, st, 10);
                strcpy(la, st);
                strcat(la, "plan");
                DrawPoly(pta,la);
}
// pontos no espaco

                for (k=1; k<=9; k++)
                {
                        for (i=0; i<ka.bl[k].np ; i++)
                        {
                                p[i][0]=ka.bl[k].xe[i+1];
                                p[i][1]=ka.bl[k].ye[i+1];
                                p[i][2]=ka.bl[k].ze[i+1];
                                pta.setAt(i,p[i]);
                        }
                        _itoa(k, st, 10);
                        strcpy(la, st);
                        strcat(la, "prof");
                        DrawPoly(pta,la);
                }

                ads_printf("\nAproximando B-splines nao-uniformes");
                for (i=1;i<=9;i++)
                {
                        ads_printf("\nAproximaando Spline: %d", i);
                        sp_cl[i].adj(ka.bl[i].xe, ka.bl[i].ye,
                                        ka.bl[i].ze, 25, 4);
                }

//desenha splines no espaco
                for (i=1; i<=9; i++)
                {
                        _itoa(i, st, 10);
                        strcpy(la, st);
                        strcat(la, "nubsp");
                        dsp(sp_cl[i], la);
                }

// desenha pontos de controle nao uniformes no espaco
                for (i=1; i<=9; i++)
                {
                        _itoa(i, st, 10);
                        strcpy(la, st);
                        strcat(la, "nucontrol");
                        dcp(sp_cl[i],la);
                }
// impoe mesmo vetor no para todas splines nao uniformes
                c=25;
                for (i=1; i<=9; i++)
                {
                        ads_printf("\nAtualizando nos da spline: %d", i);
                        np=sp_cl[i].UpgradeKnots(c);
                }
// inclui nos que nao puderam ser removidos de cada spline
// em todas as demais
for (i=1; i<=9; i++)
                for (j=1; j<=9; j++)
                        if (i != j)
                                for (k=1; k<=sp_cl[j].nn; k++)
                                        if (!sp_cl[i].KnotExists(sp_cl[j].knots[k]))

                                                sp_cl[i].InsertKnot(sp_cl[j].knots[k]);
```

```
// imprime vetores no

for (i=1; i<=9; i++)
{
        ads_printf("\nNos da spline: %d \n", i);
        for(j=1; j<=sp_cl[i].nn; j++)
            ads_printf("[%d] %1.3f ", j, sp_cl[i].knots[j]);
}
// desenha splines atualizadas no espaco
        for (i=1; i<=9; i++)
        {
            _itoa(i, st, 10);
            strcpy(la, st);
            strcat(la, "bsp");
            dsp(sp_cl[i], la);
        }

// desenha os novos pontos de controle
        for (i=1; i<=9; i++)
        {
            _itoa(i, st, 10);
            strcpy(la, st);
            strcat(la, "control");
            dcp(sp_cl[i],la);
        }

// cria superficie

// nao uniforme
        bls_cl.build(9, 9, 4, sp_cl);
        ads_printf("\nSuperficie construida...");



        ads_printf("\nn0= %d  k0= %d m0= %d l0= %d",
              bls_cl.n0, bls_cl.k0, bls_cl.m0, bls_cl.l0);
// desenha pontos de controle da superficie
            pta.setLogicalLength(bls_cl.m0);
        for (i=1; i<=bls_cl.n0; i++)
        {
            for(k=1 ; k<=bls_cl.m0; k++)
            {
                pt[0]=bls_cl.bxx[i][k];
                pt[1]=bls_cl.byy[i][k];
                pt[2]=bls_cl.bzz[i][k];
                pta.setAt(k-1, pt);
            }
            DrawPoly(pta, "long");
        }

// zoom  extents em todas viewports

        ads_command(RTSTR, "_setvar", RTSTR, "CVPORT", RTSTR, "2", RTNONE);
        ads_command(RTSTR, "_zoom", RTSTR, "e", RTNONE);

        ads_command(RTSTR, "_setvar", RTSTR, "CVPORT", RTSTR, "3", RTNONE);
        ads_command(RTSTR, "_zoom", RTSTR, "e", RTNONE);

        ads_command(RTSTR, "_setvar", RTSTR, "CVPORT", RTSTR, "4", RTNONE);
        ads_command(RTSTR, "_zoom", RTSTR, "e", RTNONE);

        ads_command(RTSTR, "_setvar", RTSTR, "CVPORT", RTSTR, "5", RTNONE);
        ads_command(RTSTR, "_zoom", RTSTR, "e", RTNONE);
}
```

(ALG 6-83)

c) A função bsplab

Esta função permite criar e manipular uma B-spline uniforme no editor gráfico do AutoCAD. Através dela foi possível fazer os diversos testes deste trabalho que proporcionaram o entendimento deste tipo de B-spline. Esta função possui os seguintes subcomandos:

1-RCP : "Read Control Points" - seleciona uma polilinha e considera os seus vértices como sendo os pontos de controle da B-spline;

2- RXYZ : "Read XYZ" - seleciona uma polilinha e considera os seus vértices pontos de entrada para aproximação de uma B-spline.;

3- Dsp : "Draw spline" - desenha a spline;

4- DCP : "Draw Control Points" - desenha os pontos de controle;

5- Dknv : "Draw Knot vector" - desenha o vetor nó;

6 - Rknv : "Read knot vector" - seleciona uma polilinha e considera as coordenadas x dos seus vértices elementos do vetor nó.

7- ADS : área de testes de comandos do AutoCAD;

8- Rxyza : "Read x,y,z and adjust" - Seleciona uma polilinha e considera seus vértices pontos de entrada para interpolação de uma B-spline;

9 - SmK : "Smooth curve at Knot" - suaviza a curva num determinado nó;

10 - Exit : Sai do comando.

O código é:

```
void bsplab(void)
{
int np, i, c, nc;
char la[20];
double x[100], y[100], z[100];

AcDbEntity *pObj;
AcDbObjectId eId;
AcGePoint3d vl[100];
AcGePoint3dArray pta;

while (c!=10)
{
ads_printf("\nSelect: <1-RCP 2-RXYZ 3-Dsp 4-DCP 5-Dknv 6-Rknv 7-ADS 8-Rxyza 9-SmK 10-Exit>: ");
ads_getint("",&c);
switch (c)
{

case 1:
                pObj = selectEntity(eId, AcDb::kForRead);
                np=bvl(eId, vl);
  pta.setLogicalLength(np);
                pObj->close();
                for (i=0; i<np; i++)
                {
                        bs.bx[i+1]=(double)vl[i][0];
                        bs.by[i+1]=(double)vl[i][1];
                        bs.bz[i+1]=(double)vl[i][2];
                }
                bs.n0=np;
                bs.k0=4;
                bs.saveknots();
                strcpy(la, "0");
                dsp(bs, la);
                ads_printf("\nn0= %d", bs.n0);
                ads_printf("\nk0= %d", bs.k0);
break;

case 2:
pObj = selectEntity(eId, AcDb::kForRead);
                np=bvl(eId, vl);
  pta.setLogicalLength(np);
                pObj->close();
                for (i=0; i<np; i++)
                {
                        x[i+1]=(double)vl[i][0];
                        y[i+1]=(double)vl[i][1];
                        z[i+1]=(double)vl[i][2];
                }
                ads_printf("\nEnter number of control points: ");
                ads_getint("",&nc);
                bs.n0=nc;
                bs.k0=4;
                bs.saveknots();
                bs.aprox(x,y,z,np,nc,4);
                strcpy(la, "0");
                dsp(bs, la);
                ads_printf("\nn0= %d", bs.n0);
                ads_printf("\nk0= %d", bs.k0);
                ads_printf("\nnp= %d", np);
                dcp(bs, la);
break;

case 3:
                strcpy(la, "0");
                dsp(bs, la);
break;
```

```
case 4:
                strcpy(la, "0");
                dcp(bs, la);
break;

case 5:
                dknv(bs,"0");

break;

case 6:
                pObj = selectEntity(eId, AcDb::kForRead);
                np=bvl(eId, vl);
  pta.setLogicalLength(np);
                pObj->close();
                for (i=0; i<np; i++)
                {
                        bs.knots[i+1]=(int)vl[i][0];
                        ads_printf("\nknot[%d] = %d", i+1,bs.knots[i+1]);
                }
                bs.nn=np;
break;

case 8:
                pObj = selectEntity(eId, AcDb::kForRead);
                np=bvl(eId, vl);
  pta.setLogicalLength(np);
                pObj->close();
                for (i=0; i<np; i++)
                {
                        x[i+1]=(double)vl[i][0];
                        y[i+1]=(double)vl[i][1];
                        z[i+1]=(double)vl[i][2];
                }
                //ads_printf("\nEnter number of control points: ");
                //ads_getint("",&nc);
                nc=np;
                bs.n0=nc;
                bs.k0=4;
                bs.saveknots();
                bs.adj(x,y,z,np,4);
                strcpy(la, "0");
                dsp(bs, la);
                ads_printf("\nn0= %d", bs.n0);
                ads_printf("\nk0= %d", bs.k0);
                ads_printf("\nnp= %d", np);
                dcp(bs, la);

break;

case 9:
                int kn;
                ads_getint("\nEnter knot number: ", &kn);
                bs.SmoothKnot(kn);
                strcpy(la, "0");
                dcp(bs, la);
break;

case 7:

char st[30];
ads_getstring(0,"\nEnter layer name: ", st);
if (strcmp(st, "no")) ads_printf("\nFlag: %d", findLayer(st));
ads_getstring(0,"\nEnter layer name: ", st);
if (strcmp(st, "no")) addLayer(st,1);
break;
} //end switch
} //end while
}
```

(ALG 6-84)

d) A função bsp_cl

Esta função é análoga à função bsplab, porém funciona para B-splines não uniformes. Possui os seguintes subcomandos:

1-RCP : "Read Control Points" - seleciona uma polilinha e considera os seus vértices como sendo os pontos de controle da B-spline;

2- RXYZ : "Read XYZ" - seleciona uma polilinha e considera os seus vértices pontos de entrada para aproximação de uma B-spline.;

3- Dsp : "Draw spline" - desenha a spline;

4- DCP : "Draw Control Points" - desenha os pontos de controle;

5- Dknv : "Draw Knot vector" - desenha o vetor nó;

6 - Rknv : "Read knot vector" - seleciona uma polilinha e considera as coordenadas x dos seus vértices elementos do vetor nó.

7- ADS : área de testes de comandos do AutoCAD;

8- Rxyza : "Read x,y,z and adjust" - Seleciona uma polilinha e considera seus vértices pontos de entrada para interpolação de uma B-spline;

9 - SmK : "Smooth curve at Knot" - suaviza a curva num determinado nó;

10- BUS : "Build Uniform Spline" - Constrói uma B-spline uniforme a partir de pontos da existente obtidos por subdivisão do comprimento da curva. O resultado obtido por esse processo não é satisfatório;

11 - Ink : "Insert knot" - Insere um nó no vetor nó e recalcula os novos pontos de controle da curva;

12 - "ReK" : "Remove Knot" - Remove um nó, se possível, e recalcula os novos pontos de controle da curva;

13 - "UpK" : "Upgrade Knots" - permite incluir nós igualmente espaçados no vetor nó e remover os anteriores se possível. Este subcomando permitiu avaliar a quantiadde ótima de nós que seriam inseridos no processo de uniformização dos vetores nó;

14 - Exit : Sai do comando.

O código é:

```
void bsplab_cl(void)
{
int np, i, j, c, nc;
char la[20];
double x[100], y[100], z[100], step, u;

AcDbEntity *pObj;
AcDbObjectId eId;
AcGePoint3d vl[100], pt;
AcGePoint3dArray pta;

while (c!=14)
{
ads_printf("\nSelect: <1-RCP 2-RXYZ 3-Dsp 4-DCP 5-Dknv 6-Rknv 7-ADS 8-Rxyza 9-SmK 10-BUS 11-InK
        12-ReK 13-UpK 14-Exit>: ");
ads_getint("",&c);
switch (c)
{

case 1: //RCP
        pObj = selectEntity(eId, AcDb::kForRead);
        np=bvl(eId, vl);
  pta.setLogicalLength(np);
        pObj->close();
        for (i=0; i<np; i++)
        {
              bs_cl.bx[i+1]=vl[i][0];
              bs_cl.by[i+1]=vl[i][1];
              bs_cl.bz[i+1]=vl[i][2];
        }
        bs_cl.n0=np;
        bs_cl.k0=4;
        //bs.saveknots();
        strcpy(la, "0");
        dsp(bs_cl, la);
        ads_printf("\nn0= %d", bs.n0);
        ads_printf("\nk0= %d", bs.k0);
break;

case 2: //RXYZ
```

```
pObj = selectEntity(eId, AcDb::kForRead);
                np=bvl(eId, vl);
  pta.setLogicalLength(np);
                pObj->close();
                for (i=0; i<np; i++)
                {
                        x[i+1]=vl[i][0];
                        y[i+1]=vl[i][1];
                        z[i+1]=vl[i][2];
                }
                ads_getint("\nEnter number of control points: ", &nc);
                bs_cl.n0=nc;
                bs_cl.k0=4;
                bs_cl.saveknots(np, x, y, z);
                bs_cl.aprox(x, y, z, np, nc, 4);
                strcpy(la, "0");
                dsp(bs_cl, la);
                ads_printf("\nn0= %d", bs_cl.n0);
                ads_printf("\nk0= %d", bs_cl.k0);
                ads_printf("\nnp= %d", np);
                ads_printf("\n");
                for(i=1; i<=bs_cl.n0+bs_cl.k0; i++)
                        ads_printf(" %1.3f ", bs_cl.knots[i]);
                for(i=1; i<=bs_cl.n0; i++)
                        ads_printf("\n x=%f y=%f z=%f",
                                bs_cl.bx[i], bs_cl.by[i], bs_cl.bz[i]);
break;

case 3:  //Dsp
                strcpy(la, "0");
                dsp(bs_cl, la);
break;

case 4:  //DCP
                strcpy(la, "0");
                dcp(bs_cl, la);
break;

case 5:   //Dkn
                strcpy(la, "0");
                dknv(bs_cl, la);
                ads_printf("\nKnots: ");
                for(i=1; i<=bs_cl.n0+bs_cl.k0; i++)
                        ads_printf("[%d] %1.3f ", i, bs_cl.knots[i]);


break;

case 6:  //Rkn
                pObj = selectEntity(eId, AcDb::kForRead);
                np=bvl(eId, vl);
  pta.setLogicalLength(np);
                pObj->close();
                for (i=0; i<np; i++)
                {
                        bs_cl.knots[i+1]=vl[i][0];
                        ads_printf("\nknot[%d] = %d", i+1, bs_cl.knots[i+1]);
                }
                bs_cl.nn=np;
break;

case 8:
                pObj = selectEntity(eId, AcDb::kForRead);
                np=bvl(eId, vl);
  pta.setLogicalLength(np);
                pObj->close();
                for (i=0; i<np; i++)
                {
                        x[i+1]=vl[i][0];
                        y[i+1]=vl[i][1];
```

```
                z[i+1]=vl[i][2];
            }
            ads_printf("\n   x      y      z");

            for(i=1; i<=np; i++)
                ads_printf("\n %f %f %f", x[i], y[i], z[i]);

            nc=np;
            bs_cl.n0=nc;
            bs_cl.k0=4;
            bs_cl.saveknots(np, x, y, z);
            bs_cl.adj(x, y, z, np, 4);
            strcpy(la, "0");
            dsp(bs_cl, la);
            ads_printf("\nn0= %d", bs_cl.n0);
            ads_printf("\nk0= %d", bs_cl.k0);
            ads_printf("\nnp= %d", np);
            ads_printf("\nKnots: ");
            for(i=1; i<=bs_cl.n0+bs_cl.k0; i++)
                ads_printf(" %1.3f ", bs_cl.knots[i]);
            for(i=1; i<=bs_cl.n0; i++)
                ads_printf("\nB%d x=%f y=%f z=%f", i,
                        bs_cl.bx[i], bs_cl.by[i], bs_cl.bz[i]);
            ads_printf("\nLength=%f", bs_cl.length(1, (double)0.01));
            ads_printf("\nu=%f for length=9.0", bs_cl.ugl((double)9.0, (double)0.01));

break;

case 9: //SmK
            int kn;
            ads_getint("\nEnter knot number: ", &kn);
            bs_cl.SmoothKnot(kn);
            strcpy(la, "0");
            dcp(bs_cl, la);
break;

case 10:
            // build uniform spline;
            double LL;
            ads_getint("\nEnter number of control points: ", &nc);

            x[1]=bs_cl.bx[1];
            y[1]=bs_cl.by[1];
            z[1]=bs_cl.bz[1];

            x[nc]=bs_cl.bx[bs_cl.n0];
            y[nc]=bs_cl.by[bs_cl.n0];
            z[nc]=bs_cl.bz[bs_cl.n0];

            LL=bs_cl.length((double)1.0, (double)0.01);

            step=LL/(double)(nc-1);
            LL=step;
            for(i=2; i<nc; i++)
            {
                u=bs_cl.ugl(LL, (double)0.01);
                x[i]=bs_cl.S(bs_cl.bx, u);
                y[i]=bs_cl.S(bs_cl.by, u);
                z[i]=bs_cl.S(bs_cl.bz, u);
                LL+=step;
            }
            ads_printf("\nSelected points:");
            ads_printf("\n      x      y      z");
            for(i=1; i<=nc; i++)
                ads_printf("\n[%d] %f %f %f", i, x[i], y[i], z[i]);
                        pta.setLogicalLength(nc);
            ads_printf("\nDrawing selected points from non uniform spline: %d", i);
                for(i=1 ; i<=nc; i++)
                {
                        pt[0]=x[i];
```

```
                        pt[1]=y[i];
                        pt[2]=z[i];
                        pta.setAt(i-1,pt);
                    }
                    addLayer("1", 1);
                    strcpy(la, "1");
                    DrawPoly(pta, la);
    ubs.adj(x, y, z, nc, 4);

                ads_printf("\nControl points:");
                ads_printf("\n   bx      by      bz");
                for(i=1; i<=nc; i++)
                    ads_printf("\n[%d] %f %f %f", i, ubs.bx[i], ubs.by[i], ubs.bz[i]);
                    dcp(ubs, la);
                    dsp(ubs, la);


break;

case 11:  //Ink
                ads_getreal("\nEnter knot: ", &u);
                j=bs_cl.InsertKnot(u);

                for(i=1; i<=bs_cl.n0; i++)
                    ads_printf("\nB%d x=%f y=%f z=%f", i,
                            bs_cl.bx[i], bs_cl.by[i], bs_cl.bz[i]);

                ads_printf("\nKnots: ");
                for(i=1; i<=bs_cl.n0+bs_cl.k0; i++)
                    ads_printf("[%d] %1.3f ", i, bs_cl.knots[i]);
                ads_printf("\nKnot index : %d", j);
                // to find a knot index add 1 to its span
  ads_printf("\nKnot index : %d", bs_cl.FindSpan(u)+1);
                addLayer("1", 1);
                strcpy(la, "1");
                dcp(bs_cl, la);

break;

case 12:  //Rmk
                ads_getint("\nEnter knot index: ", &i);
                i=bs_cl.RemoveKnot(i);
                if(i)
                {
                    ads_printf("\nKnot removed successfuly");

                    strcpy(la, "1");
                    dcp(bs_cl, la);
                    ads_printf("\nKnots: ");
                    for(i=1; i<=bs_cl.n0+bs_cl.k0; i++)
                      ads_printf("[%d] %1.3f ", i, bs_cl.knots[i]);
                    ads_printf("\nKnots: %d  Control Points: %d",
                                bs_cl.nn, bs_cl.nn);

                }
                else  ads_printf("\nKnot could not be removed");

break;

case 13:
                ads_getint("\nEnter number of knots: ", &i);
                j=bs_cl.UpgradeKnots(i);
                if (j)
                {
                    dcp(bs_cl, "0");
                    dsp(bs_cl, "0");
                    for(i=1; i<=bs_cl.n0+bs_cl.k0; i++)
                      ads_printf("[%d] %1.3f ", i, bs_cl.knots[i]);
                    ads_printf("\nKnots: %d  Control Points: %d",
                                bs_cl.nn, bs_cl.n0);
```

```
            }
            else
            {
                ads_printf("\nCould not upgrade knots");
            }

break;

case 7:
//====================
char st[30];
ads_getstring(0,"\nEnter layer name: ", st);
if (strcmp(st, "no")) ads_printf("\nFlag: %d", findLayer(st));
ads_getstring(0,"\nEnter layer name: ", st);
if (strcmp(st, "no")) addLayer(st,1);

//====================
break;

} //end switch
} //end while
}
```

(ALG 6-85)

e) A função bvl

Esta função tem o papel fundamental de permitir a interação do usuário com o aplicativo através da interface gráfica do AutoCAD. Constrói um vetor de pontos a partir dos vértices de uma polilinha. O código é:

```
int bvl(AcDbObjectId plineId, AcGePoint3d vl[])
{
  AcDb3dPolyline *pPline;
  acdbOpenObject(pPline, plineId, AcDb::kForRead);

  AcDbObjectIterator *pVertIter= pPline->vertexIterator();
  pPline->close();

  AcDb3dPolylineVertex *pVertex;

  AcDbObjectId vertexObjId;
  for (int vertexNumber = 0; !pVertIter->done();
    vertexNumber++, pVertIter->step())
  {
    vertexObjId = pVertIter->objectId();
    acdbOpenObject(pVertex, vertexObjId,
      AcDb::kForRead);
    vl[vertexNumber] = pVertex->position();
    pVertex->close();
  }
 return vertexNumber;

}
```

(ALG 6-86)

f) A função DrawPoly

Esta função desenha uma polilinha ("polyline") a partir de um vetor de pontos. O código é:

```
void DrawPoly(AcGePoint3dArray ptArr, char *la)
{
  AcDb3dPolyline *pNewPline =
                    new AcDb3dPolyline(AcDb::k3dSimplePoly, ptArr, Adesk::kFalse);

AcDbBlockTable *pBlockTable;
  acdbCurDwg()->getBlockTable(pBlockTable, AcDb::kForRead);

  AcDbBlockTableRecord *pBlockTableRecord;
  pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
    AcDb::kForWrite);
  pBlockTable->close();

AcDbObjectId plineObjId;
  pBlockTableRecord->appendAcDbEntity(plineObjId, pNewPline);
  pBlockTableRecord->close();

pNewPline->setLayer(la);


  pNewPline->close();
}
```

(ALG 6-87)

g) A função edCurv

Esta função é a mais utilizada após a criação da pá. Com ela é possível modificar as curvas que interpolam os perfis no espaço, com auxílio do editor gráfico do AutoCAD, reconstruir a superfície, obter seções planas para os moldes e gerar os scripts de plotagem automática. Possui os seguintes subcomandos:

1-Rcp : "Read Control Points" - lê uma polyline e considera seus vértices pontos de controle da B-spline do perfil corrente;

2-Dcp : "Draw Control Points" - desenha os pontos de controle da b-spline do perfil corrente;

3-Dsp : "Draw Sline" - Desenha a B-spline do perfil corrente;

4-BSS : "Build Spline Surface" - Constrói uma superfície B-spline a partir das B-splines existentes;

5-DSCP : "Draw Surface Control Points" - desenha os pontos de controle da superfície;

6-DRP : "Draw Radial Profile" - desenha uma seção cilíndrica da superfície;

7-Sections : calcula seções planas e escreve um script de plotagem automática;

9-eBi : recalcula a posição de um ponto de controle de forma que a curva passe por um determinado ponto;

10-mvY : calcula os deslocamentos em y e z de forma que um ponto possa ser deslocado em y mantendo-se na mesma superfície cilíndrica;

11-dsppl : desenha uma spline no plano expandido;

12-Smkn : "Smooth curve at knot" - Suaviza uma curva num dado nó;

13-UpKn : "Upgrade knots" - uniformiza os vetores nó das curvas;

14-Exit - Sai do comando.

O código da função é:

```
void EdCurv(void) // Edit Curves
{
                int np, i, j, k, c, spn;
                char la[20], str[20];
  AcDbEntity *pObj;
  AcDbObjectId eId;
                AcGePoint3d vl[100], pt;
  AcGePoint3dArray pta;
                if (!BLADE)
                {       // call routine for blade since it hasn't been created yet
                        HELIXSURF();
                }
ads_printf("\nSelect: <1-Rcp 2-Dcp 3-Dsp 4-BSS 5-DSCP 6-DRP 7-Sections 8-DIAG 9-eBi 10-mvY 11-
                dsppl 12-Smkn 13-UpKn 14-Exit>: ");
ads_getint("",&c);
switch (c)
{
case 1:
                pObj = selectEntity(eId, AcDb::kForRead);
                np=bvl(eId, vl);
  pta.setLogicalLength(np);
                strcpy(la, pObj->layer());
                *(la+1)='\0';
                spn=atoi(la);

                pObj->close();

                for (i=0; i<=np; i++)
                {
                        sp_cl[spn].bx[i+1]=(double)vl[i][0];
                        sp_cl[spn].by[i+1]=(double)vl[i][1];
                        sp_cl[spn].bz[i+1]=(double)vl[i][2];
                }

                strcat(la, "bsp");
                dsp(sp_cl[spn], la);
break;
case 2:
                pObj = selectEntity(eId, AcDb::kForRead);
                strcpy(la, pObj->layer());
                *(la+1)='\0';
                pObj->close();


                spn=atoi(la);
                strcat(la, "control");
                dcp(sp_cl[spn], la);
break;

case 3:
                pObj = selectEntity(eId, AcDb::kForRead);
                strcpy(la, pObj->layer());
                pObj->close();

                *(la+1)='\0';
                spn=atoi(la);

                strcat(la, "bsp");
                dsp(sp_cl[spn], la);
break;

case 4:
                bls_cl.build(9, 4, sp_cl);

break;


case 9:
                int cpn, pn, spn;
```

```
                double  N[40], u[100];

                ads_getint("\nEnter control point number: ", &cpn);
                ads_getint("\nEnter point number: ", &pn);
                ads_getint("\nEnter spline number: ", &spn);

                u[1]=0;
                for (i=2; i<=ka.bl[spn].np; i++)
                      u[i]=u[i-1] + (double)sqrt((ka.bl[spn].xe[i]-ka.bl[spn].xe[i-1])*(ka.bl[spn].xe[i]-
                ka.bl[spn].xe[i-1]) +
                                                              (ka.bl[spn].ye[i]-ka.bl[spn].ye[i-
                1])*(ka.bl[spn].ye[i]-ka.bl[spn].ye[i-1]) +
                                                              (ka.bl[spn].ze[i]-ka.bl[spn].ze[i-
                1])*(ka.bl[spn].ze[i]-ka.bl[spn].ze[i-1]));

                for (i=1; i<=ka.bl[spn].np; i++) u[i]=u[i]*sp_cl[spn].knm/u[ka.bl[spn].np];


                sp_cl[spn].BasisRow(N, u[pn]);
                double sum;

                if (N[cpn]>0)
                {
                              sum=(double)ka.bl[spn].xe[pn];
                              for (i=1; i<cpn; i++) sum=sum-N[i]*sp_cl[spn].bx[i];
                              for (i=cpn+1; i<=sp_cl[spn].n0; i++) sum=sum-N[i]*sp_cl[spn].bx[i];
                              sp_cl[spn].bx[cpn]=sum/N[cpn];

                              sum=(double)ka.bl[spn].ye[pn];
                              for (i=1; i<cpn; i++) sum=sum-N[i]*sp_cl[spn].by[i];
                              for (i=cpn+1; i<=sp_cl[spn].n0; i++) sum=sum-N[i]*sp_cl[spn].by[i];
                              sp_cl[spn].by[cpn]=sum/N[cpn];

                              sum=(double)ka.bl[spn].ze[pn];
                              for (i=1; i<cpn; i++) sum=sum-N[i]*sp_cl[spn].bz[i];
                              for (i=cpn+1; i<=sp_cl[spn].n0; i++) sum=sum-N[i]*sp_cl[spn].bz[i];
                              sp_cl[spn].bz[cpn]=sum/N[cpn];
                }
                _itoa(spn, str, 10);
                strcpy(la, str);
                strcat(la, "control");
                dcp(sp_cl[spn], la);
                strcpy(la, str);
                strcat(la, "bsp");
                dsp(sp_cl[spn], la);

break;

case 10:
  double y1, y2, z1, z2, dyy, dzz, R, t;
   int rc;
   ads_point pT, pT2;

   if ((rc = ads_getpoint(NULL, "\nSelect point: ", pT)) == RTNORM)
   rc = sa_u2w(pT, pT);

  if ((rc = ads_getpoint(pT, "\nSelect point: ", pT2)) == RTNORM)
      rc = sa_u2w(pT2, pT2);

                y1=pT[1];
                y2=pT2[1];
                z1=pT[2];
                R=sqrt(y1*y1 + z1*z1);
                dyy=y2-y1;
                t=asin(y2/R);
                z2=R*cos(t);
                dzz=z2-z1;

                ads_printf("\ndy= %f  dz=%f", dyy, dzz);
```

```
break;

case 11: //draw spline in expanded view

                pObj = selectEntity(eId, AcDb::kForRead);
                strcpy(la, pObj->layer());
                pObj->close();

                *(la+1)='\0';
                spn=atoi(la);

                strcat(la, "plan");

                dsppl(sp_cl[spn], la, (spn+1)*ka.d/20,
                (double)atan(ka.pd*ka.d/2/3.141592/((spn+1)*ka.d/20)) );

break;

case 12:
                //smooth knot

                ads_getint("\nEnter control point number: ", &cpn);
                ads_getint("\nEnter spline number: ", &spn);

                sp_cl[spn].SmoothKnot(cpn);

                _itoa(spn, str, 10);
                strcpy(la, str);
                strcat(la, "control");
                dcp(sp_cl[spn], la);


break;

case 13:  //upgrade knots;


                ads_getint("\nEnter knots: ", &c);
for (i=1; i<=9; i++)
{
                ads_printf("\nUpgrading knots for spline: %d", i);
                np=sp_cl[i].UpgradeKnots(c);
   if (!np) ads_printf("\nCould not upgrade knots");
                else
                {
                        _itoa(i, str, 10);
                        strcpy(la, str);
                        strcat(la, "control");
                        dcp(sp_cl[i], la);
                }
}

for (i=1; i<=9; i++)
                for (j=1; j<=9; j++)
                        if (i != j)
                                for (k=1; k<=sp_cl[j].nn; k++)
                                        if (!sp_cl[i].KnotExists(sp_cl[j].knots[k]))

                                                sp_cl[i].InsertKnot(sp_cl[j].knots[k]);


for (i=1; i<=9; i++)
{
                ads_printf("\nKnots for spline: %d \n", i);
                for(j=1; j<=sp_cl[i].nn; j++)
                        ads_printf("[%d] %1.3f ", j, sp_cl[i].knots[j]);
}


break;
```

```
case 5:
                DrawSurfaceControlPoints(bls_cl,"long");

break;

case 6:
                double rr;
                ads_getreal("\nEnter Radius: ", &rr);
                DrawRadialProfile(bls_cl, (double)rr, "rad");
break;

case 7:
                // find section
                double zc, zci, zcf, dz;
                double xmin, xmax, ymin, ymax, ux, x, y;
                double vv;
                char st[20];
                int ndiv=299;
                AcGePoint3d pt;
                double step=(double)bls_cl.knum/(double)ndiv;
                ads_getreal("\nEnter starting zc: ", &zci);
                ads_getreal("\nEnter ending zc: ", &zcf);
                ads_getreal("\nEnter zc step: ", &dz);
                ads_getstring(0,"\nScript plot file: <plot.scr>", st);
                if (strlen(st)==0)strcpy(st, "plot.scr");
                  FILE *f;
  f = fopen( st, "a+" );
  if( f == NULL ) printf("\nCould not open script file");
  else
  {             fprintf(f, "CMDDIA 0\n");
                        fprintf(f, "ZOOM E\n");
  }


                zc=zci;
                pta.setLogicalLength(ndiv+1);

// find global maximum and minimum coordinates
// and draw template base
if(xgmin==0.0)
{
                ux=0.0;
                while (ux<=1.0)
                {
                        vv=0.0;
                        while (vv<=1.0)
                        {
                                        x=bls_cl.SS(bls_cl.bxx, ux, vv);
                                        y=bls_cl.SS(bls_cl.byy, ux, vv);
                                        if (x>xgmax)xgmax=x;
                                        if (y>ygmax)ygmax=y;
                                        if (x<xgmin)xgmin=x;
                                        if (y<ygmin)ygmin=y;
                                        vv+=0.02;

                        }
                        ux+=0.02;
                }
                ads_printf("\nxgmin= %f ygmin= %f xgmax= %f ygmax= %f",
                                xgmin, ygmin, xgmax, ygmax);

                                pta.setLogicalLength(5);

                                        pt[0]=1.1*xgmin;
                                        pt[1]=1.1*ygmin;
                                        pt[2]=0.0;
                                        pta.setAt(0,pt);
```

```
                                        pt[0]=1.1*xgmax;
                                        pt[1]=1.1*ygmin;
                                        pta.setAt(1,pt);

                                        pt[0]=1.1*xgmax;
                                        pt[1]=1.1*ygmax;
                                        pta.setAt(2,pt);

                                        pt[0]=1.1*xgmin;
                                        pt[1]=1.1*ygmax;
                                        pta.setAt(3,pt);

                                        pt[0]=1.1*xgmin;
                                        pt[1]=1.1*ygmin;
                                        pta.setAt(4,pt);

                                        DrawPoly(pta, "base");
}


// find minimum values for lowest profile
// find xu0 and xuh

if(zu0==0.0)
{
        xu0=bls_cl.SS(bls_cl.bxx, 0, 0);
        zu0=bls_cl.SS(bls_cl.bzz, 0, 0);
        xuh=0;
        ux=0;
        for (k=1; k<((ndiv+ndiv)/3); k++)
        {
                x=bls_cl.SS(bls_cl.bxx, ux, 0);
                if (x>xuh)
                {
                        xuh=x;
                        zuh=bls_cl.SS(bls_cl.bzz, ux, 0);
                }
                ux+=step;
        }
        ads_printf("\nxu0= %f zu0= %f xuh= %f zuh= %f", xu0, zu0, xuh, zuh);
}
while (zc<=zcf)
        {       xmin=xmax=ymin=ymax=0.0;
                ads_printf("\nzc= %f",zc);
                double uu=0;
                double rmax, r=0;
                int k=0;
                pt[2]=zc; // z-coordinate is constant - no need to calculate it
                while(uu<=bls_cl.knum) //??=
                {
                        vv=bls_cl.V(zc, uu);
                        if (vv>0)
                        {
                                pt[0]=bls_cl.SS(bls_cl.bxx, uu, vv);
                                pt[1]=bls_cl.SS(bls_cl.byy, uu, vv);
                                if (uu==0){xmin=pt[0]; ymin=pt[1];}
                                r=pt[0]*pt[0]+pt[1]*pt[1];
                                if ((uu>0.2) &&
                                        (uu<0.8) &&
                                        (r>rmax))
                                {rmax=r; xmax=pt[0]; ymax=pt[1];}
                                pta.setAt(k,pt);
                                k++;
                        }
                        uu+=step;
                }
                uu=1.0;
                vv=bls_cl.V(zc, uu);
                if (vv>0)
                {
```

```
                pt[0]=bls_cl.SS(bls_cl.bxx, uu, vv);
                pt[1]=bls_cl.SS(bls_cl.byy, uu, vv);
                pta.setAt(k,pt);
}
ads_printf("\nxmin=%f xmax=%f ymin=%f ymax=%f",  xmin,  xmax,  ymin, ymax);
if (k>0)
{
        char *str2, str3[20];
        strcpy(la, "CORTE");
        _gcvt(zc, 10, str); // watch out! decimal point is not allowed in layer ames
        str2= strchr(str, '.');
        strcpy(str3, strchr(str, '.')+1);
        *str2='\0';
        strcat(str, "_");
        strcat(str, str3);
        strcat(la, str);
        addLayer(la, 1);
        pta.setLogicalLength(k);
        DrawPoly(pta, la);
        if ((zc<0.1*ka.d) && (zc>zu0))
        {
                y=sqrt(0.01*ka.d*ka.d - zc*zc);
                pta.setLogicalLength(5);
                pt[0]=xu0;
                pt[1]=y;
                pta.setAt(0,pt);

                pt[0]=xuh;
                pt[1]=y;
                pta.setAt(1,pt);

                pt[0]=xuh;
                pt[1]=-y;
                pta.setAt(2,pt);
                pt[0]=xu0;
                pt[1]=-y;
                pta.setAt(3,pt);
                pt[0]=xu0;
                pt[1]=y;
                pta.setAt(4,pt);

                DrawPoly(pta, la);
        }

        x=1.1*xgmin;
        y=(ymax-ymin)*(x-xmin)/(xmax-xmin)+ymin;
        if(y>1.1*ygmax)
        {
                y=1.1*ygmax;
                x=(xmax-xmin)*(y-ymin)/(ymax-ymin)+xmin;
        }
        xmin=x;
        ymin=y;

        x=1.1*xgmax;
        y=(ymax-ymin)*(x-xmin)/(xmax-xmin)+ymin;
        if(y<1.1*ygmin)
        {
                y=1.1*ygmin;
                x=(xmax-xmin)*(y-ymin)/(ymax-ymin)+xmin;
        }
        xmax=x;
        ymax=y;


        // draw center line
        pta.setLogicalLength(2);
        pt[0]=xmin;
        pt[1]=ymin;
        pta.setAt(0,pt);
```

```
                        pt[0]=xmax;
                        pt[1]=ymax;
                        pta.setAt(1,pt);
                        DrawPoly(pta, la);

                        // add script line for this section
                        fprintf(f, "-layer off * Y \n");
                        fprintf(f, "-layer on base \n");
                        strcpy(st, "-layer on ");
                        strcat(st, la);
                        strcat(st, " \n");
                        fprintf(f, st);
                        fprintf(f, "plot e 0\n");

                    }

                  zc+=dz;
                }
                fprintf(f, "-layer on * \n");
                fprintf(f, "-layer s 0 \n");
                fprintf(f, "CMDDIA 1\n");
fclose(f);

break;
} //end switch
}
```

(ALG 6-88)

h) A função findLayer

Esta função verifica a existência de um "layer". O código é:

```
int findLayer(char *la)
{
   AcDbLayerTable *pLayerTbl;
   acdbCurDwg()->getLayerTable(pLayerTbl, AcDb::kForRead);
if (pLayerTbl->has(la)){        pLayerTbl->close();
                    return 1;
                }
                else{
                    pLayerTbl->close();
                    return 0;
                }
}
```

(ALG 6-89)

i) A função addLayer

Esta função cria um layer com o nome apontado por la, com a cor c. O código é:

```
void addLayer(char *la, int c)
{
  AcDbLayerTable *pLayerTbl;
  acdbCurDwg()->getLayerTable(pLayerTbl, AcDb::kForWrite);

  if (!pLayerTbl->has(la)) {
    AcDbLayerTableRecord *pLayerTblRcd
      = new AcDbLayerTableRecord;
    pLayerTblRcd->setName(la);
    pLayerTblRcd->setIsFrozen(0);// layer to THAWED
    pLayerTblRcd->setIsOff(0);  // layer to ON
    pLayerTblRcd->setVPDFLT(0);  // viewport default
    pLayerTblRcd->setIsLocked(0);// un-locked

    AcCmColor color;
    color.setColorIndex(c); // set color to parameter c
    pLayerTblRcd->setColor(color);
   pLayerTbl->add(pLayerTblRcd);
    pLayerTblRcd->close();
    pLayerTbl->close();
  } else {
                    pLayerTbl->close(); //hard to find BUG!!!
    ads_printf("\nlayer already exists");
  }
}
```

(ALG 6-90)

j) A função dcp(b_Spline, char*)

Esta função desenha os pontos de controle de uma b_spline uniforme. O código é:

```
void dcp(B_Spline sp, char *la)
{
            AcGePoint3dArray pta;
              AcGePoint3d pt;
              pta.setLogicalLength(sp.n0);
              for(int k=1 ; k<=sp.n0; k++)
              {
                        pt[0]=sp.bx[k];
                        pt[1]=sp.by[k];
                        pt[2]=sp.bz[k];
                        pta.setAt(k-1,pt);
              }
              DrawPoly(pta,la);
}
```

(ALG 6-91)

l) A função dcp(b_Spline_cl, char*)

Esta função desenha os pontos de controle de uma b_spline não uniforme. O código é:

```
void dcp(B_Spline_cl sp, char *la)
{
        AcGePoint3dArray pta;
        AcGePoint3d pt;
        pta.setLogicalLength(sp.n0);
        for(int k=1 ; k<=sp.n0; k++)
        {
                pt[0]=sp.bx[k];
                pt[1]=sp.by[k];
                pt[2]=sp.bz[k];
                pta.setAt(k-1,pt);
        }
        DrawPoly(pta,la);
}
```

(ALG 6-92)

m) A função pl2sp

Esta função converte coordenadas plano-cilíndricas em coordenadas espaciais. O código é:

```
void pl2sp(double r, double phi, double xp, double yp, double& xe, double& ye, double& ze)
{
        xe=  (double)xp*(double)sin(phi)+(double)yp*(double)cos(phi);
        ye=  -(double)r*(double)sin((xp*cos(phi)-(double)yp*(double)sin(phi))/r);
        ze=  (double)r*(double)cos((xp*cos(phi)-(double)yp*(double)sin(phi))/r);

}
```

(ALG 6-93)

n) A função sp2pl

Esta função converte coordenadas espaciais em coordenadas plano-cilíndricas. O código é:

```
void sp2pl(double r, double phi, double xe, double ye, double ze, double& xp, double& yp)
{
        yp=(double)xe*(double)cos(phi)-(double)r*(double)atan(-ye/ze)*(double)sin(phi);
        xp=((double)xe-(double)yp*(double)cos(phi))/(double)sin(phi);
}
```

(ALG 6-94)

o) A função dsp(b_Spline, char*)

Esta função desenha uma B-spline uniforme no espaço. O código é:

```
void dsp(B_Spline sp, char *la)
{
  AcGePoint3dArray pta;
        AcGePoint3d pt;
        int ndiv=199, k;
        double step, uu, tt;
        pta.setLogicalLength(ndiv);
            tt=(double)sp.n0-sp.k0+1;
            step=(double)tt/(double)ndiv;
            uu=0;
            for(k=1 ; k<=ndiv; k++)
            {
                pt[0]=sp.S(sp.bx, uu);
                pt[1]=sp.S(sp.by, uu);
                pt[2]=sp.S(sp.bz, uu);
                pta.setAt(k-1,pt);
                uu=uu+step;
            }
            DrawPoly(pta,la);
}
```

(ALG 6-95)

p) A função dsp(b_Spline_cl, char*)

Esta função desenha b-spline não uniforme no espaço. O código é:

```
void dsp(B_Spline_cl sp, char *la)
{
  AcGePoint3dArray pta;
        AcGePoint3d pt;
        int ndiv=199, k;
        double step, uu;
        pta.setLogicalLength(ndiv);
        step=(double)1.0/(double)ndiv;
        uu=(double)0.0;
        for(k=1 ; k<=ndiv; k++)
```

```
        {
            pt[0]=sp.S(sp.bx, uu);
            pt[1]=sp.S(sp.by, uu);
            pt[2]=sp.S(sp.bz, uu);
            pta.setAt(k-1,pt);
            uu=uu+step;
        }
            DrawPoly(pta,la);
}
```

(ALG 6-96)

q) A função dsppl(b_Spline, char*)

Esta função desenha uma B-spline uniforme no plano expandido. O código é:

```
void dsppl(B_Spline sp, char *la, double r, double phi)
//draw expanded view spline
{
   AcGePoint3dArray pta;
            AcGePoint3d pt;
            int ndiv=199, k;
            double step, uu, tt, xe, ye, ze, xp, yp;
            pta.setLogicalLength(ndiv);

                tt=(double)sp.n0-sp.k0+1;
                step=(double)tt/(double)ndiv;
                uu=0;
                for(k=1 ; k<=ndiv; k++)
                {
                        xe=sp.S(sp.bx, uu);
                        ye=sp.S(sp.by, uu);
                        ze=sp.S(sp.bz, uu);
                        sp2pl(r, phi, xe, ye, ze, xp, yp);
                        pt[0]=xp;
                        pt[1]=yp;
                        pt[2]=0;
                        pta.setAt(k-1,pt);
                        uu=uu+step;
                }

                DrawPoly(pta,la);
}
```

(ALG 6-97)

r) A função dsppl(b_Spline_cl, char*)

Esta função desenha uma B-spline não uniforme no plano expandido. O código é:

```
void dsppl(B_Spline_cl sp, char *la, double r, double phi)
//draw expanded view spline
{
  AcGePoint3dArray pta;
        AcGePoint3d pt;
        int ndiv=199, k;
        double step, uu, tt, xe, ye, ze, xp, yp;
        pta.setLogicalLength(ndiv);
            tt=(double)sp.n0-sp.k0+1;
            step=(double)tt/(double)ndiv;
            uu=0;
            for(k=1 ; k<=ndiv; k++)
            {
                  xe=sp.S(sp.bx, uu);
                  ye=sp.S(sp.by, uu);
                  ze=sp.S(sp.bz, uu);
                  sp2pl(r, phi, xe, ye, ze, xp, yp);
                  pt[0]=xp;
                  pt[1]=yp;
                  pt[2]=0;
                  pta.setAt(k-1,pt);
                  uu=uu+step;
            }
            DrawPoly(pta,la);
}
```

(ALG 6-98)

s) A função dknv(b_Spline, char*)

Esta função desenha uma polilinha ("polyline") que reproduz visualmente o vetor nó de uma B-spline uniforme. O código é:

```
void dknv(B_Spline sp, char *la)
{
        AcGePoint3dArray pta;
            AcGePoint3d pt;
            pta.setLogicalLength(sp.nn);
            for(int k=1; k<=sp.nn; k++)
            {
                  pt[0]=(double)sp.knots[k];
                  pt[1]=pt[2]=0;
                  pta.setAt(k-1,pt);
            }
            DrawPoly(pta,la);
}
```

(ALG 6-99)

t) A função dknv(b_Spline_cl, char*)

Esta função desenha uma polyline que reproduz visualmente o vetor nó de uma B-spline não uniforme. O código é:

```
void dknv(B_Spline_cl sp, char *la)
{
        AcGePoint3dArray pta;
            AcGePoint3d pt;
            pta.setLogicalLength(sp.nn);
            for(int k=1; k<=sp.nn; k++)
            {
                    pt[0]=(double)sp.knots[k];
                    pt[1]=pt[2]=0;
                    pta.setAt(k-1,pt);
            }
            DrawPoly(pta,la);
}
```

(ALG 6-100)

u) A função DrawSurfaceControlPoints(B_Surf, char *)

Esta função desenha a poligonal dos pontos de controle de uma superfície B-spline uniforme. O código é:

```
void DrawSurfaceControlPoints(B_Surf bls, char *la)
{
  AcGePoint3dArray pta;
        AcGePoint3d pt;

        pta.setLogicalLength(bls.m0);
        for (int i=1; i<=bls.n0; i++)
        {
            ads_printf("\nDrawing surface control points fo spline: %d", i);
            for(int k=1 ; k<=bls.m0; k++)
            {
                    //ads_printf("\nDrawing surface control point: %d", k);
                    pt[0]=bls.bxx[i][k];
                    pt[1]=bls.byy[i][k];
                    pt[2]=bls.bzz[i][k];
                    pta.setAt(k-1,pt);
            }
            DrawPoly(pta,la);
        }
}
```

(ALG 6-101)

v) A função DrawSurfaceControlPoints(B_Surf_cl, char *)

Esta função desenha a poligonal dos pontos de controle de uma superfície B-spline não uniforme. O código é:

```
void DrawSurfaceControlPoints(B_Surf_cl bls, char *la)
{
  AcGePoint3dArray pta;
        AcGePoint3d pt;

        pta.setLogicalLength(bls.m0);
        for (int i=1; i<=bls.n0; i++)
        {
            ads_printf("\nDrawing surface control points for spline: %d", i);
            for(int k=1 ; k<=bls.m0; k++)
            {
                //ads_printf("\nDrawing surface control point: %d", k);
                pt[0]=bls.bxx[i][k];
                pt[1]=bls.byy[i][k];
                pt[2]=bls.bzz[i][k];
                pta.setAt(k-1, pt);
            }
            DrawPoly(pta, la);
        }
}
```

(ALG 6-102)

x) A função DrawRadialProfile(B_Surf, double, char*)

Esta função desenha o perfil obtido através da intersecção de uma superfície B-spline uniforme com uma superfície cilíndrica. O código é:

```
void DrawRadialProfile(B_Surf bs, double rr, char *la)
{       AcGePoint3dArray pta;
        AcGePoint3d pt;
        int ndiv=199, k;
        double step, uu, vv, tt;
        pta.setLogicalLength(ndiv);
            tt=(double)bs.knum;
            step=(double)tt/(double)ndiv;
            uu=0;
            for(k=1 ; k<=ndiv; k++)
            {       vv=bs.Vr(rr, uu);
                    pt[0]=bs.SS(bs.bxx, uu, vv);
                    pt[1]=bs.SS(bs.byy, uu, vv);
                    pt[2]=bs.SS(bs.bzz, uu, vv);
                    pta.setAt(k-1,pt);
                    uu=uu+step;}
            DrawPoly(pta,la);
}
```

(ALG 6-103)

y) A função DrawRadialProfile(B_Surf_cl , double, char*)

Esta função desenha o perfil obtido através da intersecção de uma superfície B-spline não uniforme com uma superfície cilíndrica. O código é:

```
void DrawRadialProfile(B_Surf_cl bs, double rr, char *la)
{
  AcGePoint3dArray pta;
            AcGePoint3d pt;
            int ndiv=199, k;
            double step, uu, vv, tt;
            pta.setLogicalLength(ndiv);
                  tt=(double)bs.knum;
                  step=(double)tt/(double)ndiv;
                  uu=0;
                  for(k=1 ; k<=ndiv; k++)
                  {
                        vv=bs.Vr(rr, uu);
                        pt[0]=bs.SS(bs.bxx, uu, vv);
                        pt[1]=bs.SS(bs.byy, uu, vv);
                        pt[2]=bs.SS(bs.bzz, uu, vv);
                        pta.setAt(k-1,pt);
                        uu=uu+step;
                  }
                  DrawPoly(pta,la);
}
```

(ALG 6-104)

z) A função UBgNUB

Esta função permite aproximar uma B-spline uniforme num conjunto de pontos igualmente espaçados sobre uma curva B-spline não uniforme. O código é:

```
void UBgNUB(B_Spline& ubs, B_Spline_cl nubs, int nc, int k, double eps)
{
// build uniform spline;
            int i;
            double LL, x[50], y[50], z[50], step;

            ubs.n0=nc;
            ubs.k0=k;
            ubs.saveknots();

            x[1]=nubs.bx[1];
            y[1]=nubs.by[1];
            z[1]=nubs.bz[1];
```

```
x[nc]=nubs.bx[nubs.n0];
y[nc]=nubs.by[nubs.n0];
z[nc]=nubs.bz[nubs.n0];

LL=nubs.length((double)1.0, eps);

step=LL/(double)(nc-1);
LL=step;
for(i=2; i<nc; i++)
{
        x[i]=nubs.S(nubs.bx, nubs.ugl(LL, eps));
        y[i]=nubs.S(nubs.by, nubs.ugl(LL, eps));
        z[i]=nubs.S(nubs.bz, nubs.ugl(LL, eps));
        LL+=step;
}
ubs.adj(x, y, z, nc, k);
}
```

(ALG 6-105)

6.15 - O Aplicativo HELIXSURF

O aplicativo HELIXSURF é uma DLL (Dynamic Link Library) do sistema operacional Windows 95 que contém todas as classes e funções descritas acima. A sua implementação basea-se no ponto de entrada da função extern "C" AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode msg, void*);

Após compilado, pode ser carregado no ambiente de execução do AutoCAD, permitindo acesso direto à base de dados de entidades de um desenho, que poderá ser manipulada com os novos comandos nele implementados, e. g., edCurv, Helixsurf, bsplab, etc.

Optou-se por desenvolver um aplicativo dentro do AutoCAD pelos seguintes motivos:

- utilizar um editor gráfico existente;
- aproveitar o gerenciamento do banco de dados de entidades inerente;
- usufruir das facilidades de impressão.

Assim, foi possível uma grande economia de tempo de desenvolvimento, uma vez que rotinas para as operações acima descritas não tiveram de ser implementadas, permitindo exclusiva dedicação ao método B-spline.

7 - UMA APLICAÇÃO PRÁTICA

Neste capítulo será mostrado um exemplo de utilização do aplicativo HELIXSURF para a modelagem de uma pá de hélice da série sistemática Kaplan com 500mm de diâmetro, 4 pás, razão passo/diâmetro igual a 0,9 e razão área expandida/área projetada igual a 0,6.

7.1 - Carregando o aplicativo HELIXSURF no AutoCAD R. 14

O aplicativo HELIXSURF pode ser carregado no AutoCAD R14 através do menu "Tools" - "Load Application".

As funções disponíveis para o usuário foram abreviadas a fim de facilitar o uso conforme mostrado a seguir:

- "HS" - executa o aplicativo HELIXSURF;
- "ED" - carrega a função EdCurv;
- "BS" - carrega a função bsplab;
- "KA" - executa a função KAPLAN;
- "BC" - executa a função bsplab_cl.

7.2 - Geração inicial da pá

A geração de uma pá é feita executando-se a função HELIXSURF, que solicitará as seguintes características do hélice:

- diâmetro (D);
- número de pás (Z);
- razão área expandida/área projetada (ae/a0);
- razão passo/diâmetro (p/d);
- a precisão de cálculo (eps).

Serão construídas "polylines" tridimensionais, que representam:

- a poligonal dos pontos dos perfis no plano expandido ("layers" 1plan, ..., 9plan);
- a poligonal dos pontos dos perfis no espaço ("layers" 1prof, ..., 9prof);
- a poligonal dos pontos de controle da interpolação B-spline não uniforme de cada perfil no espaço ("layers" 1nucontrol, ..., 9nucontrol);
- as curvas B-Spline interpoladas em cada perfil no espaço ("layers" 1nubsp, ..., 9nubsp);
- a poligonal dos pontos de controle das curvas B-spline de cada perfil após a uniformização dos nós ("layers" 1control, ..., 9control);
- as curvas B-splines de cada perfil após a uniformização dos nós ("layers" 1bsp, ..., 9bsp)
- Os pontos de controle da superfície B-spline ("layer" Long)

Será exibida uma tela semelhante à mostrada na fig. 7.1.

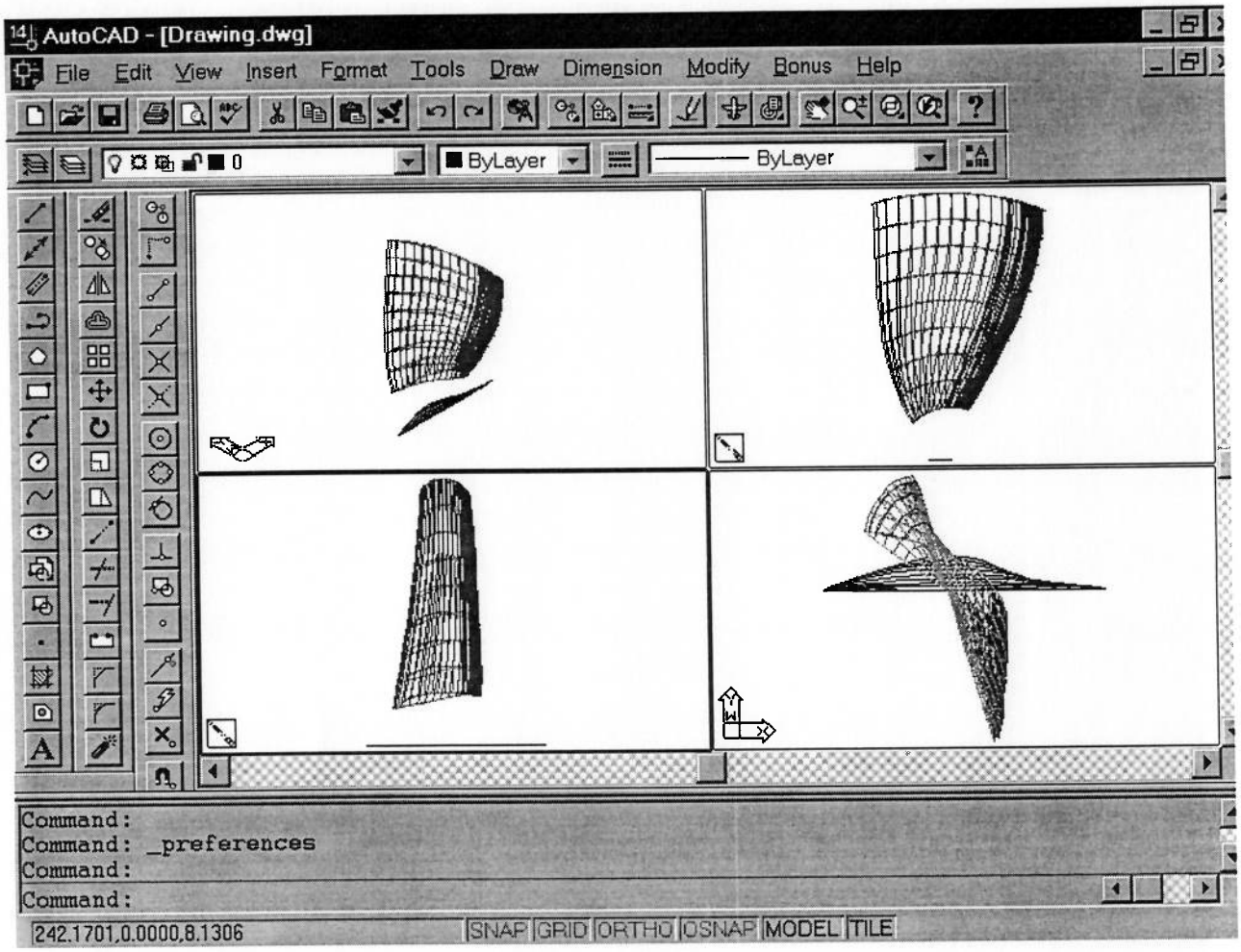

Fig. 7.1 - Hélice kaplan: diâmetro=500mm, Z=4, ae/a0=0,6 e p/d=0,9

7.3 - Edição das curvas e geração da superfície

A edição das curvas é feita manipulando-se diretamente os pontos de controle das B-splines de cada perfil contidas nos "layers" 1bsp a 9bsp, com auxílio dos recursos disponíveis no editor gráfico do AutoCAD, tais como ZOOM e STRETCH. Apesar do resultado gerado logo após a entrada dos dados ser bastante bom, às vezes, torna-se necessário eliminar pequenas inflexões de alguns dos perfis. Isso pode ser feito movendo os pontos de controle das B-splines individuais de cada perfil, através do comando EdCurv - subcomando 1-RCP. É importante observar que a única coordenada que é idêntica no plano de referência global e no sistema de referência de coordenadas cilíndricas é a coordenada **X**. Observou-se, porém, que pequenos deslocamentos na direção da abscissa são suficientes para a maioria dos perfis. Nas situações em que for necessário deslocar-se na direção da ordenada, pode-se empregar o subcomando 10-mvY, que calcula os deslocamentos em **Y** e **Z** de forma que um ponto possa ser deslocado em **Y** mantendo-se na mesma superfície cilíndrica.

Após a edição das curvas individuais de cada perfil, a superfície é gerada com o subcomando 4-BSS. A superfície gerada é de ordem 4 na direção v.

7.4 - Geração das seções planas

As seções planas podem ser geradas com o subcomando 7-Sections do comando EdCurv, definindo-se o valor inicial da coordenada **Z** (zci), o valor final (zcf) e o espaçamento (dz). As seções planas são criadas cada uma no "layer" corteX_Y, onde X contém a parte inteira de zc e Y a parte fracionária, isso porque não é permitido usar o ponto decimal no nome de um "layer". O resultado para algumas seções pode ser visualizado nas figuras 7.2 e 7.3.

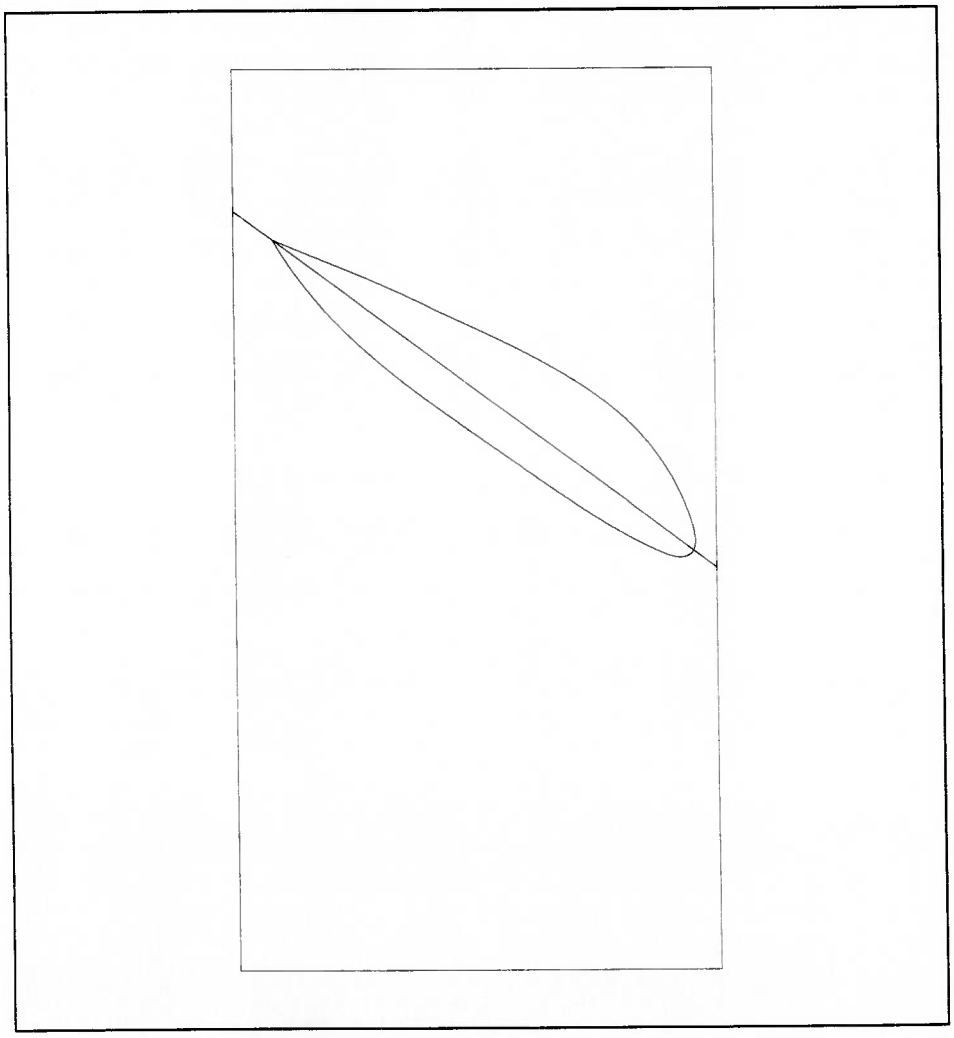

Fig. 7.2 - Seção plana obtida para **Z**c=50mm.

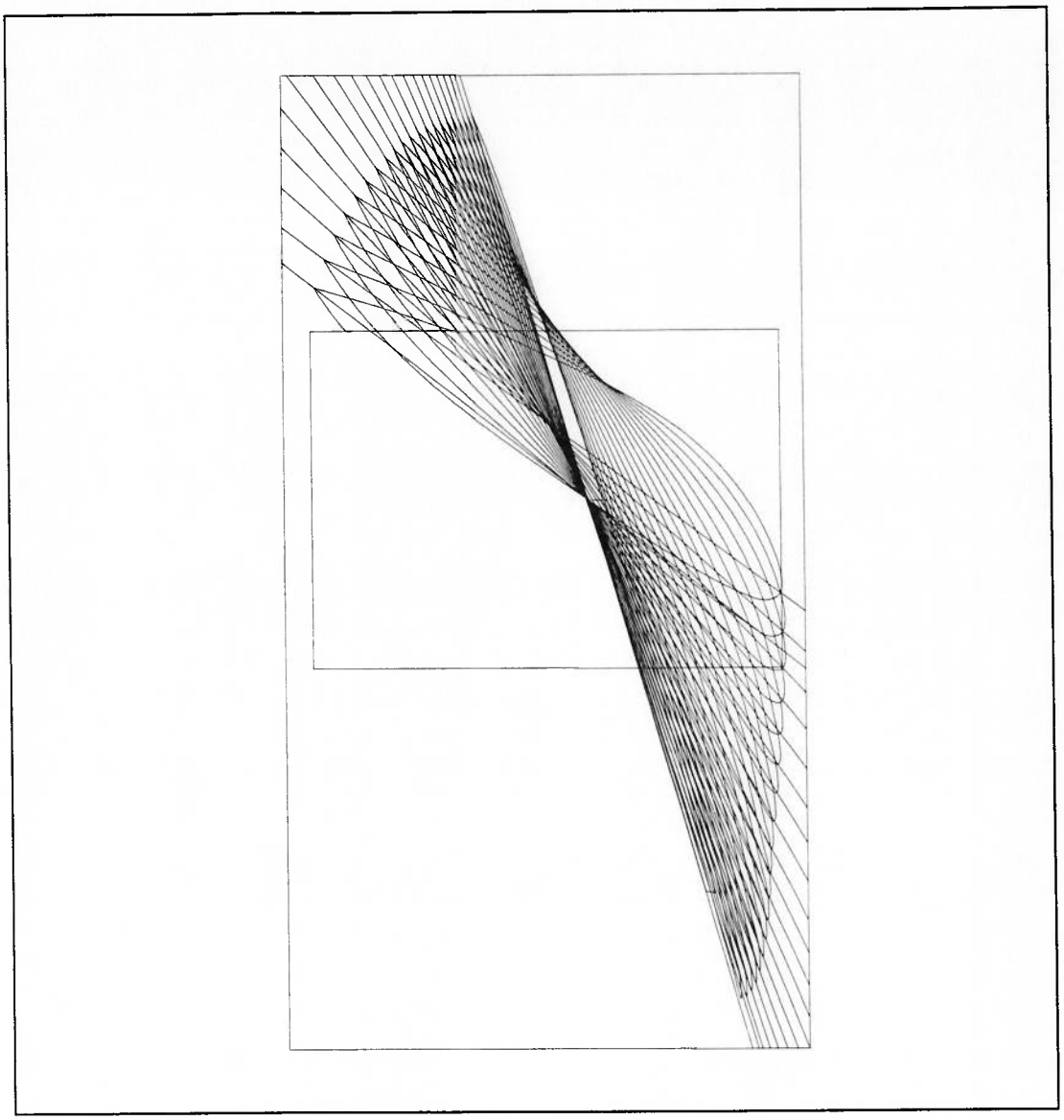

Fig. 7.3 - Seções planas entre as cotas 40 e 250mm a cada 10mm

O retângulo em torno das seções representa a envoltória de todos os possíveis cortes, e é usado para alinhamento dos moldes. A linha transversal é a linha de corte do molde, que foi obtida calculando-se um ponto para u=0 e outro para 0,2<u<0,8 tal que a distância até a origem seja máxima.

Para cada seção obtida é gerado um conjunto de linhas em linguagem "script" que é armazenado num arquivo texto, que poderá ser usado posteriormente para a plotagem automática das seções, conforme exemplificado abaixo:

```
CMDDIA 0
ZOOM E
-layer off * Y
-layer on base
-layer on CORTE50_
plot e 0
-layer on *
-layer s 0
CMDDIA 1
```

8 - ELEMENTOS PARA FABRICAÇÃO CASEIRA DE UMA PÁ

Neste capítulo pretende-se apresentar orientações gerais para permitir a fabricação caseira de uma pá de hélice utilizando o método proposto neste trabalho.

Após a escolha das características do hélice e edição das curvas geram-se seções equi-espaçadas da mesma medida da espessura das placas que serão usadas na confecção dos moldes. Por exemplo, podem ser usadas placas de cartolina de 2mm de espessura, que resultará em 108 seções planas para o hélice de 500mm de diâmetro modelado no capítulo anterior.

8.1 - Impressão dos moldes

Os moldes podem ser impressos na escala 1:1 de forma automática com auxílio do "script" de plotagem em uma impressora laser, por exemplo. O "script" foi criado durante a geração das seções, ficando armazenado num arquivo tipo texto. A execução do "script" é feita com o comando SCRIPT do AutoCAD. Nas figuras 8.1 a 8.12 são mostrados alguns exemplos de moldes.

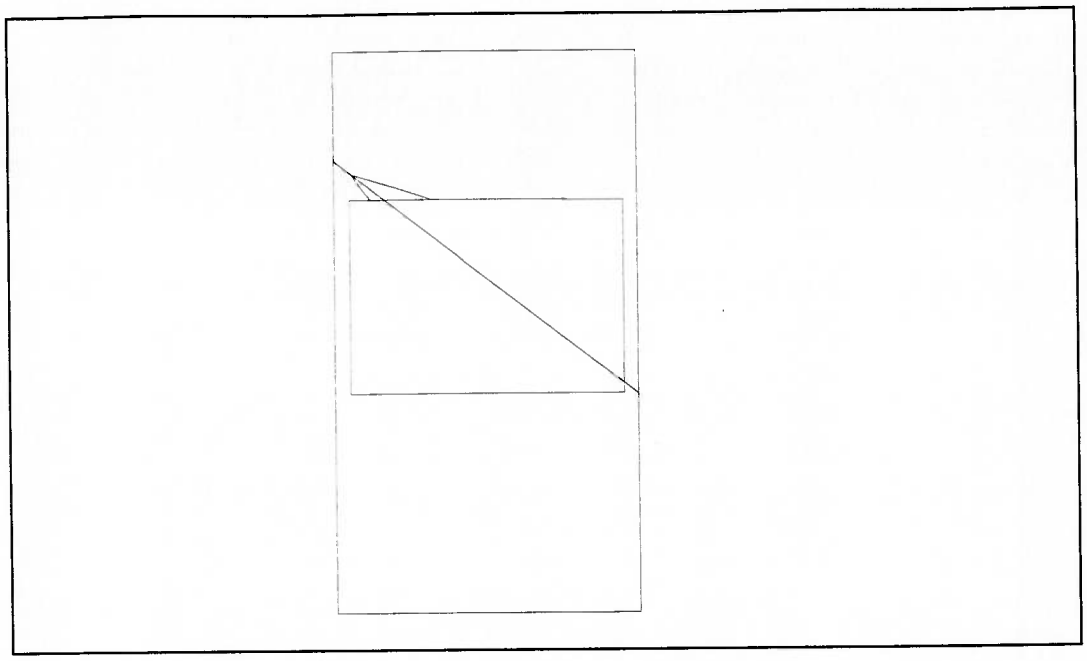

Fig. 8.1 - Seção a 40mm

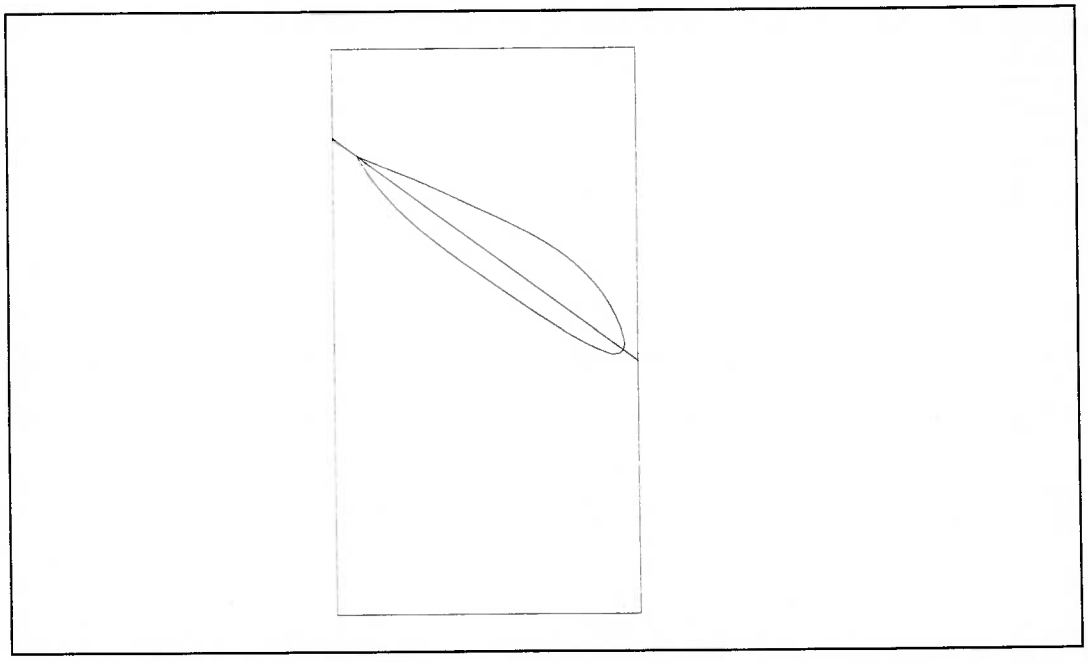

Fig. 8.2 - Seção a 50mm

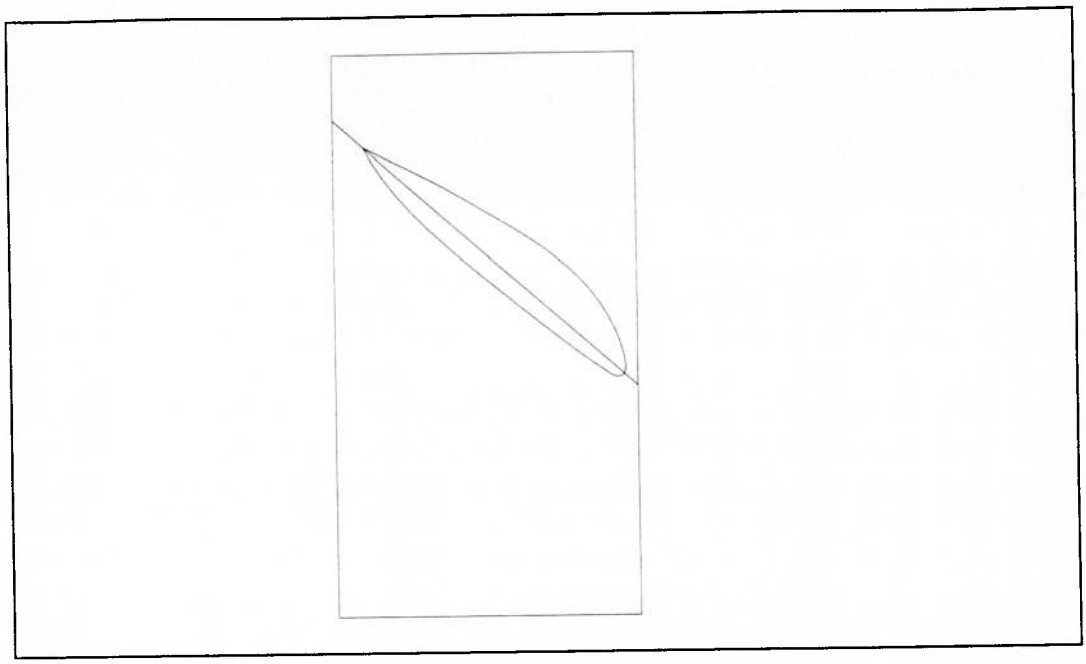

Fig. 8.3 - Seção a 60mm

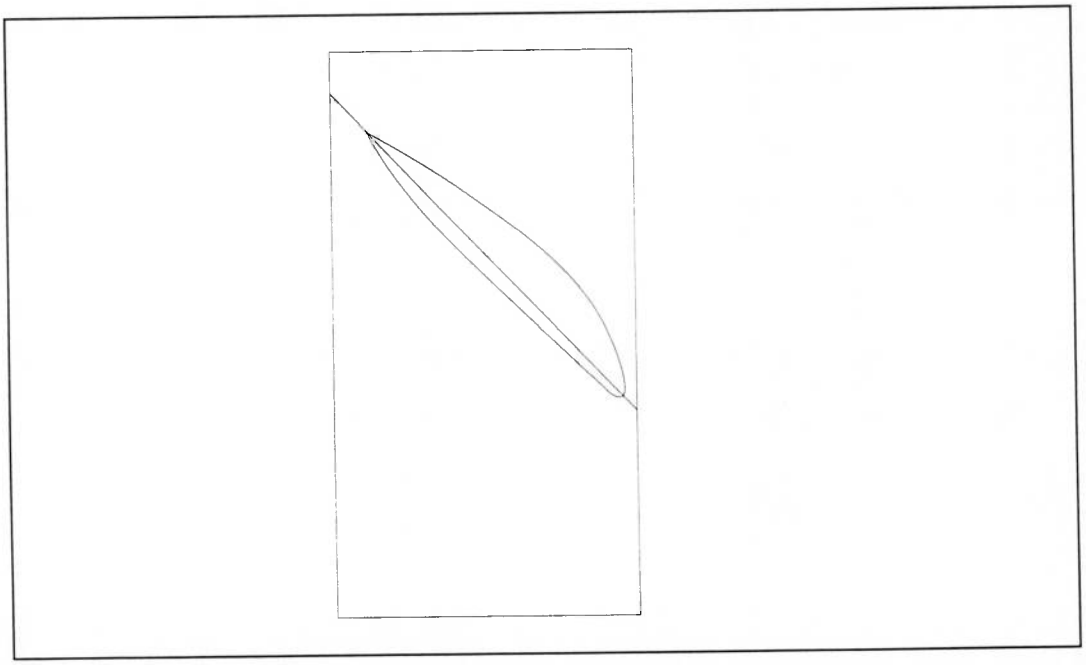

Fig. 8.4 - Seção a 70mm

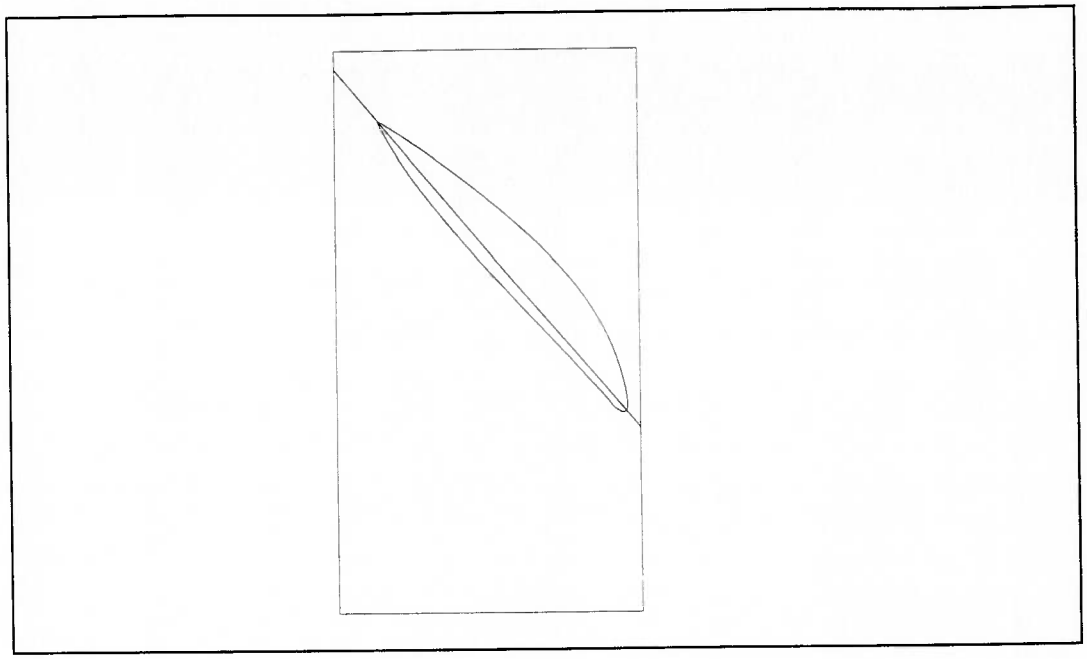

Fig. 8.5 - Seção a 80mm

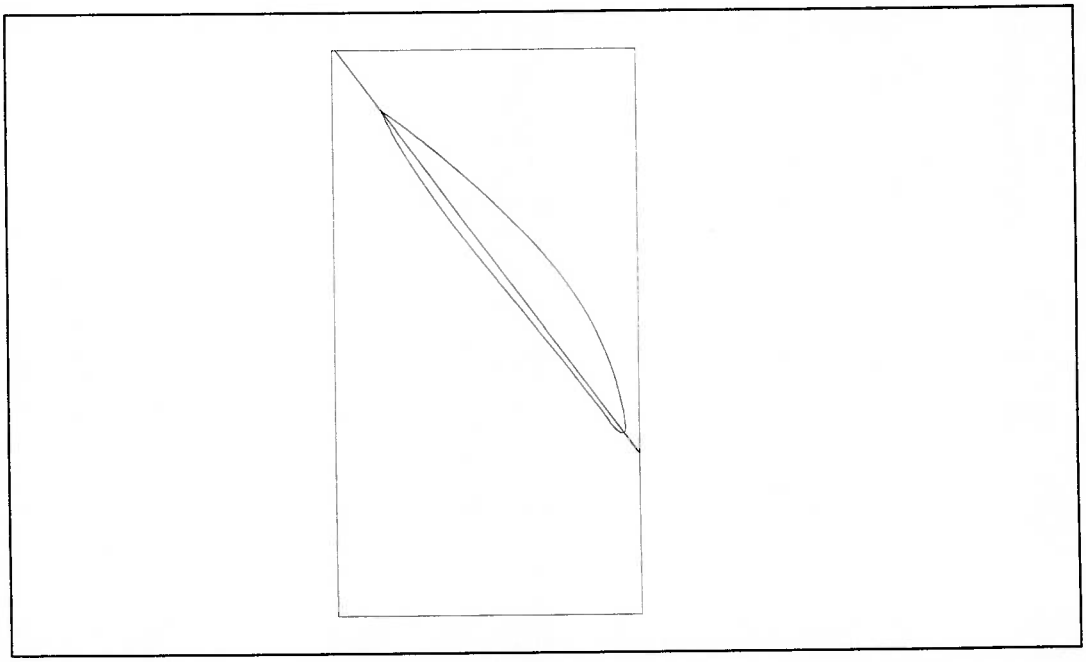

Fig. 8.6 - Seção a 90mm

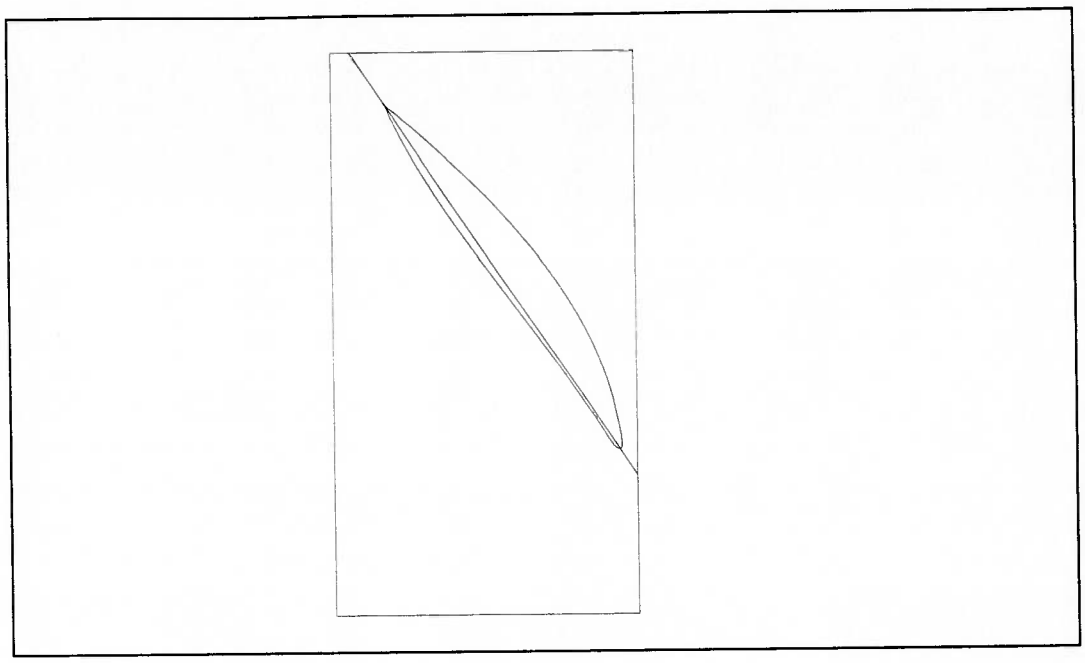

Fig. 8.7 - Seção a 100mm

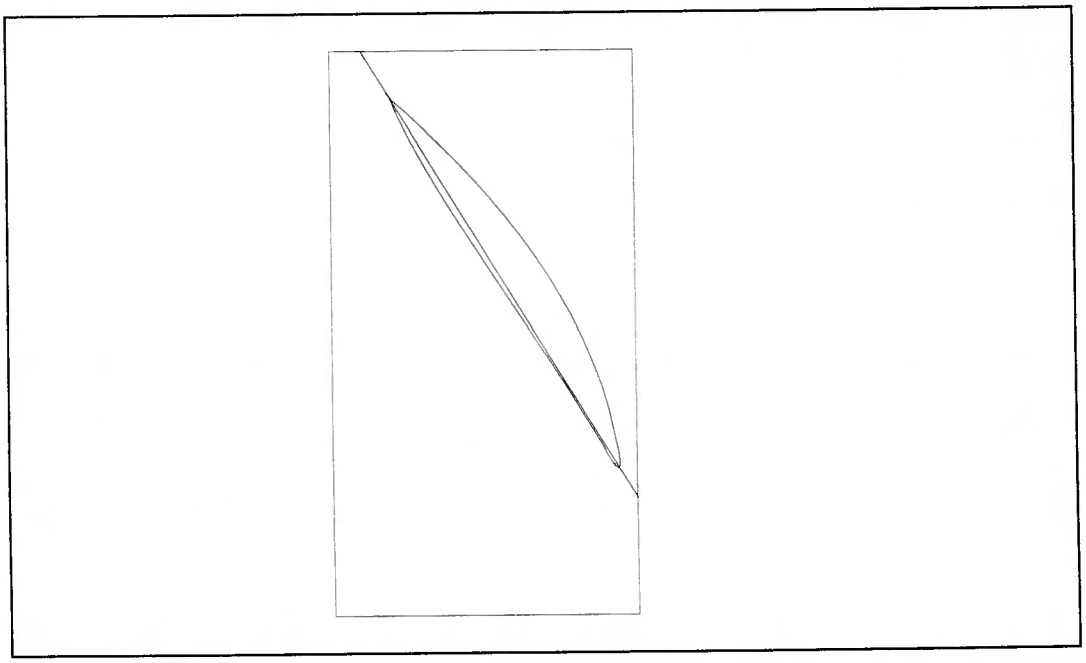

Fig. 8.8 - Seção a 110mm

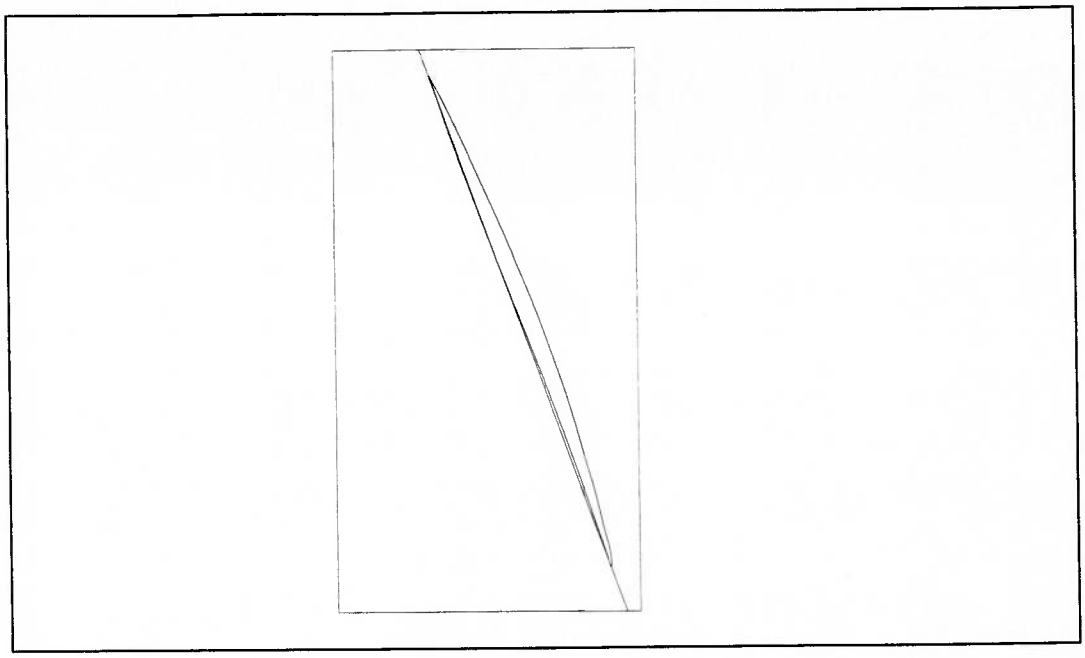

Fig. 8.9 - Seção a 190mm

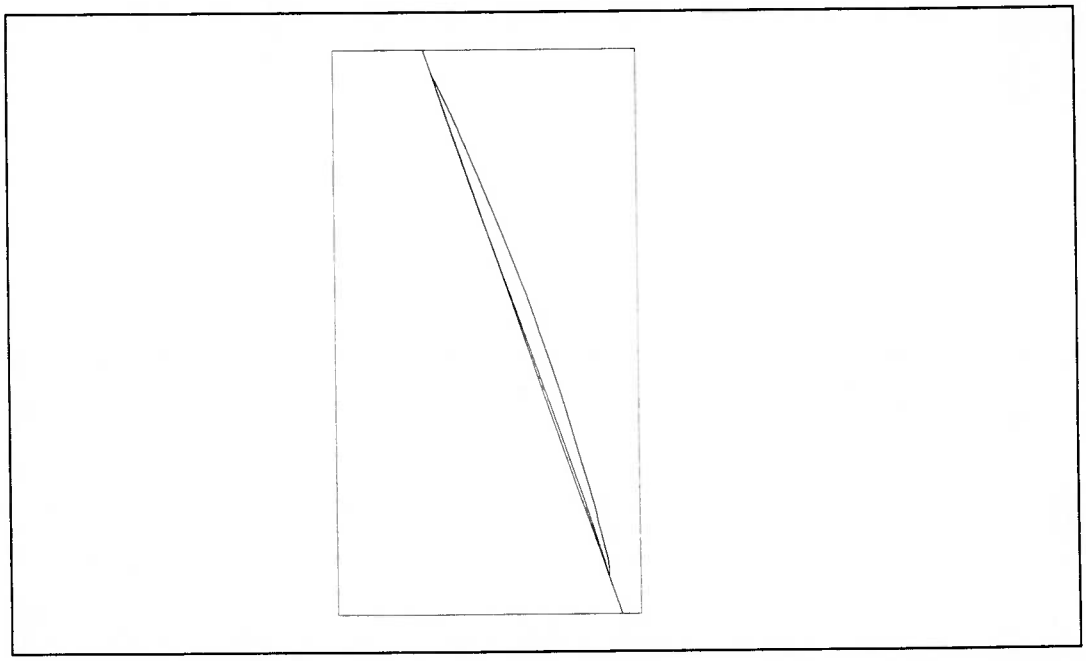

Fig. 8.10 - Seção a 200mm

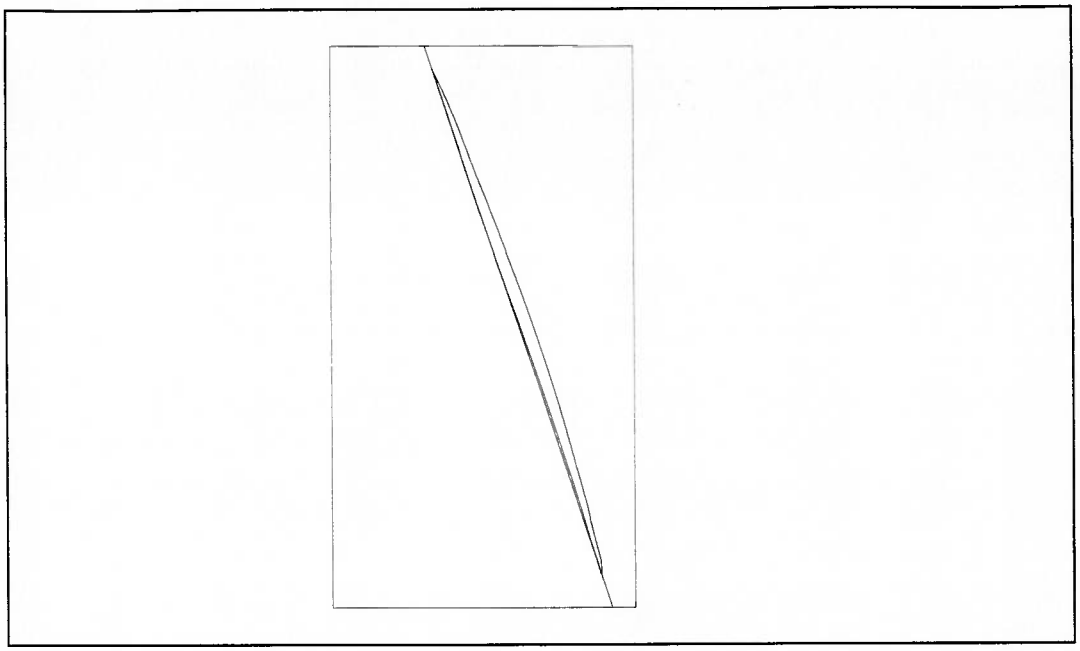

Fig. 8.11 - Seção a 210mm

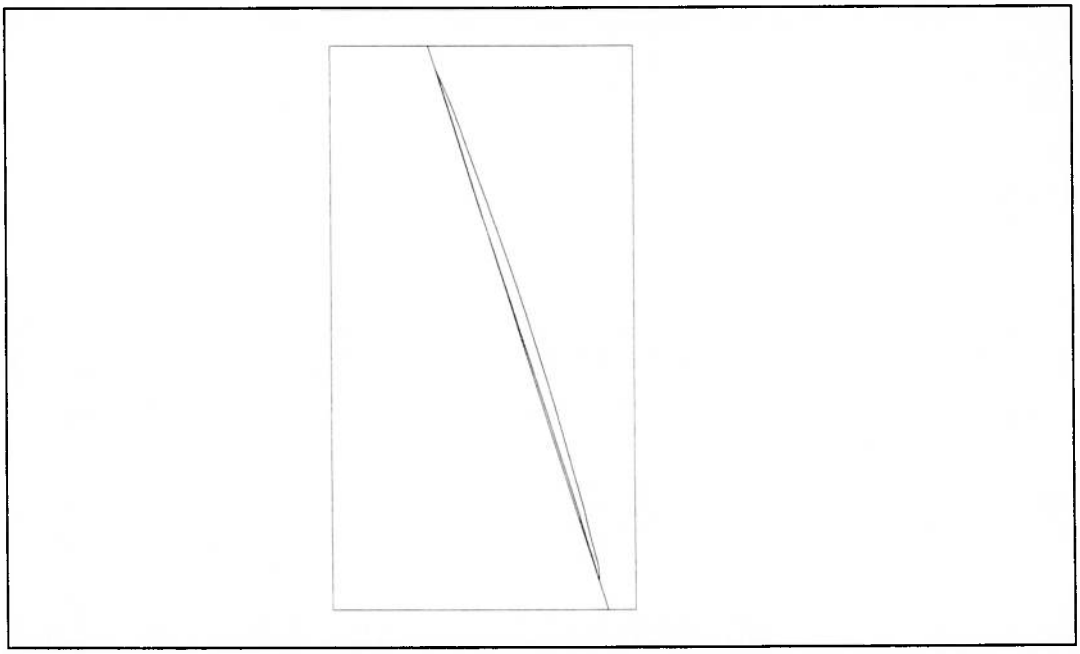

Fig. 8.12 - Seção a 220mm

8.2 - Corte das placas planas

O corte das placas de cartolina pode ser feito manualmente com auxílio de serra tico-tico de mesa, posicionando os moldes sobre as placas de cartolina. Deve-se notar que cada placa será dividida em duas seguindo a linha longitudinal da seção.

8.3 - Montagem e carenamento do molde tri-dimensional

Após o corte, as placas dos moldes deverão ser empilhadas e alinhadas de forma a formar dois sanduíches de placas coladas entre si com cola de madeira ou similar de forma a garantir rigidez ao conjunto que resultará em dois blocos que formarão o molde tri-dimensional.

O carenamento das duas metades do molde tri-dimensional consiste em eliminar os degraus na cavidade interna do sanduíche, preenchendo os vazios com massa de ponçar. O acabamento final poderá ser feito manualmente com lixas d'água.

8.4 - Fundição das pás

Após o acabamento superficial do molde pode-se seguir o procedimento normal de modelagem com fibra de vidro e resina. Inicalmente deve-se aplicar cera de carnaúba e, sobre ela uma fina película de álcool polivinílico, ambos com a função de desmoldantes. Em seguida, após a junção das duas partes do molde, resina de poliéster termofixa é depositada na cavidade pela sua abertura correspondente à intersecção com o bosso, que é posicionada para cima. Após a cura deve-se ter uma pá. O processo deve ser repetido até completar o número total de pás. Finalmente pode-se proceder à montagem do hélice fixando-se as pás no bosso.

9 - CONCLUSÕES

O presente trabalho permitiu apresentar o método B-spline através de sua aplicação na modelagem de uma superfície complexa como é o caso de uma pá de um hélice da série sistemática Kaplan. Inicialmente buscou-se entender o comportamento das B-splines de quarta ordem (terceiro grau) uniformes e não-uniformes quando empregadas na aproximação e na interpolação de curvas no espaço tri-dimensional. Observou-se que bons resultados são obtidos utilizando-se interpolação por B-splines não uniformes. A seguir, procurou-se uma proposta de sistema de referência para a representação matemática da geometria do hélice no espaço tri-dimensional, tendo sido encontrada e adaptada para este estudo a proposta de KLEIN (1975). Com esse sistema de referência, inicialmente interpolaram-se curvas B-splines pelos pontos dos perfis da série sistemática.

Finalmente, seguindo a proposta de PIEGL; TYLER (1997) e NOWACKI el al. (1995) empregou-se o método denominado "skinning" para produzir uma superfície tri-dimensional B-Spline a partir das curvas individuais de cada perfil. Tal processo, denominado, uniformização dos vetores nós, exige que todas as curvas possuam o mesmo vetor nó, o que poderia ser obtido através da

inserção de todos os nós de cada curva nos vetores nós de todas as demais, conforme sugere PIEGL; TYLER (1997), demandando enormes espaços de armazenamento em memória e perda de desempenho. Foi nesta fase que foi feita uma proposta original deste trabalho, quando notou-se que é possível obter vetores nós uniformes através de um algoritmo repetitivo de inserção e remoção de nós, que consite da inserção inicial de 25 nós equi-espaçados nos vetores nós de cada curva, seguida da remoção de todos os nós passíveis de remoção e finalmente na inserção dos nós que não puderam ser removidos de cada curva nos vetores nós de todas as demais. Tal processo permitiu a redução significativa do número de equações e espaço de memória, assim como na melhoria de desempenho.

Por último, o método permitiu na implementação computacional de um aplicativo na linguagem C++, executável no programa AutoCAD R. 14, com o qual foi possível levar a cabo a modelagem de uma pá de hélice da série sistemática Kaplan que permitiu produzir moldes que poderiam ser usados na fabricação caseira de um hélice em resina plástica.

Deve-se salientar que, uma vez aplicado o método para uma pá, tem-se uma expressão matemática que permite obter com exatidão qualquer ponto sobre a pá. Desta forma a reprodução dessa pá será tão precisa quanto o método de fabricação o permitir.

Para fins de exemplificação da aplicação do método apresenta-se uma proposta de fabricação caseira de uma pá de hélice através da elaboração de um molde fêmea tri-dimensional que poderá ser preenchido com resina plástica termofixa reforçada ou não com fibra de vidro. As pás individuais construídas por esse processo podem, então, ser fixadas ao bosso produzindo o hélice final.

10 - PROPOSTAS FUTURAS DE DESENVOLVIMENTO

A fim de dar prosseguimento a este trabalho propõe-se o seguinte desenvolvimento futuro:

a) estudar o comportamento das derivadas primeira e segunda parciais em relação aos eixos globais de referência a fim de buscar rotinas auxiliares que melhorem o resultado da interpolação e, assim, reduzindo a necessidade de interação humana;

b) estudar rotinas de cálculo de trajetória de ferramentas de corte de máquinas de controle numérico e implementar algoritmos que permitam reproduzir as superfícies geradas;

c) fabricar e testar um protótipo;

d) adicionalmente, poder-se-ia empregar o método para a modelagem de outras superfícies importantes na área naval, e. g., a superfície do casco do navio.