Analyse par intervalles pour le lancé de rayon et pour l'analyse de stabilité

Fabrice Le Bars, Jan Sliwka, Luc Jaulin

JD-JN-MACS 2009
Analyse par intervalles pour le lancé de rayon et pour l'analyse de stabilité

Fabrice Le Bars, Jan Sliwka, Luc Jaulin

Laboratoire des Développements des Technologies Nouvelles
Ecole Nationale Supérieure des Ingénieurs des Etudes et Techniques d'Armement
2 Rue François Verny, 29806 Brest, France

lebarsfa@ensieta.fr, sliwkaja@ensieta.fr, jaulinlu@ensieta.fr
http://www.ensieta.fr/

Résumé— **Dans cet article, nous allons montrer que le problème du lancé de rayon couramment utilisé en graphisme et un problème d'analyse en stabilité d'un système paramétrique linéaire invariant sont deux problèmes très semblables. Nous allons proposer un algorithme unique utilisant le calcul par intervalles capable de résoudre ces deux problèmes de façon garantie et efficace.**

Mots-clés— **MACS, lancé de rayon, ray tracing, ray casting, Phong, analyse de stabilité, Routh, intervalles, méthodes ensemblistes, SIVIA, inversion ensembliste.**

I. Introduction

Le but de cet article est de proposer un algorithme rapide pour caractériser une fonction du type

$$f(\mathbf{p}) = \min_{\substack{d\in[d]\\ g(\mathbf{p},d)\leq 0}} d \tag{1}$$

Ici, $[d]$ est un intervalle de $\mathbb{R}$ et $[\mathbf{p}]$ est un pavé de $\mathbb{R}^n$. Par *caractériser*, nous entendons ici tracer (il faut pour cela que $n = \dim \mathbf{p} = 2$), mais on peut aussi le comprendre comme obtenir un codage efficace qui nous permet d'évaluer rapidement $f(\mathbf{p})$.

Dans un premier temps, nous ferons quelques rappels sur le calcul par intervalles. Ensuite, nous décrirons un algorithme permettant de calculer la plus petite racine d'une fonction, en utilisant les intervalles. Nous réutiliserons et adapterons cet algorithme dans la partie suivante pour des calculs de lancé de rayon. Enfin, nous montrerons dans une dernière partie comment les algorithmes réalisés pour le lancé de rayon peuvent être repris pour calculer le degré de stabilité d'un système paramétrique linéaire invariant.

II. Calcul par intervalles

Un intervalle[1] est un sous-ensemble fermé et borné de $\mathbb{R}$. Par exemple, $[1,3]$, $\{1\}$, $]-\infty,6]$, $\mathbb{R}$ et $\emptyset$ sont des intervalles alors que $]1,3[$ et $[1,2]\cup[3,4]$ n'en sont pas. On notera $\mathbb{IR}$ l'ensemble des intervalles de $\mathbb{R}$.

Un pavé, ou *intervalle vectoriel* $[\mathbf{x}]$ de $\mathbb{R}^n$ est un vecteur dont les composantes sont des intervalles :

$$[\mathbf{x}] = [x_1^-, x_1^+] \times \cdots \times [x_n^-, x_n^+] = [x_1] \times \cdots \times [x_n].$$

La *longueur* $w([\mathbf{x}])$ d'un pavé $[\mathbf{x}]$ est la longueur de son côté le plus long. Par convention $w(\emptyset) = -\infty$. Si $w([\mathbf{x}]) = 0$, $[\mathbf{x}]$ est dit *dégénéré*. Dans ce cas, $[\mathbf{x}]$ est un singleton de $\mathbb{R}^n$ et sera noté $\{\mathbf{x}\}$.

La fonction $[f] : \mathbb{IR} \to \mathbb{IR}$ est une *fonction d'inclusion* de la fonction réelle $f : \mathbb{R} \to \mathbb{R}$ si et seulement si pour tout $[x] \in \mathbb{IR}$

$$f([x]) \subset [f]([x])$$

III. Calcul de la plus petite racine d'une fonction

Nous cherchons ici à proposer un algorithme qui calcule un intervalle $[a,b]$ tel que

$$a \leq \min_{\substack{d\in[d]\\ g(d)\leq 0}} d \leq b \tag{2}$$

où g est une fonction non linéaire pour laquelle nous disposons d'une fonction d'inclusion.

En d'autres termes, on recherche l'abscisse pour laquelle la fonction g s'annule en devenant négative pour la première fois sur $[d]$.

L'intervalle $[a,b]$ sera tel que la fonction y est décroissante, de façon à ce qu'on puisse par la suite appliquer une dichotomie pour déterminer plus précisément $\min_{\substack{d\in[d]\\ g(d)\leq 0}} d$.

L'intervalle $[a,b]$ sera donc caractérisé par ceci :
- $0 \in [g]([a,b])$
- $[g']([a,b]) \subset]-\infty, 0[$
- $g(a) > 0$
- $g(b) < 0$

L'idée est de découper l'intervalle initial $[d]$ jusqu'à ce qu'on obtienne un intervalle $[a,b]$ satisfaisant les règles précédentes. Pour cela, une liste de type pile est utilisée pour

stocker les différents découpages dans un ordre croissant. Voici un début d'algorithme simple permettant cela :

```
Algorithme GetDMinInterval(in : [d] ;out : [a, b])
1   L = {[d]} ;a = d⁻ ;b = d⁺ ;
2   while L ≠ ∅,
3       pop the first interval [d] out of L ;
4       if w([d]) < ε,
5           return PERHAPS ;
6       if ([g]([d]) ⊂ [0, ∞[),
7           a = d⁺ ;
8           continue ;
9       if ([g′]([d]) ⊂ ]−∞, 0]),
10          if ([g]([d⁺]) < 0),
11              b = d⁺ ;
12              return TRUE ;
13      else
14          bissect [d] into [d1] and [d2] ;
15          L = {[d1], [d2]} ∪ L ;
16  end while
17  return FALSE ;
```

Algorithme retournant un intervalle $[a, b]$ tel que $a \leq \min_{\substack{d\in[d]\\ g(d)\leq 0}} d \leq b$ et g décroissante sur $[a, b]$

La fonction GetDMinInterval() prend en entrée l'intervalle $[d]$ sur lequel on veut étudier g et retourne TRUE lorsqu'un intervalle $[a, b]$ satisfaisant a été trouvé, FALSE lorsque g ne s'annule pas sur $[d]$ en devenant négative ou PERHAPS lorsque c'est indéterminé. A la suite de cet algorithme, il suffit d'appliquer une dichotomie sur l'intervalle $[a, b]$ pour obtenir une estimation plus précise de $\min_{\substack{d\in[d]\\ g(d)\leq 0}} d$.

IV. Lancé de rayon

A. Principe du lancé de rayon

Le lancé de rayon[3] (ou ray tracing, ray casting en anglais) est une technique utilisée pour l'affichage d'images simulées sur ordinateur et notamment la gestion des effets de lumière. Le principe du lancé de rayon est de reconstituer le trajet inverse de la lumière en partant de l'écran vers l'objet. La technique classique est de travailler pixel par pixel : pour chaque pixel de l'écran (=chaque pixel de l'image), on trace un rayon ayant pour origine l'oeil (qui regarde l'écran) et passant par le centre du pixel et on regarde quel est le premier objet touché par ce rayon. Les caractéristiques de cet objet et sa distance à l'écran détermineront la couleur du pixel. Ceci permet d'obtenir des images réalistes, mais demande beaucoup de calculs. Nous allons voir comment il pourrait être possible d'améliorer cette technique en utilisant des algorithmes ensemblistes.

Comme dans les méthodes de lancé de rayon classiques, les objets seront décrits par des fonctions implicites avec une équation du type :

$$f(x, y, z) \leq 0 \text{ avec } f : \mathbb{R}^3 \mapsto \mathbb{R} \tag{3}$$

L'écran (ou l'image) correspond à un pavé $[\mathbf{p}] \subset \mathbb{R}^2$. L'oeil sera placé à l'origine du repère $\mathtt{R}(O,\vec{i},\vec{j},\vec{k})$ et l'écran sera placé en $z = 1$, parallèle au plan $(O,\vec{i},\vec{j})$. Un rayon passant par le pixel $\mathbf{p} = (p_1, p_2) \in [\mathbf{p}]$ sera donc décrit par les équations

$$\begin{aligned} x &= p_1 * d \\ y &= p_2 * d \\ z &= d \end{aligned} \tag{4}$$

avec $d \geq 1$.

B. Utilisation d'algorithmes d'inversion ensembliste pour une première partie d'algorithme de ray tracing

Dans un premier temps, on se propose de déterminer l'ensemble des pixels de l'écran qui affichent l'objet, sans se soucier de la couleur précise qu'on va leur donner. Les pixels dans la projection de l'objet sur l'écran seront marqués en rouge, ceux en dehors seront en bleu et ceux qui sont incertains en jaune. L'ensemble des pixels rouges est donc le suivant :

$$\{\mathbf{p} = (p_1, p_2) \in [\mathbf{p}], \exists d \in [d], f(p_1 * d, p_2 * d, d) \leq 0\}. \tag{5}$$

Pour cela, nous utiliserons l'algorithme d'inversion ensembliste SIVIA (Set Inverter Via Interval Analysis)[1] donné ici dans sa version récursive :

```
Algorithme Sivia(in : [p])
1   if w([p]) < ε_p,
2       draw [p] in yellow ;
3   else
4       test = Inside([p]) ;
5       if (test = TRUE),
6           draw [p] in red ;
7       else
8           if (test = PERHAPS),
9               bissect [p] into [p1] and [p2] ;
10              Sivia([p1]) ;
11              Sivia([p2]) ;
12          else
13              draw [p] in blue ;
```

Algorithme SIVIA dessinant en rouge les parties de l'image correspondant à la projection de l'objet, en bleu les parties vides et en jaunes les parties indéterminées (frontières de l'objet)

La fonction Inside() décrite ci-dessous retourne TRUE lorsque le pavé $[\mathbf{p}]$ (représentant un ensemble de pixels de l'écran) est dans la projection de l'objet sur l'écran, FALSE lorsqu'on est en dehors et PERHAPS lorsque c'est indéterminé. Dans ce cas, on découpe le pavé $[\mathbf{p}]$ en deux et on rappelle Inside() pour ces deux sous pavés pour tenter de

lever l'ambiguïté.

Algorithme INSIDE(in : $[\mathbf{p}]$)

```
1   inObject = FALSE;
2   [d]= [1, 20];
3   L = {[d]};
4   while L ≠ ∅,
5       pop the first interval [d] out of L;
6       if w([d]) < w([p]),
7           inObject = PERHAPS;
8       else
10          if ([g]([p], [d]) ⊂ ]−∞, 0[),
11              return TRUE;
12          else
13              if (0 ∈ [g]([p], [d])),
14                  bissect [d] into [d1] and [d2];
15                  L := {[d1], [d2]} ∪ L;
16  end while
17  return inObject
```

Algorithme retournant TRUE lorsque le pavé $[\mathbf{p}] \times [d]$ est à l'intérieur de l'objet,
FALSE lorsqu'il est en dehors et PERHAPS quand c'est toujours indéterminé au bout d'un certain nombre d'itérations
(lorsqu'on a découpé $[d]$ jusqu'à une longueur $\varepsilon = w([\mathbf{p}])$)

Dans Inside(), on note :

$$\begin{array}{c} g : \mathbb{R}^3 \mapsto \mathbb{R} \\ (p_1, p_2, d) \mapsto f(p_1 * d, p_2 * d, d) \end{array} . \tag{6}$$

Le test $[g]([\mathbf{p}], [d]) \subset]-\infty, 0[$ cherche donc à déterminer si l'ensemble des rayons passant par l'ensemble des pixels correspondant au pavé $[\mathbf{p}]$ (un pixel a pour coordonnées $(p_1, p_2) \in [\mathbf{p}]$) et pris à une distance de l'oeil comprise entre d^- et d^+ (avec $[d] = [d^-, d^+]$) est dans l'objet. Si tel est le cas, Inside() retourne TRUE. Si cet ensemble est clairement en-dehors de l'objet, Inside() retourne FALSE. Si c'est indéterminé, Inside() découpe l'intervalle $[d]$ en 2 intervalles et les place dans une liste, afin de recommencer le test sur ces nouveaux intervalles. Si la longueur d'un intervalle découpé devient trop petite, Inside() retourne PERHAPS, à moins qu'un des intervalles restant dans la liste soit clairement identifié comme étant dans l'objet. Il est à noter que cet algorithme n'est pas optimisé, il est juste présenté ici comme étape intermédiaire.

Voici l'image que trouve l'algorithme, avec pour objet une boule :

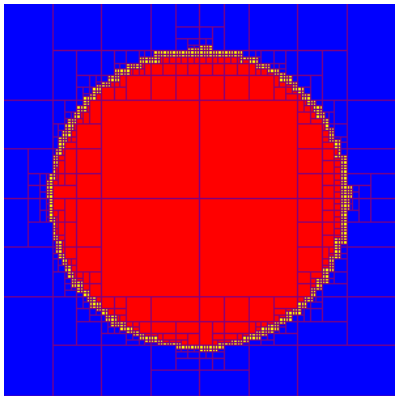

Image 1 : Découpage en $[\mathbf{p}]$ réalisé dans le cas de l'image d'une boule.

C. *Amélioration de l'algorithme précédent en utilisant l'algorithme de calcul de la plus petite racine d'une fonction*

Pour commencer à pouvoir dessiner des images plus réalistes et notamment des effets de lumières, il est nécessaire de connaître la distance à l'écran de la surface de l'objet à afficher. Dans les programmes réalisés, nous avons en effet choisi d'utiliser le modèle d'illumination de Phong[3] et la formule d'atténuation des couleurs en milieu sous-marin de Stéphane Bazeille[2] pour tenter de simuler des images d'objets dans l'eau, qui nécessitent cette donnée pour chaque pixel de l'écran. Il nous faut donc évaluer pour chaque pixel $\mathbf{p}$

$$d_{\min}(\mathbf{p}) = \min_{\substack{d \in [d] \\ g(\mathbf{p},d) \leq 0}} d. \tag{7}$$

De plus, si on choisit de ne pas chercher à afficher les cas où l'écran serait à l'intérieur d'un objet, on a

$$d_{\min}(\mathbf{p}) = \min_{\substack{d \in [d] \\ g(\mathbf{p},d) \leq 0 \\ g(\mathbf{p},d^-) \geq 0}} d. \tag{8}$$

Nous devons donc modifier la fonction Inside() précédente pour qu'elle puisse nous retourner un intervalle $[a, b]$ pour un pavé $[\mathbf{p}]$ de l'écran tel que :

$$a \leq \min_{\substack{d \in [d] \\ g(\mathbf{p},d) \leq 0 \\ g(\mathbf{p},d^-) \geq 0}} d \leq b \tag{9}$$

On voit donc qu'on retombe sur un problème du même type que celui traité dans la partie III, à la différence que la fonction g dépend d'un paramètre $\mathbf{p}$. Inside() reprend

donc le même principe que GetDminInterval().

Algorithme INSIDE(in : $[\mathbf{p}]$;in : $[d]$;out : $[a,b]$)

```
1   L = {[d]} ; a = d⁻ ; b = d⁺ ;
2   while L ≠ ∅,
3     pop the first interval [d] out of L ;
4     if w([d]) < w([p]),
5       return PERHAPS ;
6     if g([p], d⁻) ⊂ ]−∞, 0[,
7       return FALSE ;
8     else
9       if 0 ∈ g([p], d⁻) and d⁻ = D_INF,
10        return PERHAPS ;
11      else
12        if g([p], d⁻) ⊂ ]0, ∞[,
13          a = d⁻ ;
14        if g([p], [d]) ⊂ ]0, ∞[,
15          a = d⁺ ;
16        else
17          if ∂g/∂d ([p], [d]) ⊂ ]−∞, 0[,
18            if g([p], d⁺) ⊂ ]−∞, 0[,
19              b = d⁺ ;
20              return TRUE ;
21            else
22              if g([p], d⁺) ⊂ ]0, ∞[,
23                a = d⁺ ;
24          else
25            bissect [d] into [d1] and [d2] ;
26            L = {[d1], [d2]} ∪ L ;
27  end while
28  return FALSE
```

Algorithme retournant un intervalle $[a,b]$ tel que $a \leq \min_{\substack{d\in[d] \\ g(\mathbf{p},d)\leq 0 \\ g(\mathbf{p},d^-)\geq 0}} d \leq b$ et g décroissante, avec g dépendant d'un paramètre $[\mathbf{p}]$

Sivia() ne change pratiquement pas par rapport à la version précédente : seul un appel à la fonction Display(), qui calcule par dichotomie $\min_{\substack{d\in[d] \\ g(d)\leq 0}} d$ et calcule la couleur de chaque pixel du pavé $[\mathbf{p}]$ en utilisant le modèle de Phong vient remplacer l'affichage d'un pavé d'une couleur uniforme :

Algorithme SIVIA(in : $[\mathbf{p}]$)

```
1   [d]= [D_INF, D_SUP] = [1, 20] ;
2   if w([p]) < ε,
3       draw [p] in yellow ;
4   else
5       test = Inside([p], [d], [d_min]) ;
6       if (test = TRUE),
7           Display([p], [d_min]) ;
8       else
9           if (test = PERHAPS),
10              bissect [p] into [p1] and [p2] ;
11              Sivia([p1]) ;
12              Sivia([p2]) ;
13          else
14              draw [p] in blue ;
```

Algorithme 5 : Algorithme dessinant l'objet en utilisant le modèle d'illumination de Phong, en bleu les parties vides et en jaunes les parties indéterminées (frontières de l'objet)

ε correspond ici à la largeur d'un pixel de l'écran (il est en effet inutile de découper l'écran en des pavés $[\mathbf{p}]$ qui ne seraient pas visibles car plus petits qu'un pixel).

Algorithme DISPLAY(in : $[\mathbf{p}]$; in : $[d_{\min}]$)

```
1   for each pixel (px, py) of [p],
2       Dichotomie(px,py,[d_min],d_min) ;
3       DrawPixel(px,py,d_min) ;
4   end for
```

Algorithme déterminant $\min_{\substack{d\in[d] \\ g(d)\leq 0}} d$ par dichotomie pour chaque pixel de $[\mathbf{p}]$ et dessinant le pixel avec une couleur dépendant de sa distance à la surface de l'objet, selon le modèle de Phong.

Voici les résultats de l'algorithme dans le cas de l'image d'une boule :

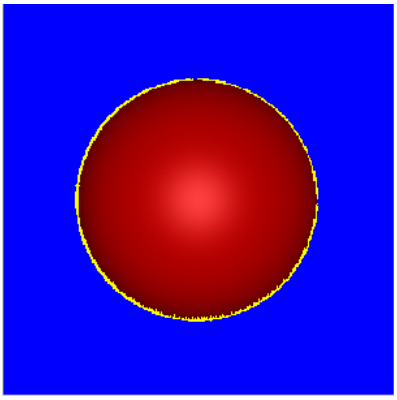

Image 2 : Image d'une boule.

D. Gestion de l'affichage de plusieurs objets

Supposons maintenant qu'on ait plusieurs objets devant l'écran, chacun décrit par une fonction implicite différente. Les fonctions f et g précédentes auront donc maintenant plusieurs composantes, chaque composante i correspondant à un objet. Vu que seul le premier objet traversé par le rayon passant par le pixel $(p_1, p_2) \in [\mathbf{p}]$ nous intéresse (les parties d'objets cachées par d'autres objets n'ont pas à être affichées), il ne faut considérer que l'objet i pour lequel g_i est minimale pour ce pixel et à la distance de l'oeil considérée. Il est alors possible de reprendre l'algorithme précédent sans modification en l'appliquant pour la fonction

$$g_{\min} : (\mathbf{p}, d) \rightarrow \min_i g_i(\mathbf{p}, d). \tag{10}$$

Pour chaque $\mathbf{p} \in [\mathbf{p}]$, Inside() devra donc évaluer :

$$d_{\min}([\mathbf{p}]) = \min_{\substack{d \in [d] \\ g_{\min}(\mathbf{p},d) \leq 0 \\ g_{\min}(\mathbf{p},d^-) \geq 0}} d. \tag{11}$$

La quantité $\frac{\partial g_{\min}}{\partial d}$, qui est utilisée dans Inside(), doit alors être calculée par l'algorithme suivant :

Algorithme GRADGDMIN(in : $[\mathbf{p}]$; in : $[d]$)
1 $[a] = \infty$;$[b] = \infty$;
2 for each component i of $\mathbf{g}$,
3 $[a] = \chi([a] - g_i([\mathbf{p}], [d]), [b], \frac{\partial g_i}{\partial d}([\mathbf{p}], [d]))$;
4 $[b] = g_i([\mathbf{p}], [d])$;
5 end for
6 return $[b]$;

Algorithme calculant $\frac{\partial g_{\min}}{\partial d}$.

Ce dernier appelle[5] :

Algorithme CHI(in : $[a]$; in : $[b]$; in : $[c]$)
1 if $[a] \subset]-\infty, 0[$,
2 return $[b]$;
3 else
4 if $[a] \subset]0, \infty[$,
5 return $[c]$;
6 else
7 return $[\min(b^-, c^-), \max(b^+, c^+)]$;

Algorithme retournant $[b]$ si $[a]$ est négatif, $[c]$ si $[a]$ est positif ou un intervalle contenant $[b]$ et $[c]$ si son signe est incertain.

Inside() ne change donc que par l'appel de $g_{\min}$ et $\frac{\partial g_{\min}}{\partial d}$:

Algorithme INSIDE(in : $[\mathbf{p}]$;in : $[d]$;out : $[a, b]$)
1 $\mathcal{L} := \{[d]\}$;$a = d^-$;$b = d^+$;
2 while $\mathcal{L} \neq \emptyset$,
3 pop the first interval $[d]$ out of $\mathcal{L}$;
4 if $w([d]) < w([\mathbf{p}])$,
5 return PERHAPS ;
6 if $g_{\min}([\mathbf{p}], d^-) \subset]-\infty, 0[$,
7 return FALSE ;
8 else
9 if $0 \in g_{\min}([\mathbf{p}], d^-)$ and $d^- = D\_INF$,
10 return PERHAPS ;
11 else
12 if $g_{\min}([\mathbf{p}], d^-) \subset]0, \infty[$,
13 $a = d^-$;
14 if $g_{\min}([\mathbf{p}], [d]) \subset]0, \infty[$,
15 $a = d^+$;
16 else
17 if $\frac{\partial g_{\min}}{\partial d}([\mathbf{p}], [d]) \subset]-\infty, 0[$,
18 if $g_{\min}([\mathbf{p}], d^+) \subset]-\infty, 0[$,
19 $b = d^+$;
20 return TRUE ;
21 else
22 if $g_{\min}([\mathbf{p}], d^+) \subset]0, \infty[$,
23 $a = d^+$;
24 else
25 bissect $[d]$ into $[d_1]$ and $[d_2]$;
26 $\mathcal{L} = \{[d_1], [d_2]\} \cup \mathcal{L}$;
27 end while
28 return FALSE ;

Sivia() et Display() restent inchangés.

Voici le résultat de l'algorithme pour une image de deux boules :

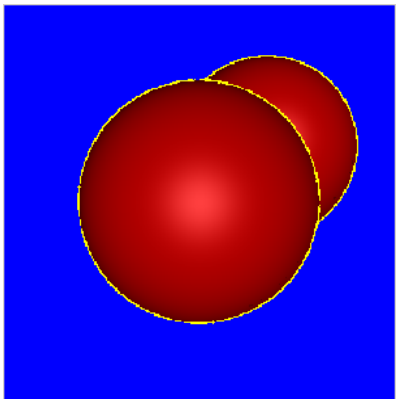

Image 3 : Image de deux boules.

L'affichage de cette image sur un ordinateur portable (Pentium M 1.5GHz) avec un programme développé sous C++ Builder (sans optimisations particulières et avec options de débogages activées) prend 15 secondes. A titre de comparaison, des méthodes plus classiques (notamment basées sur une subdivision de l'espace avec des arbres de type octree) mettent plus de 20 secondes à afficher une image de 300 x 300 pixels (comme celle-ci) avec une seule boule[3].

V. Analyse de la stabilité d'un système paramétrique

On rappelle que le degré de stabilité[4] d'un système linéaire invariant de polynôme caractéristique $P(s)$ est donné par

$$\delta^* = \min_{P(s-\delta) \text{ instable}} \delta. \tag{12}$$

On considère un système linéaire invariant paramétré par un vecteur de paramètre $\mathbf{p}$[6][7] dont le polynôme caractéristique est donné par

$$\begin{aligned} P(s,\mathbf{p}) &= s^3 + (p_1+p_2+2)s^2 + (p_1+p_2+2)s \\ &\quad +2p_1p_2 + 6p_1 + 6p_2 + 2.25. \end{aligned} \tag{13}$$

Donc le degré de stabilité devient

$$\delta^*(\mathbf{p}) = \min_{P(s-\delta,\mathbf{p}) \text{ instable}} \delta. \tag{14}$$

que nous allons chercher à représenter graphiquement. Nous avons

$$\begin{aligned} P(s-\delta,\mathbf{p}) &= (s-\delta)^3 + (p_1+p_2+2)(s-\delta)^2 \\ &\quad +(p_1+p_2+2)(s-\delta) \\ &\quad +2p_1p_2 + 6p_1 + 6p_2 + 2.25, \end{aligned} \tag{15}$$

qui peut se mettre sous la forme

$$P(s-\delta,\mathbf{p}) = s^3 + b_2 s^2 + b_1 s + b_0 \tag{16}$$

avec

$$\begin{aligned} b_0 &= 6p_1 - 2\delta + 6p_2 - \delta p_1 - \delta p_2 + 2p_1p_2 \\ &\quad +2\delta^2 - \delta^3 + \delta^2 p_1 + \delta^2 p_2 + 2.25 \\ b_1 &= p_1 - 4\delta + p_2 - 2\delta p_1 - 2\delta p_2 + 3\delta^2 + 2 \\ b_2 &= p_1 - 3\delta + p_2 + 2. \end{aligned} \tag{17}$$

La table de Routh associée à ce polynôme est donnée par

1	b_1
b_2	b_0
$b_1b_2 - b_0$	0
b_0	0

Le polynôme est stable si tous les éléments de la première colonne sont stable, c'est-à-dire

$$\begin{pmatrix} b_2 \\ b_1b_2 - b_0 \\ b_0 \end{pmatrix} \geq 0. \tag{18}$$

ou encore

$$P(s-\delta,\mathbf{p}) \text{ instable } \Leftrightarrow r_{\min}(\mathbf{p},\delta) \leq 0 \tag{19}$$

avec

$$r_{\min}(\mathbf{p},\delta) = \min\left(b_2, b_1b_2 - b_0, b_0\right). \tag{20}$$

Le degré de stabilité s'écrit alors

$$\delta^*(\mathbf{p}) = \min_{\substack{\delta \geq 0 \\ r_{\min}(\mathbf{p},\delta) \leq 0 \\ r_{\min}(\mathbf{p},0) \geq 0}} \delta. \tag{21}$$

La caractérisation de la fonction $\delta^*(\mathbf{p})$ est donc un problème de lancé de rayon. Nous pouvons réutiliser les algorithmes de la partie IV D. pour évaluer $\delta^*(\mathbf{p})$ en considérant les correspondances suivantes :

$$\begin{array}{ccc} \text{Lancé de rayon} & & \text{Degré de stabilité} \\ d & \longleftrightarrow & \delta \\ g & \longleftrightarrow & r \end{array}. \tag{22}$$

Si on prend le pavé $[\mathbf{p}] = [-3,7] \times [-3,7]$, on trouve la figure suivante (30 secondes de calcul, avec le même programme et les mêmes outils que pour le lancé de rayon) :

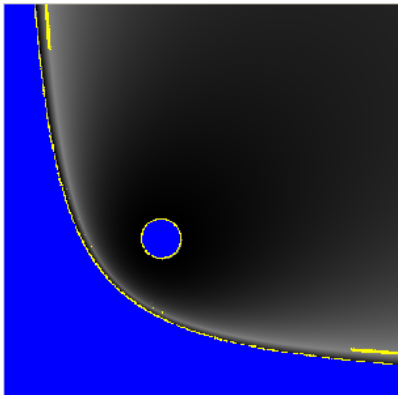

Image 4 : Degré de stabilité δ d'un système linéaire invariant (bleu : instable, jaune : indéterminé, noir à blanc : δ allant de 0 à 1).

VI. Conclusion

Dans cet article, nous avons montré que nous pouvions résoudre avec la même approche ensembliste deux problèmes a priori différents : le lancé de rayon et le calcul du degré de stabilité d'un système linéaire invariant. L'algorithme commun est basé sur une inversion ensembliste, utilisant le calcul par intervalles.

Références

[1] L. Jaulin. *Solution globale et garantie de problèmes ensemblistes ; Application à l'estimation non linéaire et à la commande robuste.* PhD thesis, Université Paris XI Orsay, 1994.

[2] S. Bazeille. *Vision sous-marine monoculaire pour la reconnaissance d'objets.* PhD thesis, Université de Bretagne Occidentale, 2008.

[3] J. Flórez. *Improvements in the ray tracing of implicit surfaces based on interval arithmetic.* PhD thesis, Universitat de Girona, 2008.

[4] L. Jaulin, M. Kieffer, O. Didrit et E. Walter, *Applied interval analysis*, Springer-Verlag, London, Great Britain, 2001.

[5] L. Jaulin, E. Walter, O. Lévêque et D. Meizel, « Set inversion for chi-algorithms, with application to guaranteed robot localization », *Math. Comput Simulation*, 52, pp. 197–210, 2000.

[6] J. Ackermann, « Does it suffice to check a subset of multilinear parameters in robustness analysis ? », *IEEE Transactions on Automatic Control*, 37(4), pp. 487–488, 1992.

[7] J. Ackermann, H. Hu et D. Kaesbauer, « Robustness analysis : a case study », *IEEE Transactions on Automatic Control*, 35(3), pp. 352–356, 1990.