Calculs et visualisation en nombres complexes

Laurent Testard

▶ **To cite this version:**

Laurent Testard. Calculs et visualisation en nombres complexes. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1997. Français. NNT : . tel-00004965

HAL Id: tel-00004965
https://theses.hal.science/tel-00004965

Submitted on 20 Feb 2004

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

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

THÈSE

présentée par

Laurent TESTARD

pour obtenir le grade de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 30 mars 1992)

(spécialité : **Mathématiques Appliquées**)

Calculs et visualisation en nombres complexes

Date de soutenance : 27 novembre 1997.

Composition du Jury :

F. ROBERT (président)
J.-C. YAKOUBSOHN (rapporteur)
D. BECHMANN (co-rapporteur)
P. SCHRECK (co-rapporteur)
J. DELLA DORA (examinateur)
F. JUNG (examinateur)

Thèse préparée au sein du Laboratoire LMC–IMAG.

Table des matières

Bibliographie

[Ahl66] Ahlfors (L. V.). – *Complex Analysis.* – McGraw-Hill, 1966, second édition.

[Aid96] Aid (R.). – *Estimation de l'erreur globale pour l'intégration numérique d'équations différentielles ordinaires.* – Rapport technique n° 159, Grenoble, LMC-IMAG, Mars 1996.

[Aid99] Aid (R.). – *Unkown PHD thesis.* – Thèse de PhD, INPG, 1999.

[AL97] Aid (R.) et Levacher (L.). – Numerical investigations on global error estimation for ordinary differential equations. *Journal of Computational and Applied Mathematics*, 1997.

[Asl96] Aslaksen (H.). – Multiple-valued complex functions and computer algebra. *SIGSAM Bulletin*, 1996.

[ATV97] Aïd (R.), Testard (L.) et Villard (G.). – Global error visualization. *In : SCAN97.*

[AvLS96] Abbot (J.), van Leeuwen (A.) et Strotmann (A.). – *Objectives of OpenMath.* – Rapport technique n° TR 12, RIACA, 1996.

[Bea84] Beardon (A. F.). – *A Primer on Riemann Surfaces.* – Cambridge University Press, 1984.

[Ber97] Beringer (F.). – *Recherche des cycles limites d'un système dynamique polynomial plan, intégration dans LAMEX.* – Rapport de DEA, LMC, IMAG, 1997.

[BK86] Brieskorn (E.) et Knorrer (H.). – *Plane Algebraic curves.* – Birkhauser, 1986.

[BO78] Bender (C. M.) et Orszag (S. A.). – *Advanced Mathematical Methods for Scientists and Engineers.* – McGraw-Hill, 1978.

[Boy93] Boyd (J. P.). – Chebyshev and legendre spectral methods in algebraic manipulation languages. *Journal of Symbolic computation*, 1993.

[Car61] Cartan (H.). – *Théorie élémentaire des fonctions analytiques d'une ou plusieurs variables complexes.* – Hermann, 1961.

[Cha95] Chaffy (C.). – The analytic continuation process : from computer algebra to numerical analysis. *In : ISSAC'95.* pp. 216–222. – ACM.

[CHQZ88] Canuto (C.), Hussaini (M. Y.), Quarteronu (A.) et Zang (T. A.). – *Spectral methods in fluid dynamics.* – Springer Verlag, 1988.

[CJ96] Corless (R. M.) et Jeffrey (D. J.). – Editor's corner : The unwinding number. *SIGSAM Bulletin*, 1996.

[CM84] Crouzeix (M.) et Mignot (A. L.). – *Analyse numérique des équations différentielles.* – Masson, 1984.

[CNP93] Candelpergher (B.), Nosmas (J. C.) et Pham (F.). – *Approche de la résurgence.* – Hermann, 1993.

[CS91] Comer (D. E.) et Stevens (D. L.). – *Internetworking with TCP-IP : 2 : design, implementation and internals.* – Prentice-Hall, 1991.

[CS93] Comer (D. E.) et Stevens (D. L.). – *Internetworking with TCP-IP : 3 : client-server programming and applications.* – Prentice-Hall, 1993.

[dd94] (documentation developpeur). – *pthreads and Solaris threads : A comparison of two user level threads APIs.* – Rapport technique, SunSoft, 1994.

[Dem89] Demazure (M.). – *Catastrophes et Bifurcations.* – Ellipses, 1989.

[Die62] Dieudonné (J.). – *Éléments d'analyse - tome 1.* – Gauthier-Villars, 1962.

[Die68] Dieudonné (J.). – *Calcul infinitésimal.* – Hermann, 1968.

[DJ95] Dicrescenzo (C.) et Jung (F.). – The COMPASS Package. *In : Cathode Proceedings, Nijmegen, 1995.*

[Fed87] Fedoriouk (M.). – *Méthodes asymptotiques pour les équations différentielles ordinaires linéaires.* – Mir, 1987.

[FK80] Farkas (H. M.) et Kra (I.). – *Riemann Surfaces.* – Springer Verlag, 1980.

[For81] Forster (O.). – *Lectures on Riemann Surfaces.* – Springer-Verlag, 1981.

[Ged77] Geddes (K.). – Symbolic computation of recurrence relations for the chebyshev series solution of linear ode. *In : MACSYMA user's conference.*

[GGM93] Goodman (M. A.), Goyal (M.) et Massoudi (R. A.). – *Solaris Porting Guide.* – SunPress, 1993.

[GOP+96] Gunn (C.), Ortmann (A.), Pinkall (U.), Polthier (K.) et Schwarz (U.). – Oorange : A Virtual Laboratory for Experimental Mathematics. *http://www-sfb288.math.tu-berlin.de/oorange*, 1996.

[Hen62] Henrici (P.). – *Discrete Variable Methods in Ordinary Differential Equations.* – Wiley, 1962.

[HLL97] Hesselink (L.), Levy (Yuval) et Lavin (YingMei). – The topology of symmetric second-order 3d tensor fields. *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, n° 1, janvier 1997.

[HNW87] Hairer (E.), Norsett (S.P.) et Wanner (G.). – *Solving Ordinary Differential Equations I Nonstiff Problems.* – Springer-Verlag, 1987.

[HW95] Hubbard (J. H.) et West (B.H.). – *Differential Equations : A Dynamical Systems Approach.* – Springer-Verlag, 1995.

[Ise96] Iserles (A.). – *A First Course in the Numerical Analysis of Differential Equations.* – Cambridge University Press, 1996.

[JT92] Jenks (R. D.) et Tutor (R. S.). – *AXIOM, The Scientific Computation System.* – Springer-Verlag, 1992.

[Kil96] Kilgard (M. J.). – *Programming OpenGL for the X Window System.* – Addison-Wesley, 1996.

[Kil97] Kilgard (M. J.). – Realizing OpenGL: Two implementations of One architecture. *In : SIGGRAPH 97.*

[NDW93] Neider (J.), Davis (T.) et Woo (M.). – *OpenGL(tm) Programming Guide.* – Addison-Wesley, 1993.

[Ope97] OpenMath. – The OpenMath Home Page. – http://www.openmath.org, 1997.

[OQL88] O'Reilly (T.), Quercia (V.) et Lamb (L.). – *The definitive guides to the X WINDOW system : 3 : X WINDOW system.* – O'Reilly and associates, 1988.

[Ous94] Ousterhout (J. K.). – *Tcl and the Tk toolkit.* – Addison-Wesley, 1994.

[Pat96] Patton (C. M.). – A representation of branch-cut information. *SIGSAM Bulletin*, 1996.

[Reb96] Rebillard (L.). – *Séries de Chebyshev formelles, RR 960 M.* – Rapport technique, IMAG, septembre 1996.

[Ric88] Richard (F.). – Computer graphics and complex ordinary differential equations. *New Trends in Computer Graphics*, 1988.

[Riv97] Rivière (M.). – *Concepts structurants pour la mise en oeuvre d'applications irrégulières : application au support exécutif parallèle Athapascan0$_{mp}$.* – Thèse de PhD, INPG, 1997.

[RJ91] Richard-Jung (F.). – *Le phénomène de Stokes en image - RT 65.* – Rapport technique, LMC-Imag, 1991.

[SA96] Segal (M.) et Akeley (K.). – *The OpenGL(tm) Graphics System : A specification (version 1.1).* – Rapport technique, Silicon Graphics, 1996.

[SGM97] SGML. – The SGML Open Home Page. – http://www.sgmlopen.org, 1997.

[SH96] Stuart (A. M.) et Humphries (A. R.). – *Dynamical Systems and Numerical Analysis.* – Cambridge University Press, 1996.

[Sha94] Shampine (L.). – *Numerical Solution of Ordinary Differential Equations.* – Chapman and Hall, 1994.

[Ske86] Skeel (R. D.). – Thirteen ways to estimate global error. *Numer. Math.*, vol. 48, 1986, pp. 1–20.

[Spr57] Springer (G.). – *Introduction to Riemann Surfaces.* – Addison-Wesley, 1957.

[Ste90] Stevens (W. Richard). – *UNIX Network Programming.* – Prentice-Hall Software Series, 1990.

[Str95] Stroustrup (B.). – *The C++ programming language – second edition.* – Addison-Wesly, 1995.

[SW76] Shampine (L. F.) et Watts (H. A.). – Global error estimation for ordinary differential equations. *ACM Trans. Math. Softw.*, vol. 2, 1976, pp. 172–186.

[Tan92] Tanenbaum (A.). – *Modern Operating Systems.* – Prentice-Hall, 1992.

[Tes93] Testard (L.). – *Calcul automatique des multiplicateurs de Stokes.* – rapport de dea, LMC–IMAG, 1993.

[Tes97] Testard (L.). – Visualization of complex ode solutions. *In: Visualization and Mathematics 1997.*

[Was76] Wasow (W.). – *Asymptotic expansions for ordinary differential equations.* – Robert E. Krieger publishing co., 1976.

[Wel95] Welch (B.). – *Practical Programming in Tcl and Tk.* – Prentice-Hall, 1995.

[Wey55] Weyl (Hermann). – *The concept of a Riemann Surface.* – Addison-Wesley, 1955.

[Zad76] Zadunaisky (P. E.). – On the estimation of error propagated in the numerical solution of a system of ordinary differential equations. *Num. Math.*, vol. 27, 1976, pp. 21–39.

Remerciements

Mieux vaut tard que jamais, voici enfin les remerciements que je tenais à faire, et couvrant l'ensemble de ma période de thèse.

Je tiens tout d'abord à remercier François Robert qui a eu l'amabilité d'accepter d'assumer la lourde tâche de président du jury. Outre la gentillesse du personnage, j'ai particulièrement apprécié les conseils personnalisés qui m'ont été prodigués, et je tiens aussi à m'excuser pour les séquelles des écrasages de main qui pourraient être la conséquence de mes accès de stress pré-soutenance.

Je tiens aussi à remercier les personnes qui ont bien voulu rapporter mon manuscrit de thèse, car je pense réellement qu'ils ont du fournir des efforts importants pour comprendre ce que je n'ai su exprimer clairement. Je les remercie en particulier pour le nombre et la qualité des remarques qu'ils ont pu faire sur ce manuscrit. Faire la connaissance de Jean-Claude Yakoubsohn a été pour moi faire la connaissance d'une personne très ouverte, dont les remarques judicieuses m'ont permis de comprendre que le sujet d'une thèse, si exotique soit-il, s'inscrit dans une cadre plus général, en particulier d'un point de vue mathématique. Merci beaucoup ! Je tiens à remercier aussi Pascal Schreck dont la patience et la gentillesse m'ont permis de m'expliquer de vive voix sur des passages obscurs de ma thèse, possibilité qui n'est probablement pas offerte à tous les thésards par leurs rapporteurs. Je ne sais pas si ces explications ont débouché sur des éclaircissements, mais en tous les cas, je tenais à dire que j'avais apprécié la pertinence ainsi que l'aspect très technique des questions qui m'ont été posées pendant ma soutenance. Je voulais aussi remercier Mme Dominique Bechmann d'avoir bien voulu cosigner le rapport de thèse.

Enfin, mes remerciements vont à mes deux encadreurs de thèse. Jean Della Dora a bien voulu me laisser aller jusqu'au bout de mes convictions, notamment en ce qui concerne les choix mathématiques que j'ai du faire pour mener à bien mon travail personnel de recherche. Françoise Jung a été l'instigatrice des directions de recherche adoptées dans ce manuscrit, et je tiens à la remercier tout particulièrement pour son travail de correction de certaines parties de ce manuscrit qui, sans sa rigueur, n'aurait pas sa forme actuelle.

L'équipe Calcul Formel du LMC, petite par la taille, possède néanmoins quelques personnes avec qui j'ai pu avoir le plaisir de travailler ou tout simplement de discuter. En plus des personnes pré-citées, je tiens à remercier l'ensemble de l'équipe pour son assiduité à mes séminaires, et ce même lorsque le sujet n'avait pas grand chose à voir avec leurs intérêts. Claire Di Crescenzo, en acceptant de bien vouloir mettre le nez dans l'immense volume de code que j'ai pu produire, m'a été d'un très grand secours : en tant qu'utilisateur numéro un, dont les remarques pertinentes m'ont permis de stabiliser le fameux environnement graphique GANJ, ainsi qu'en tant que personne humaine, avec tout le dévouement et la sympathie qui la caractérise. Je remercie ici aussi Gilles Villard qui a permis de fusionner, le temps d'un papier, les travaux de plusieurs personnes pour en faire un tout cohérent.

Pour terminer l'aspect "collaborations dans le travail" de ces remerciements, je voulais tout d'abord remercier René Aïd, dont la collaboration a été extrèmement précieuse pendant tout le temps où nous avons partagé le même bureau. De plus, j'aimerais lui exprimer ma reconnaissance quant à son soutien quasi permanent, notamment lorsque nous avons *véritablement* collaboré, qui fait bien vite oublier tous les "aspects irritants" (la formule n'est pas de moi) de son caractère. Enfin, en tant qu'ami personnel, je tiens à lui affirmer ma plus grande reconnaissance. Michel Rivière est lui aussi un ami avant d'être un collègue de travail, et il est difficile de dire à quel point les quelques semaines où nous avons travaillé ensemble ont été pour moi précieuses. Thierry Gautier, par sa maîtrise technique, par ses facilités à réfléchir efficacement, ainsi que par ses qualités humaines, est vraiment un personnage très précieux dès qu'il s'agit de discuter, sciences ou autres. Enfin, je tenais à remercier Yann Macutan, dont les capacités mathématiques et le sens de l'écoute m'ont permis de cerner avant qu'il ne soit trop tard les points de théorie suffisants pour la rédaction de cette thèse. J'ai particulièrement apprécié ses explications éclairées, concernant l'analyse complexe en particulier. Ensuite, je tenais à remercier en vrac tous ceux qui se sont penchés plus d'une minute sur mes travaux, et qui ont donc leur part de responsabilité dans l'accomplissement de cette thèse : merci donc à Loïc Benayoun, à Stéphane Boucherau, et je tiens à apporter une mention spéciale à Evelyne Hubert, qui a été une des premières à bien vouloir jouer le jeu de la visualisation de surfaces avec moi, avec qui j'ai eu le bonheur (bon d'accord pas tous les jours) de partager le même bureau.

Parmi les divers amis que j'ai pu me faire dans ce laboratoire, je tenais à citer, dans le désordre, Luc, Loïc, Rico, Manu, avec qui les journées devant les machines se terminent à grands coups de bombres, électroniques bien sûr, à l'indémodable Eckhard, avec sa bonne humeur quasi-permanente, à Frédéric, à Yann, Gabriel, Josselin et Claude-Pierre, sans oublier Gérard, ainsi que Marie-Pierre !

Ensuite, je tiens à remercier tous ceux qui suivent pour leur amitié constante depuis des années, ainsi que pour leur soutien durant la fin de thèse : Olaf, Jo, Fab, Heather, Mathieu et Myriam, et je pourrais certainement en citer d'autres, je leur prie de bien vouloir m'excuser. Merci aussi à Pierre-Yves qui a bien voulu se déplacer jusqu'à l'INPG pour ma soutenance, et à Patrick, Jean Marc, dont la présence à mon pot de thèse était apréciable (et je suis sur que j'en oublie)...

Je tiens enfin à remercier ma mère et ma soeur, qui m'ont aidé à supporter de loin mes trois ans de thèse, et particulièrement mon père qui a bien voulu assister en direct à mes élucubrations.

Je tiens surtout à remercier Nathalie pour TOUT.

Introduction

Ce document est la concrétisation de trois ans de travail, axé principalement sur deux centres d'intérêts : les calculs en nombres complexes d'une part, et d'autre part la visualisation d'objets issus de l'analyse complexe, plus particulièrement adaptée aux solutions d'équations différentielles complexes. Les motivations de ce travail sont à l'origine d'ordre mathématique : les problèmes de résolutions d'équations différentielles dans le plan complexe et de calculs avec des fonctions multiformes rencontrés dans [Tes93] se sont bien vite transformés en besoins d'expérimentations variées. Le premier problème rencontré a été de trouver des outils pour effectuer des opérations élémentaires dans des domaines tels que l'évaluation de fonctions de la variable complexe, la visualisation de ces fonctions, la spécification de chemins, la vérification des calculs, etc... Malheureusement, ce genre d'outils n'est pas actuellement très fréquent dans les principaux systèmes de calculs, que ceux-ci soient formels ou numériques.

La partie I de ce document présente l'origine mathématique des différents problèmes rencontrés, que ceux-ci soient du domaine du calcul avec des nombres complexes (chapitre 1), ou du domaine des équations différentielles de la variable complexe (chapitre 2). En ce qui concerne les calculs, bien que les systèmes de calcul formel ou numérique s'enrichissent de jour en jour de fonctionnalités nouvelles, le plan complexe n'est représenté de manière interne à ces systèmes que comme une transcription "simpliste" de $\mathbb{R}^2$, ce qui est une grande source de difficultés. Dès que le stade des évaluations de formules simples est dépassé, les résultats sont généralement incomplets. Par exemple, la détermination continue d'une fonction multiforme reste encore un problème difficile, dont le traitement, même dans les cas les plus simples, est impossible de manière automatique dans les systèmes de calcul actuels. De même, les méthodes de visualisation des fonctions et autres objets issus de l'analyse complexe sont la plupart du temps implémentées de manière trop générale dans ces systèmes pour qu'une exploitation efficace en soit possible à peu de frais de programmation supplémentaire. Les méthodes de visualisation étant de plus basées au-dessus des méthodes de calcul, la visualisation des fonctions nécessitant des calculs évolués reste encore hors de portée d'un utilisateur quelconque.

La démarche proposée tout au long de ce document est à base d'expérimentations : le travail de programmation que cela a demandé a été particulièrement important, et nous avons essayé de formaliser ce qui a été implémenté, afin que ces expériences puissent être réutilisées par la suite, éventuellement par d'autres utilisateurs. Nous proposerons dans le chapitre 3 un environnement d'expérimentations, qui sera appliqué par la suite aux expériences issues de l'analyse complexe, mais qui pourrait facilement être exploité dans d'autres domaines du calcul scientifique. Le principe de base que nous allons décrire est celui d'une plate-forme d'expérimentations, et nous avons appliqué des notions classiques (en termes de modèles de programmation) aux problèmes de calcul scientifique. Il s'agit, dans un premier temps, de fournir un environnement *homogène* permettant de coupler calculs et visualisation. Ainsi, les interactions entre modules de calculs et modules de visualisation ont été complètement spécifiées. Cela signifie en particulier que des modules de calculs intégrés dans une telle plate-forme disposent de méthodes permettant de visualiser rapidement les résultats produits.

La partie II présente un environnement de calculs en nombres complexes qui a été effectivement réalisé, et qui permet de gérer des notions telles que les indéterminations (chapitre 4), indéterminations qui peuvent se produire lors d'évaluations de fonctions simples, telles que les racines carrées, les logarithmes, ... Notre approche a été de considérer ces phénomènes comme des phénomènes *normaux*, et donc d'essayer d'en tirer un maximum d'informations en vue d'un éventuel traitement spécifique à un problème. Nous proposons de plus dans ce chapitre des outils logiciels inédits pour résoudre ces indéterminations : le principe de programmation exposé dans le chapitre 4 permet de séparer la partie déterminée d'un calcul du mécanisme de résolution des indéterminations. Le chapitre 5 montre une application des deux chapitre précédents, qui consiste à intégrer numériquement des équations différentielles ordinaires de la variable complexe définies par des fonctions dont le comportement peut être de nature multiforme. Le but de ce chapitre, sans trop rentrer dans la théorie, est de montrer que les environnements décrits précédemment peuvent, sans trop de travail spécifique, être utilisés efficacement pour résoudre des problèmes d'indéterminations dont une solution peut être donnée par la connaissance de données spécifiques à un problème, comme cela est le cas dans le cadre des équations différentielles ordinaires. Cette connaissance permet ainsi de résoudre des problèmes d'indéterminations, problèmes pouvant avoir des répercussions non négligeables sur les expériences les ayant provoqués.

La visualisation des différents objets manipulés est présente dans tous les aspects de ce document, et pour cela l'outil graphique qui a été réalisé pour mettre en place les expériences de visualisation mérite bien la totalité d'une partie de ce document, la partie III.

Les environnements complets de visualisation commerciaux se développent de jour en jour (les plus importants étant probablement AVS et Iris Explorer), et sont pour la plupart des environnements modulaires de visualisation. Ils offrent ainsi de nombreuses facilités pour être développés de façon modulaire, et ils permettent d'être couplés à n'importe quel code de calcul. Ces différentes caractéristiques les rendent

particulièrement adaptés à la visualisation d'objets mathématiques. Leur seul problème (de taille) est qu'ils sont payants. De plus, pour être exploités pleinement, ces environnements nécessitent des stations de travail accélérées matériellement pour le graphique. Le besoin d'une solution générique, gratuite et facile à utiliser est donc bien réel. La réalisation de l'environnement graphique GANJ a été particulièrement motivée par sa souplesse d'utilisation et sa facilité de manipulation. Son fonctionnement général est expliqué dans le chapitre 6, chapitre qui, avec le chapitre 7, peut sembler assez *technique* pour certains lecteurs. Ces deux chapitres peuvent être parcourus rapidement, car le plus important de cette partie en ce qui concerne l'exploitation graphique de résultats mathématiques se trouve dans le chapitre 8, qui présente les différentes interfaces de programmation qui ont été conçues afin de faciliter le travail de visualisation.

La dernière partie de ce document (la partie IV) présente quelques applications. Tout d'abord, le chapitre 9 montre l'implémentation des chemins dans le plan complexe, qui a été intégrée à la fois dans l'environnement de calcul du chapitre 4 et dans l'environnement graphique de la partie III. Étant donné que les chemins dans le plan complexe sont les objets de base de l'analyse complexe, l'implémentation qui en a été faite est particulièrement soigneuse. Après avoir précisé les conditions d'expérimentation et montré comment les environnements définis précédemment peuvent être appliqués dans ce cadre, le chapitre 10 montre quelques expériences sur la visualisation des solutions d'équations différentielles complexes. Les méthodes de visualisation seront illustrées par de nombreux exemples, puis le problème de l'étude des comportements multiformes des solutions des équations différentielles sera lui aussi abordé. Il est à noter que ce chapitre a fait l'objet d'une communication à la conférence "Visualisation and Mathematics, VISMATH97" [Tes97], et qu'il donnera lieu à publication. Enfin, le dernier chapitre (le chapitre 11) montrera une application à la visualisation de l'erreur globale commise pendant une intégration d'une équation différentielle de la variable complexe. Ce chapitre expose non seulement un formalisme de représentation inédit de l'erreur globale qui permet son interprétation visuelle et sa localisation dans le plan complexe, mais il montre aussi un environnement expérimental permettant de prendre en compte les données précédentes afin d'améliorer les résultats de l'intégration, grâce notamment à des intéractions avec l'utilisateur de l'environnement expérimental. Ce travail a fait l'objet d'une communication à SCAN97[ATV97].

Ce manuscrit comporte des planches en couleurs. Malheureusement, il n'a pas été possible d'en tirer tous les exemplaires en couleurs, et si jamais vous avez entre les mains la version noir et blancs, les planches couleurs devraient être disponibles de manière électronique, probablement accessibles à partir des pages de l'équipe.

Première partie

Nombres complexes et équations différentielles

Chapitre 1

Quelques notions d'analyse complexe

1.1 Fonctions de la variable complexe

1.1.1 Fonctions analytiques, fonctions méromorphes

Nous commencerons par introduire les *fonctions analytiques*, qui sont les fonctions de base de l'analyse complexe.

1.1.1.1 Définition (Fonction analytique [Die68]) Soit D un ouvert dans $\mathbb{C}$. On dit qu'une fonction complexe $f : D \Leftrightarrow\rightarrow \mathbb{C}$ est *analytique* dans D si, pour tout point $z_0 \in D$, il existe un disque ouvert $\Delta : |z \Leftrightarrow z_0| < r$ contenu dans D tel que l'on ait dans ce disque

$$f(z) = \sum_{n=0}^{\infty} c_n (z \Leftrightarrow z_0)^n$$

où le second membre est une série entière en $z \Leftrightarrow z_0$, convergente dans Δ.

Nous introduisons aussi les *fonctions holomorphes* :

1.1.1.2 Définition (Fonction holomorphe [Car61]) Une fonction de la variable complexe f est *holomorphe* dans D si, en tout point $z_0 \in D$, la quantité

$$\frac{f(z_0 + h) \Leftrightarrow f(z_0)}{h}$$

possède une limite lorsque h tend vers 0 (h nombre complexe non nul).

Une fonction de la variable complexe est holomorphe si et seulement si elle est analytique[Car61]. La définition 1.1.1.2 permet de percevoir les fonctions analytiques comme des fonctions de deux variables réelles (parties réelle et imaginaire), mais qui se comportent, du point de vue de la dérivation, comme des fonctions d'une seule variable. Cette propriété est obtenue en considérant une version particulière des *conditions de Cauchy* :

1.1.1.3 Theorème (Conditions de Cauchy[Car61]) Soit f une fonction de la variable complexe. Étant donné $x = x_1 + ix_2$, nous pouvons noter $f(x) = \tilde{f}(x_1, x_2)$.

Pour que f soit holomorphe en un point, il faut et il suffit que la condition

$$\frac{\partial f}{\partial x_1} + i\frac{\partial f}{\partial x_2} = 0$$

soit vérifiée en ce point.

En écrivant $x_1 = \frac{1}{2}(z + \bar{z})$ et $x_2 = \frac{-1}{2}(z \Leftrightarrow \bar{z})$ nous pouvons aussi exprimer cette condition en fonction de z et $\bar{z}$:

$$d\tilde{f} = \frac{\partial f}{\partial z}dz$$

ou encore

$$\frac{\partial f}{\partial \bar{z}} = 0$$

Ainsi, une fonction est holomorphe si sa différentielle peut s'écrire en fonction d'une seule variable. Ceci peut expliquer l'intérêt des fonctions holomorphes en analyse complexe, qui bien que représentant une classe *réduite* de fonctions à étudier, permettent néanmoins de considérer les fonctions de la variable complexe qui se dérivent comme des fonctions d'une seule variable.

1.1.1.4 Définition (Fonction Méromorphe) On définit l'ensemble des fonctions *méromorphes* sur un domaine D par le corps des fractions de l'anneau intègre des fonctions analytiques sur D.

Soit $f = \frac{g}{h}$ une fonction méromorphe sur un domaine D. Pour $x_0 \in D$, si $g(x_0) \neq 0$, on dit que x_0 est un *pôle* de f si $h(x_0) = 0$.

1.1.1.5 Définition (Série de Laurent [Car61]) Soit $x_0 \in \mathbb{C}$. On appelle *série de Laurent* au voisinage de x_0 une série de la forme

$$\sum_{-\infty}^{\infty} a_i(x \Leftrightarrow x_0)^i$$

Si f est une fonction méromorphe sur un domaine D, alors f est développable en série de Laurent au voisinage de tous les points de D [Car61]. De plus, la partie principale des développements en série de Laurent (c'est-à-dire les termes dans l'expression d'une série de Laurent correspondant à des exposants i négatifs) a toujours un nombre fini de termes. Au voisinage d'un pôle, ce nombre de termes est appelé *l'ordre* de ce pôle. Au voisinage d'un point de D qui n'est pas un pôle, la partie principale est nulle.

Exemples de visualisation de fonctions analytiques et méromorphes.

De nombreux outils existent pour visualiser les fonctions méromorphes (en particulier celles qui sont analytiques), aussi bien expliqués dans la littérature qu'implémentés dans les principaux systèmes de calculs, formels ou numériques. Il faut préciser que ces fonctions ne sont pas très difficiles à visualiser, comme le montrent les exemples de modes de visualisation suivants :

- Les représentations graphiques traditionnelles, telles que des représentations en partie réelle/partie imaginaire, en module/argument, ou alors avec des contours représentant les courbes iso-modules : cela revient essentiellement à considérer les fonctions analytiques comme des fonctions de deux variables réelles à valeur dans $\mathbb{R}^2$, et donc à appauvrir la portée de ces objets mathématiques (c'est-à-dire à oublier tout ce qui en fait leur spécificité, comme par exemple les conditions de Cauchy).
- Des représentations plus élaborées, sur un seul graphe, qui permettent d'avoir une visualisation plus globale de la fonction analytique. Ces représentations sont basées sur le plan, qui représente les valeurs prises par la variable complexe, et en altitude sont dessinées les modules pris par la fonction, avec un codage de couleur qui représente un échantillonage de l'intervalle $[0, 2\pi[$, permettant de visualiser l'argument de la fonction représentée. Une telle méthode est intégrée aux routines graphiques de Maple, d'Axiom[JT92], ...

Nous allons maintenant étudier des fonctions plus "complexes", au sens où généralement il est difficile d'obtenir des évaluations de ces fonctions en certains points, ou alors d'en avoir des évaluations de manière globale. Ces comportements numériques nuisent en particulier à la visualisation de ces fonctions.

1.1.2 Singularités

Nous avons pu comprendre dans le chapitre précédent que l'étude des fonctions de la variable complexe passait par une caractérisation des comportements de ces fonctions au voisinage de certains points. D'où l'importance de *classifier* les différents comportements possibles des fonctions de la variable complexe, au voisinage de certains points. La notion de *singularité* d'une fonction, attachée à un point du plan complexe, permet de relier le comportement de la fonction au point autour duquel ce comportement peut être observé.

1.1.2.1 Définition (Singularités [Bea84]) Considérons f une fonction, et x_0 un point de $\mathbb{C}$.

- Si f admet un développement en série de Taylor au voisinage de x_0, alors x_0 est appelé *point régulier*.
- Si f admet un développement en série de Laurent au voisinage de x_0, avec N_0 termes en puissances négatives, alors x_0 est un *pôle d'ordre* N_0.

– Si f n'admet aucun des développements en série mentionnés ci-dessus, alors x_0 est une *singularité essentielle* de f.

Les fonctions possédant des singularités essentielles présentent de nombreuses difficultés à qui veut visualiser leur comportement au voisinage de ces points. Un théorème dû à Weïerstrass (présenté dans [Ahl66]) montre qu'au voisinage d'une singularité essentielle, une fonction s'approche arbitrairement près de n'importe quel nombre complexe (c'est-à-dire que l'image par la fonction de n'importe quel voisinage d'une singularité essentielle est dense dans $\mathbb{C}$). Cela induit des difficultés de calcul (la fonction admet un comportement tourbillonant, ce qui pose de nombreux problèmes d'instabilité numérique), ainsi que de visualisation. Afin d'étudier certains types de singularités essentielles, nous pouvons introduire les développements en *séries de Puiseux*.

1.1.2.2 Définition (Série de Puiseux) Si on note $\mathbb{C}[[X]]$ l'anneau des séries formelles à coefficients dans $\mathbb{C}$, et $\mathbb{C}((X))$ son corps des fractions, on peut alors définir

$$\mathbb{C}^*((X)) = \bigcup_{q \in \mathbb{N}^*} \mathbb{C}((X^{1/q}))$$

Un élément de ce corps sera une série formelle de la forme

$$\sum_{i=-N_0}^{\infty} a_i X^{i/q}$$

Si une fonction admet comme développement au voisinage d'un point x_0 la série

$$\sum_{i=-N_0}^{\infty} a_i (x \Leftrightarrow x_0)^{i/q}$$

où le nombre de termes ayant un exposant négatif est fini ($N_0 \in \mathbb{N}$), on dit que x_0 est un point de branchement de degré q.

Cette définition peut être généralisée à l'aide des *paires de Puiseux*[BK86], qui permettent de mélanger plusieurs comportements, chacun étant associé à une puissance.

L'intérêt des fonctions qui admettent un développement en série de Puiseux au voisinage d'un point de branchement est qu'elles bénéficient de facilités de calculs et de visualisation, par rapport aux fonctions admettant d'autres types de singularités essentielles. Le comportement des fonctions au voisinage des points de branchement peut être expliqué par des méthodes algébriques[BK86], ce qui donne lieu à des visualisations intéressantes.

Évidemment, les séries de Puiseux n'ont un intérêt que limité dans le domaine du calcul numérique des valeurs d'une fonction, puisqu'elles font intervenir des fonctions puissances qui posent d'importants problèmes de *détermination*[Spr57] (c'est-à-dire que l'évaluation d'une telle fonction peut mener à *plusieurs* résultats, sans qu'il soit possible de décider lequel de ces résultats est celui de l'évaluation de la fonction). Cependant, leurs motivations algébriques en font un outil très puissant pour l'étude de fonctions de la variable complexe au voisinage de points de branchement.

Visualisation de fonctions admettant un point de branchement.

Si nous considérons un cercle (paramétré par un réel $t \in [0,1]$) centré en x_0, point de branchement de la fonction f, nous pouvons donner à partir des valeurs de la fonction sur ce cercle un mode de visualisation qui permet de tirer des renseignements algébriques sur le comportement de la fonction au voisinage de son point de branchement : en effet, la courbe image de cercle est un noeud, et les caractéristiques topologiques de ce noeud permettent de retrouver les caractéristiques algébriques de la fonction au voisinage du point de branchement [BK86].

Cette méthode ressemble à méthode décrite dans [Ric88] qui permet de visualiser le graphe d'une fonction de la variable complexe lorsque cette variable est astreinte à rester sur un chemin du plan complexe. En particulier, la fonction peut être étudiée au voisinage d'une singularité. La correspondance entre x et $f(x)$ se fait sur la courbe image en codant, par une couleur notamment, les valeurs de x utilisées. La même couleur est affectée à x et à la valeur de $f(x)$ calculée.

Nous allons exposer cette méthode en considérant que nous connaissons le degré du branchement de la fonction, mais nous verrons par la suite que cela n'est pas forcément nécessaire. Cette méthode peut se décomposer en quatre étapes :

1. Évaluation de la fonction sur le cercle : cela ne va pas sans poser de problèmes de calculs, nous proposerons dans la partie II des méthodes effectives de calculs de fonctions au voisinage de points de branchements. Pour l'instant, l'évaluation de la fonction revient à choisir une *détermination* de la fonction, et à calculer les valeurs de cette détermination sur le cercle. On obtient ainsi, pour chaque valeur du paramètre t, un nombre complexe $x_1(t) + ix_2(t)$.
2. Construction d'une courbe tri-dimensionnelle en affectant à chaque valeur calculée une altitude correspondant à la valeur du paramètre t :
$$\{(x_1(t), x_2(t), t), t \in [0,1]\}$$
3. Évaluation sur les autres branches de la fonction (c'est-à-dire le calcul des différentes déterminations que la fonction peut posséder sur le cercle) : on obtient alors une *tresse*. Cette évaluation peut se faire connaissant le degré du point de branchement de la fonction, en multipliant les valeurs calculées pendant l'étape *1.* par chacune des racines q^{eme} de l'unité[BK86].
4. Fusion des différents brins de la tresse : on joint les brins de la tresse deux par deux, selon les valeurs affectées au extrémités des différents brins. On obtient alors un *noeud*, dont les caractéristiques (algébriques) reflètent le comportement de la fonction au voisinage du point de branchement.

Cette méthode peut etre utilisée, au prix d'un certain travail, dans les systèmes de calcul les plus classiques[1]. Néanmoins, il n'existe pas, à notre connaissance, de facilités pour, étant donné un ensemble de points correspondant aux valeurs d'une fonction *a priori* quelconque, les visualiser avec la méthode décrite précédemment dans un système de calcul formel ou numérique.

[1] c'est-à-dire que dans la plupart des systèmes de Calcul Formel notamment, il existe de nombreux moyens pour visualiser des noeuds connaissant leurs propriétés algébriques.

Exemple de visualisation.

Si nous voulons visualiser le comportement de la fonction

$$f(x) = x^{\frac{3}{2}} \tag{1.1}$$

autour de $x = 0$, nous pouvons appliquer la méthode décrite précédemment. Nous choisissons un cercle centré en 0, de rayon 1.

Les différentes étapes sont montrées sur la figure 1.1.

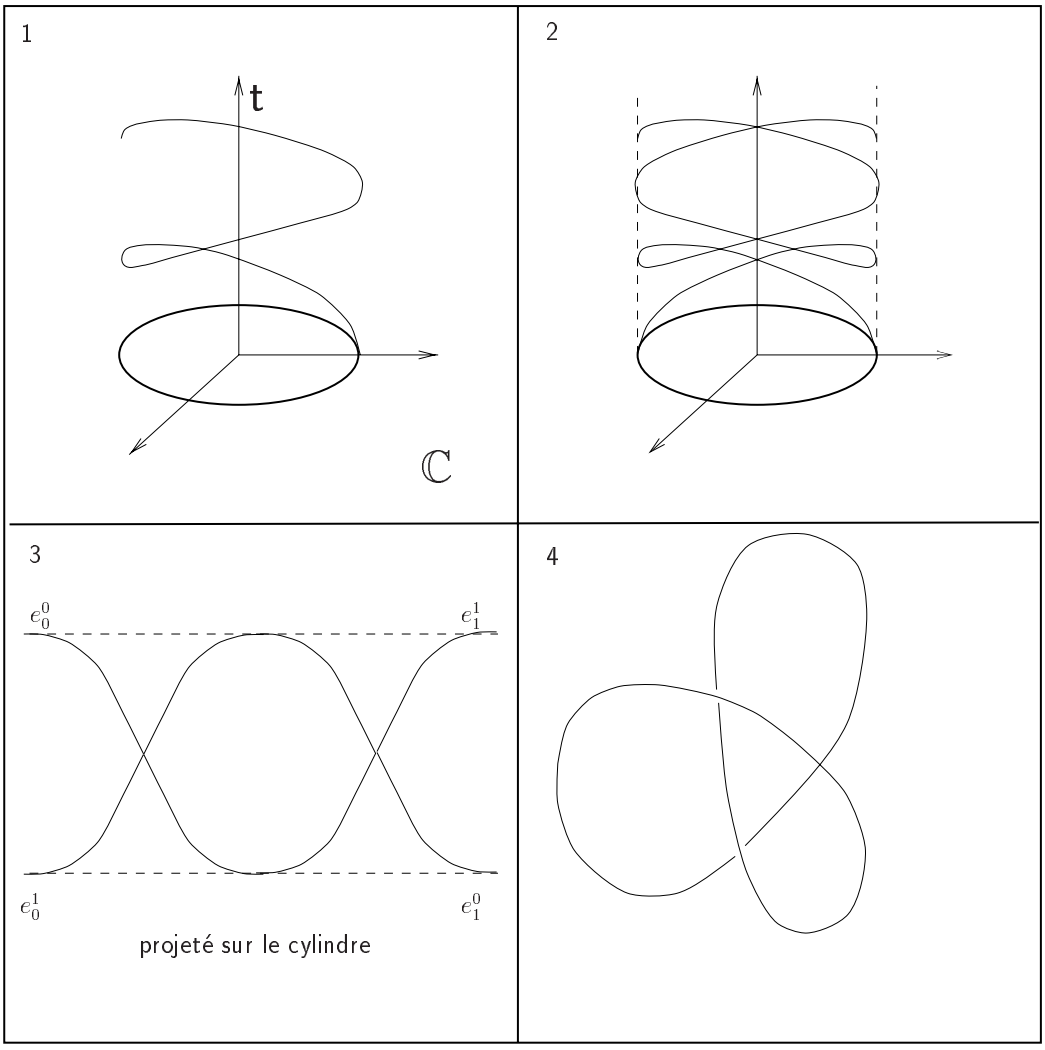


FIG. 1.1: Visualisation de fonctions admettant un point de branchement (simple) en 0

L'étiquette 1 présente une évaluation de la fonction f, celle qui pour $x = 1$ donne $f(x) = 1$. Les valeurs montrées sur la figure sont calculées avec cette détermination, et elles sont dessinées sur une courbe paramétrée par t, le paramètre de parcours du cercle. Nous obtenons alors une courbe tri-dimensionnelle.

Sur l'étiquette 2 de cette figure, nous avons dessiné une deuxième évaluation de la fonction f, celle-ci obtenue en considérant la seconde détermination de la puissance $\frac{3}{2}$. Pour cela, nous pouvons le faire comme expliqué précédemment, c'est-à-dire en multipliant les valeurs précédemment calculées par $\Leftrightarrow 1$, c'est-à-dire la deuxième racine carrée de l'unité. Nous obtenons de cette manière une autre courbe tri-dimensionnelle.

Sur l'étiquette 3, nous avons projeté les deux courbes obtenues précédemment sur le cylindre obtenu en "remontant" le cercle sur l'axe des paramètres. Le dessin ainsi obtenu est une *tresse*.

Enfin, sur l'étiquette 4, les extrémités de la tresse correspondant à une même valeur dans le plan complexe sont réunies : ainsi, sur la figure 1.1, les extrémités e_0^0 et e_1^1 correspondant à la même valeur dans le plan complexe (1) sont identifiées, de même que les extrémités e_0^1 et e_1^0. Nous obtenons ainsi une courbe simple, qui se trouve être un noeud dans l'espace.

L'intérêt de cette méthode de visualisation pour ce genre de fonction est multiple :

- Toutes les composantes de la figure sont facilement *calculables*.
- On n'a pas besoin de connaître *a priori* toutes les caractéristiques algébriques de la fonction étudiée : cette méthode fonctionne correctement avec un paquet de points.

Extension de la méthode.

Au lieu de considérer la fonction sur un cercle, on peut la considérer sur un *anneau* centré en x_0. Cet anneau peut être paramétrisé par deux variables, t et u, et donc le procédé précédent peut donner une représentation surfacique. Cette méthode n'a pas, à notre connaissance, été implémentée.

L'intérêt de ce mode de représentation en ce qui concerne les fonctions admettant un point de branchement est de fournir une visualisation du comportement *local* de la fonction au voisinage du point de branchement. Par contre, il est hors de question d'envisager une visualisation *globale* uniquement avec cette méthode. Il semblerait[BK86] que certaines méthodes de recollement existent afin de donner une vision *globale* du comportement d'une fonction sur un domaine de $\mathbb{C}$ contenant des points de branchement, et non plus seulement au voisinage de certains points.

1.2 Fonctions multiformes

1.2.1 Prolongement analytique

Comme le fait remarquer Dieudonné dans [Die68], les fonctions réelles C^∞ peuvent être modifiées dans un intervalle arbitrairement petit inclus dans leur intervalle de définition, sans que cela puisse affecter le comportement de la fonction sur d'autres intervalles. Ce n'est pas le cas pour les fonctions *analytiques*.

1.2.1.1 Theorème (Le principe du prolongement analytique [Die68, Die62]) Soient f, g deux fonctions analytiques dans un ensemble ouvert connexe $D \subset \mathbb{C}$. S'il existe un ensemble ouvert non vide (arbitrairement petit) $U \subset D$ tel que

$$f_{|U} = g_{|U}$$

alors on a $f = g$ sur D.

Nous pouvons introduire une *relation d'équivalence* basée sur le principe du prolongement analytique de la manière suivante :

1.2.1.2 Définition (Relation d'équivalence du prolongement analytique [For81]) Soit $x_0 \in \mathbb{C}$. Soit f une fonction analytique dans un ouvert connexe Ω_1, et g une fonction analytique dans un ouvert connexe Ω_2.

On dit que f et g sont équivalentes par la relation $\sim_{x_0}$ si il existe un ouvert $\Omega \subset \Omega_1 \cap \Omega_2$ tel que $x_0 \in \Omega$ et

$$f_{|\Omega} = g_{|\Omega}$$

1.2.1.3 Définition (Germe des fonctions analytiques en x_0 [For81]) Notons $\mathcal{F}_{x_0}$ l'ensemble des classes d'équivalence par la relation d'équivalence $\sim_{x_0}$. Pour un voisinage ouvert U de x_0, on définit l'application

$$\rho_{x_0} : \mathcal{F}(U) \mapsto \mathcal{F}_{x_0}$$

qui associe à $f \in \mathcal{F}(U)$, une fonction analytique dans U, sa classe d'équivalence par la relation d'équivalence $\sim_{x_0}$. $\rho_{x_0}(f)$ est appelé le *germe* de f en x_0. On notera $\mathcal{O}_{x_0}$ l'ensemble des germes de fonctions analytiques en x_0.

Le *principe du prolongement analytique* permet de prolonger *effectivement* certaines fonctions analytiques le long de chemins[Cha95]. Considérons pour cela γ, un chemin dans $\mathbb{C}$:

$$\gamma : [0, 1] \to \mathbb{C}$$

d'extrémités $x_0 = \gamma(0)$ et $x_f = \gamma(1)$.

1.2.1.4 Définition (Prolongement analytique le long d'un chemin [For81]) Le germe $\psi \in \mathcal{O}_{x_f}$ est un *prolongement analytique le long de* γ du germe $\phi \in \mathcal{O}_{x_0}$ si il existe une famille $\phi_t \in \mathcal{O}_{\gamma(t)}$ pour $t \in [0, 1]$ avec $\phi_0 = \phi$ et $\phi_1 = \psi$ telle que pour τ dans l'intervalle$[0, 1]$, il existe un voisinage de τ, $T \subset [0, 1]$ et un ouvert $U \subset \mathbb{C}$ avec $\gamma(T) \subset U$ et une fonction analytique f dans U tels que

$$\forall t \in T, \rho_{\gamma(t)}(f) = \phi_t$$

De par la compacité de $[0, 1]$, cette définition permet de calculer *effectivement* des prolongements analytiques d'une fonction analytique dont on connaît le développement en série de Taylor au voisinage d'un point de $\mathbb{C}$, comme expliqué sur la figure 1.2. Le disque Δ_0 est le plus grand disque centré en x_0 dans lequel le développement en

série de Taylor est convergent. Si ce disque a un rayon fini, alors il existe un point sur le bord de ce disque où la série est divergente.

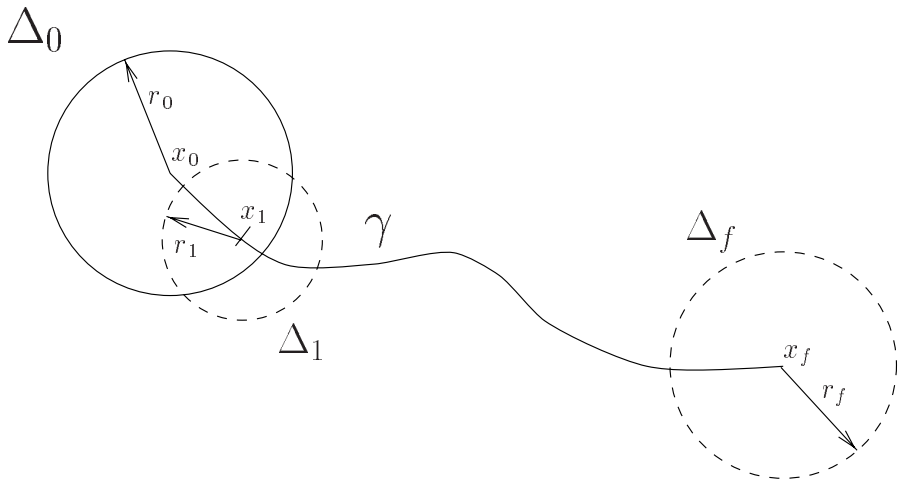


FIG. 1.2: Prolongement analytique le long d'un chemin

Si nous nous donnons une fonction analytique au voisinage de x_0, dont le développement en série de Taylor est convergent dans Δ_0, un disque de rayon r_0,

$$\sum_{i=0}^{\infty} a_i (x \Leftrightarrow x_0)^i$$

nous pouvons alors choisir un point $x_1 \in \gamma \cap \Delta_0$. Puisque $x_1 \in \Delta_0$, la valeur de la fonction f en x_1 peut être obtenue en sommant la série. En dérivant k fois la série, la valeur de la dérivée i^{eme} de la fonction peut être aussi obtenue en x_1. Ainsi, un développement en série de f peut être obtenu dans un disque Δ_1, centré en x_1 et de rayon r_1 par la formule suivante, où les b_i représentent les valeurs des dérivées i^{eme} de la fonction calculées précédemment :

$$\sum_{i=0}^{\infty} \frac{b_i}{i!} (x \Leftrightarrow x_1)^i$$

Nous savons de plus que $r_1 \geq d(x_1, \Delta_0)$. En effet, si tel n'était pas le cas, il existerait une singularité à l'intérieur de Δ_0, et ainsi la série de Taylor ne serait pas convergente à l'intérieur de Δ_0. Ainsi, si cette inégalité est *stricte*, nous avons prolongé la fonction f à l'extérieur de Δ_0.

Il se peut que le point x_1 choisi ne permette pas de prolonger la fonction f au delà de Δ_0. Dans ce cas, il suffit de choisir un autre point x_1, plus proche de la frontière de Δ_0. Si, pour toutes les valeurs x_1 choisies sur γ, aucune ne convient pour le prolongement de la fonction f, alors le processus échoue. Notons que dans ce cas, le chemin γ rencontre une singularité de la fonction f.

Dans l'hypothèse (abstraite) où la fonction est analytique dans un voisinage ouvert de γ, ce prolongement est toujours possible. Dans ce cas, après un nombre *fini*

(puisque $[0,1]$ est compact) de prolongements, la fonction peut être prolongée jusqu'à x_f, c'est-à-dire que nous obtenons une série convergente dans un disque Δ_f, de rayon r_f non nul.

Ce procédé effectif peut être utilisé dans de nombreuses applications, comme nous le verrons dans le chapitre 2.

1.2.2 Exemple

Considérons, à titre d'exemple, la "fonction racine carrée de z" définie par la solution f telle que $f(1) = 1$ de l'équation algébrique

$$(f(z))^2 \Leftrightarrow z = 0$$

Cette fonction a pour développement en série de Taylor au voisinage de 1 la série

$$T_0(z) = 1 + \frac{1}{2}(z \Leftrightarrow 1) \Leftrightarrow \frac{1}{8}(z \Leftrightarrow 1)^2 + \cdots$$

Notons C le cercle centré en 0, de rayon 1, et pour $t \in [0,1]$, notons

$$\gamma(t) = e^{2i\pi t}$$

une paramétrisation de ce cercle.

Nous nous intéresserons aux couples de la forme $(\gamma(t), \rho_{\gamma(t)}(T_t))$, où $t \in [0,1]$ et $\rho_{\gamma(t)}(T_t)$ est obtenu par prolongement analytique à partir de $\rho_1(T_0)$.

La série T_0 est convergente dans un disque centré en 1 de rayon 1, que nous pouvons noter Δ_0, comme sur la figure 1.3.

Nous pouvons appliquer le principe du prolongement analytique sur le cercle C, et donc choisir $t_1 \in [0,1]$ de telle manière que $\gamma(t_1) \in \Delta_0$. La série $T_{t_1}(z)$ représentant le développement en série de f au voisinage de $\gamma(t_1)$ est telle que

$$T_{t_1}(z) T_{t_1}(z) = z$$

Les coefficients α_k, $k \geq 1$, de cette série sont donc les solutions du système (triangulaire)

$$\sum_{i=0}^{k} \binom{k}{i} \alpha_i \alpha_{k-i} = 0$$

où $\alpha_0 = e^{i\pi t_1}$. Ainsi, le développement en série de f au voisinage de z_1 s'écrit

$$T_{t_1}(z) = e^{i\pi t_1} + \frac{e^{-i\pi t_1}}{2}(z \Leftrightarrow \gamma(t_1)) \Leftrightarrow \frac{e^{-3i\pi t_1}}{8}(z \Leftrightarrow \gamma(t_1))^2 + \cdots$$

où T_{t_1} est convergente dans un disque Δ_{t_1}, de rayon 1, de centre $\gamma(t_1)$.

Poursuivant le procédé, jusqu'à ce que $t = 1$, c'est-à-dire $\gamma(t) = 1$, nous obtenons alors la série

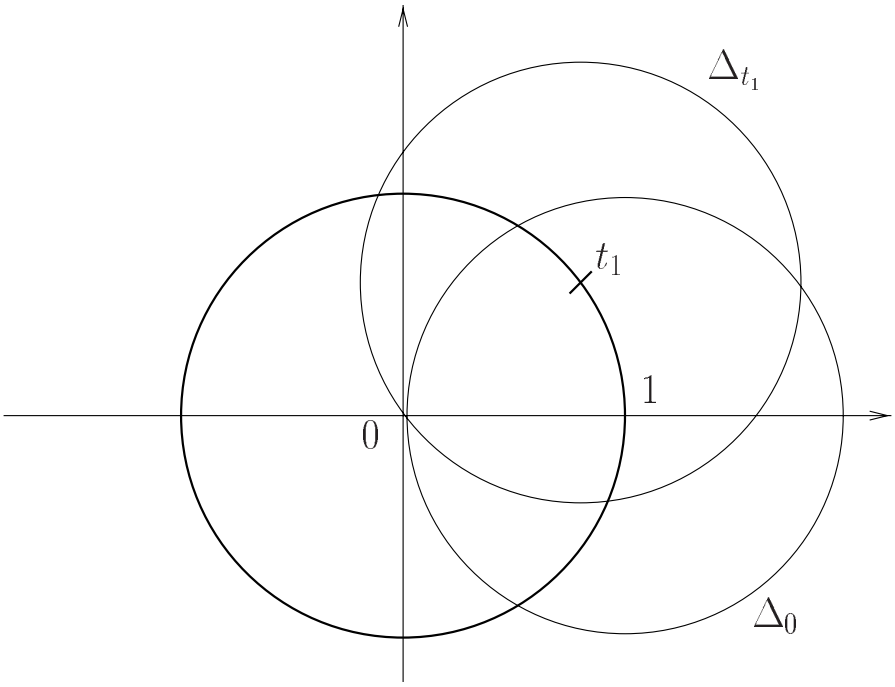


FIG. 1.3: Prolongement analytique du germe de la "fonction racine carrée"

$$T_1(z) = \Leftrightarrow 1 \Leftrightarrow \frac{1}{2}(z \Leftrightarrow 1) + \frac{1}{8}(z \Leftrightarrow 1)^2 + \cdots$$

à savoir $T_1(z) = \Leftrightarrow T_0(z)$, alors que puisque ces deux séries représentent le développement en série de Taylor de la "fonction racine carrée", elles devraient être égales. On a donc

$$(1, \rho_1(T_0)) \neq (1, \rho_1(T_1))$$

On peut introduire les notions suivantes pour caractériser de manière plus pratique les conséquences de ce phénomène.

1.2.2.1 Définition (Indétermination et fonctions multiformes) Lorsqu'une opération arithmétique ou l'évaluation numérique d'une fonction peut présenter plusieurs résultats qualitativement et quantitativement différents, nous dirons que le calcul présente une *indétermination*. Une *fonction multiforme* est une fonction dont l'évaluation en un point du plan complexe présente des indéterminations.

Ainsi, l'évaluation de la "fonction racine carrée" en un point quelconque du plan complexe (sauf 0) présente des indéterminations, et cette fonction est donc multiforme.

Sans pour autant remettre en question la définition de fonction, Riemann a introduit l'idée que le plan complexe n'est pas le domaine de définition idéal pour ce genre de fonctions, et cette remarque a mené plus tard à la notion de *Surface de Riemann*

pour définir correctement les fonctions multiformes, et donner un cadre théorique au phénomène mis en évidence précédemment.

1.2.3 Surfaces de Riemann

Pour une fonction multiforme f donnée, au lieu de considérer que f est définie sur un sous-ensemble de $\mathbb{C}$, nous pouvons considérer l'ensemble

$$\mathcal{M}_f = \{(z, \rho_z(f)), z \in \mathbb{C}\}$$

où $\rho_z(f)$ désigne un germe de f en z.

Pour revenir à l'exemple précédent, l'ensemble $\mathcal{M}_{\surd}$ est constitué des couples $(z, \rho_z(f))$, obtenus comme précédemment par prolongement analytique. On a donc $(1, \rho_1(T_0)) \in \mathcal{M}_{\surd}$ et $(1, \rho_1(T_1)) \in \mathcal{M}_{\surd}$, avec $(1, \rho_1(T_0)) \neq (1, \rho_1(T_1))$.

Les valeurs de la fonction f peuvent être retrouvées [CNP93] en considérant l'application, définie sur $\mathcal{M}$, qui à un couple $(z, \rho_z(f))$, associe la valeur $P_z(0)$, où P_z est un représentant de $\rho_z(f)$. Mais cette fois, à un point de $\mathcal{M}_f$ correspond *une unique* valeur.

Pour être complet, nous pouvons exhiber une topologie simple pour l'ensemble $\mathcal{M}_f$ de la manière suivante :

1.2.3.1 Définition (Topologie de $\mathcal{M}_f$ [CNP93]) Soit $(a, \rho_a(f)) \in \mathcal{M}_f$, et r_a le rayon de convergence d'un représentant de $\rho_a(f)$. On peut définir un *voisinage* de $(a, \rho_a(f))$ dans $\mathcal{M}_f$ par

$$\mathcal{V}((a, \rho_a(f)), r) = \{(b, \rho_b(f)), |b \Leftrightarrow a| < r\}$$

où $r < r_a$.

Ces voisinages représentent des *disques* de centre $(a, \rho_a(f))$, de rayon r.

L'ensemble $\mathcal{M}_f$ muni de cette topologie possède une structure de *surface de Riemann* :

1.2.3.2 Définition (Surface de Riemann [For81]) Soit M un espace topologique de Hausdorff connexe tel que :

- $(U_i)_{i \in I}$ est un recouvrement de M par des ouverts,
- $(\Phi_i)_{i \in I} : U_i \rightarrow V_i$, où V_i est un ouvert de $\mathbb{C}$ et Φ_i est un homéomorphisme (appelé *coordonnée locale*),
- si $U_i \cap U_j \neq \emptyset$, alors la fonction

$$f_{i,j}(z) = \Phi_j(\Phi_i^{-1}(z))$$

est analytique pour $z \in \Phi_i(U_i \cap U_j)$. Cette fonction est appelée *fonction de raccordement analytique*.

M a une structure de *variété complexe analytique de dimension 1*. On dit alors que M est une *Surface de Riemann*.

Cette définition abstraite fournit un jeu de coordonnées locales, et donc permet de paramétrer localement la surface afin de pouvoir projeter ses éléments sur le plan complexe.

1.2.3.3 Définition (Surface de Riemann associée à une fonction) La *surface de Riemann associée à la fonction* f est l'ensemble $\mathcal{M}_f$, muni de la topologie précédente.

On choisit les U_i comme des disques centrés en des éléments $(z_i, \rho_{z_i}(f))$. Les fonctions Φ_i sont définies par

$$\forall (z, \rho_z(f)) \in U_i, \Phi_i((z, \rho_z(f))) = P_{z_i}(z \Leftrightarrow z_i)$$

où P_{z_i} est un représentant de $\rho_{z_i}(f)$.

De nombreuses propriétés découlent de cette définition de surface de Riemann associée à une fonction. Dans le cas particulier des fonctions *algébriques*, la surface de Riemann associée est de *genre fini*[FK80]. Les caractéristiques topologiques des surfaces de Riemann (appelées surfaces de Riemann compactes) peuvent être calculées à partir de l'équation algébrique définissant la fonction algébrique, ce qui en fait un objet particulièrement bien manipulable. Dans ce cas, nous pouvons construire effectivement la surface de Riemann sur laquelle la fonction multiforme a les propriétés d'une fonction usuelle : cette surface de Riemann est un recouvrement de $\mathbb{C}$ à un nombre fini de feuillets, où tout élément de $\mathbb{C}$ qui n'est pas un point de branchement pour la fonction f a le même nombre d'antécédents par les projections associées à la surface de Riemann[CNP93].

Dans le cas particulier où la fonction f possède un comportement de type logarithmique au voisinage d'un point, la surface de Riemann associée à f n'est plus compacte. On peut néanmoins en donner ses caractéristiques topologiques dans certains cas[Bea84].

1.2.4 Intérêt pratique des Surfaces de Riemann

L'intérêt des surfaces de Riemann, du moins en ce qui concerne les visualisations de fonctions de la variable complexe et les calculs que nous pouvons être amenés à y faire, est de fournir une "extension" du plan complexe où les fonctions multiformes sont vraiment des fonctions.

Applications aux problèmes de calculs avec des fonctions complexes.

Les surfaces de Riemann sont le cadre naturel de définition des fonctions multiformes. En effet, nous avons construit les surfaces de Riemann associée à un germe de telle manière que les fonctions qui en découlent n'admettent qu'une seule valeur comme image d'un point d'une surface de Riemann. De plus, ces points des surfaces de Riemann ne sont pas très éloignés de leur homologue complexe, car il suffit d'une projection pour retrouver la valeur complexe y correspondant.

D'un point de vue pratique, il s'agit de manipuler les nombres complexes implicitement comme des éléments de surface de Riemann. La logique du procédé peut

sembler douteuse, car il implique la connaissance *a priori* de la structure, topologique notamment, de la surface elle-même, ce qui n'est pas forcément le cas lorsque nous récupérons une suite de nombres complexes issus d'un calcul numérique.

Dans le cas particulier des fonctions racines, nous pouvons considérer que la connaissance des degrés d'une telle fonction peut nous aider à anticiper le comportement de la fonction au voisinage de ses points de branchements. Mais cette connaissance n'est généralement pas acquise lors d'un procédé numérique. La solution du problème passe ici par un codage des diverses transformations que le calcul peut faire subir aux nombres complexes. En effet, les opérations arithmétiques classiques telles que les multiplications, additions, etc ... peuvent avoir des effets différents selon la structure topologique de la surface de Riemann sur laquelle ils sont considérés. La technique ici consiste *à ne pas "tout à fait" évaluer* les résultats des opérations arithmétiques que les nombres complexes subissent.

Prenons un exemple : considérons que nous voulons évaluer la fonction

$$f(z) = z^{\alpha}$$

en différents points du plan complexe, pour $\alpha \in \mathbb{C}$. Nous avons vu dans ce qui précède que, selon la valeur du paramètre α, cette évaluation peut donner des résultats très différents. Comme généralement, notamment lorsque cette fonction est la solution d'une équation différentielle dans le plan complexe, la valeur de α n'est pas connue, voire même pas calculable (en particulier si α est irrationel), l'évaluation d'une telle fonction peut donner des résultats particulièrement aléatoires.

La technique consiste alors à *coder* les différentes opérations arithmétiques qui sont appliquées à z pour l'évaluation de la fonction ci-dessus, notamment en considérant des quantités comme le *numéro de feuillet*. Une fois que le calcul est fini, la valeur de cette quantité peut être interprétée, et des résultats généraux peuvent être tirés du calcul. Nous verrons des exemples concrets de ce procédé dans la partie II. Dans cette partie, nous commencerons par présenter une étude "en profondeur" des problèmes que peuvent poser les indéterminations (aussi bien dans des calculs arithmétiques simples que dans des problèmes de résolution d'équations différentielles), puis nous proposerons une solution logicielle qui, à défaut d'être universelle, est simple à mettre en oeuvre (par la mise à disposition d'opérateurs arithmétiques modifiés) et efficace sur certains exemples. Nous donnerons de plus dans cette partie des méthodes logicielles de traitement des indéterminations.

Applications aux problèmes de visualisation des fonctions complexes.

Les fonctions multiformes sont délicates à visualiser, car non seulement les phénomènes d'indéterminations nuisent à la consistance de leur évaluation, mais de plus elles nécessitent des procédés *adaptés* de visualisation afin d'en tirer des renseignements utiles. Nous étudierons la visualisation des fonctions multiformes dans deux cadres différents : tout d'abord dans la partie II, avec les problèmes de déterminations continues et d'évaluations de fonctions sur des chemins, puis de manière plus intrinsèque aux équations différentielles dans la partie IV.

Dans la partie II, nous nous concentrerons sur les problèmes de visualisation de fonctions multiformes, en particulier sur les différents formalismes de visualisation des éléments de surface de Riemmann, selon la topologie de ces surfaces. Nous verrons en particulier plusieurs méthodes de représentation des éléments de ces ensembles, et comment interpréter graphiquement les numéros de feuillet.

Dans la partie IV, nous verrons, sur la base de solutions d'équations différentielles calculées par des méthodes numériques, comment interpréter les comportements multiformes de ces fonctions. Nous verrons que ces comportements peuvent quand même, par le biais d'une étude visuelle des solutions, mener à des caractérisations graphiques de certains phénomènes, et donc donner des renseignements sur les solutions des équations différentielles considérées.

Chapitre 2

Équations différentielles dans le plan complexe

2.1 Les équations différentielles

2.1.1 Le théorème de Cauchy-Lipschitz complexe

2.1.1.1 Theorème (Version complexe du théorème de Cauchy–Lipschitz [Was76]) Soit $f(x, y)$ une fonction de $\mathbb{C} \times \mathbb{C}^N$ dans $\mathbb{C}^N$, holomorphe pour toutes ses variables dans un domaine Ω de $\mathbb{C}^{N+1}$. Si nous supposons que (x_0, y_0) appartient à Ω, alors l'équation différentielle

$$\frac{dy}{dx} = f(x, y) \tag{1.1}$$

admet exactement une solution $y = y(x)$ telle que $y(x_0) = y_0$. De plus, cette solution est holomorphe en x_0.

Ce théorème est *local* : on est assuré de l'existence d'une solution à l'équation (1.1) holomorphe en x_0. Par contre, dans le cas général des équations différentielles non-linéaires, rien n'assure que cette solution va pouvoir être étendue à l'ensemble du domaine Ω (voir la notion de singularités mobiles [BO78], p. 149), même par prolongement analytique. De plus, dans certains cas, cette solution peut être multiforme, ou présenter des points singuliers dont la présence n'aurait pu être devinée au vu de l'équation (1.1) ([Fed87], p. 10).

La restriction d'holomorphie de la fonction f est en fait assez naturelle (voir [BO78], p.29). En effet, le fait d'écrire une équation différentielle avec la fonction y implique naturellement que y est dérivable dans $\mathbb{C}$, et donc holomorphe. Si tel n'était pas le cas l'équation (1.1) ne serait plus une équation différentielle ordinaire. Cela sort donc de notre propos.

2.1.2 Résolution des équations différentielles

De nombreux moyens *effectifs* existent à ce jour pour résoudre de telles équations. On distingue généralement deux grandes classes de méthodes : les méthodes numériques des méthodes formelles.

Méthodes numériques.

Les méthodes numériques classiques, importées du domaine réel, sont généralement transposables facilement aux équations différentielles complexes. Étant donnée une condition de Cauchy en un point, (x_0, y_0), le problème consiste à trouver une solution de l'équation différentielle telle que $y(x_0) = y_0$. Les méthodes numériques classiques [HNW87, Ise96] permettent de prolonger de manière numérique les valeurs calculées localement.

Dans le plan complexe, ces méthodes s'appliquent sur des chemins, et le chemin est parcouru avec un *pas*. Généralement, le pas est calculé à partir d'une discrétisation de l'intervalle de paramétrisation du chemin, et les valeurs de la discrétisation sont reportées sur le chemin, ce qui donne un ensemble de points du plan complexe, $(x_n)_{n<N}$. En chaque point ainsi calculé, une expression de la forme

$$y_n = \Phi(x_n, y_{n-1}, \cdots)$$

est évaluée, et cette valeur représente une approximation de la valeur $y(x_n)$. Ainsi, la valeur y_0 est propagée le long du chemin.

La difficulté dans ce genre de méthodes est de fournir tout au long du calcul des critères de qualité des valeurs calculées : *l'estimation de l'erreur globale* [AL97] permet d'y arriver, l'erreur globale étant la différence entre la "vraie" valeur de la solution au point x_n et la valeur numérique calculée.

L'intérêt de ce genre de méthodes est de fournir des valeurs numériques facilement et rapidement, ce qui n'est pas forcément le cas des autres types de méthodes que nous verrons par la suite.

Méthodes formelles.

Nous allons exposer le principe de deux types de méthodes formelles : la résolution d'équation différentielle avec des séries formelles et la résolution par méthodes spectrales.

Résolution sous forme de séries formelles La démarche consiste dans ce cas à rechercher des solutions d'une équation ou d'un système différentiel appartenant à une certaine classe de solutions, contenant des séries formelles, mais aussi des parties exponentielles, logarithmiques, etc ... Les coefficients de ces séries sont calculés en injectant l'expression dans l'équation différentielle, et en déduisant des relations entre les coefficients des séries.

Cette méthode fonctionne bien dans le cas des équations différentielles linéaires à coefficients polynomiaux, car on dispose dans ce cas d'une forme générale des solutions, contenant des séries de Puiseux, des exponentielles et des logarithmes. De plus, l'expression de la partie exponentielle peut être connue à l'avance, en composant un polygone de Newton à partir des différents coefficients de l'équation différentielle. Connaissant la partie exponentielle, les coefficients de la série peuvent généralement être calculés, grâce à une relation de récurence déduite de l'équation différentielle.

Les évaluations numériques de ces séries peuvent par contre poser des problèmes : en effet, il peut arriver, même dans le cas linéaire, que les séries obtenues soient divergentes. Il faut ainsi recourir à des méthodes de resommation pour avoir des valeurs numériques d'une solution sous forme de série. Cependant, de telles méthodes mixtes (c'est-à-dire formelles/numériques) donnent généralement de très bons résultats, et constituent dans certains cas la seule manière d'avoir une solution numérique satisfaisante [Sha94].

Méthodes spectrales L'utilisation des méthodes spectrales pour la résolution d'équations différentielles (ou d'équations aux dérivées partielles) est motivée par les bonnes propriétés d'approximation des décompositions spectrales (par exemple, les séries de Fourier, Chebyshev ou de Legendre). Le principe consiste, pour résoudre une équation du type

$$Lu(x) = f(x)$$

à en approcher la solution par une série spectrale tronquée à un ordre fixé,

$$u_N = \sum_{i=0}^{N} a_i P_i$$

et à considérer un *résidu* $R(a_0, a_1, \cdots, a_N; x) = Lu_N \Leftrightarrow f$. Les contraintes imposées à R définissent les différentes méthodes[CHQZ88] : orthogonalité avec certains sous-espaces pour les méthodes de Galerkin et les τ–méthodes, annulation en un nombre fini de points pour les méthodes de collocation. De plus, les manières de considérer les conditions initiales pour ces problèmes varient selon les méthodes.

L'utilisation du calcul formel a permis l'augmentation des champs d'application de ces méthodes jusqu'ici limitées à des problèmes simples et de petites tailles[Boy93], et ce grâce au calcul automatique des résidus (tâche impossible à la main), et à l'intoduction des paramètres symboliques[Ged77, Reb96].

Les méthodes formelles présentées donnent des résultats qui doivent être évalués numériquement pour être pleinement exploités. Par exemple, la visualisation des solutions d'une équation différentielle doit généralement passer par l'évaluation numérique des résultats renvoyés par une méthode formelle. Dans la suite de ce chapitre, nous mettrons de côté l'aspect résolution des équations différentielles dans le plan complexe, car ce qui nous intéresse dans ce chapitre est *la visualisation des solutions calculées, d'une manière ou d'une autre.*

2.2 Visualisation de solutions d'équations différentielles complexes

2.2.1 Difficultés du problème

Les solutions d'équations différentielles linéaires sont avant tout des fonctions de la variable complexe x (la *variable d'intégration*). Mais en considérant ces solutions uniquement commes des fonctions de la variable complexe, de nombreuses informations peuvent être perdues.

En ce qui concerne les équations et systèmes différentiels *linéaires*, de nombreux efforts ont été faits sur des visualisations rendant compte des phénomènes locaux au voisinage des singularités des équations linéaires[Ric88]. De tels procédés permettent de mettre en évidence des phénomènes *locaux*, comme par exemple la comparaison de plusieurs techniques d'intégration sur une même équation [RJ91]. Mais ils ne sont pas à l'étude de la dépendance de la solution aux conditions initiales dans le cas des équations différentielles non linéaires, car ils ne permetttent de visualiser qu'un petit nombre de solutions à la fois, nombre au-delà duquel la complexité de la visualisation est trop élevée.

Il existe de plus d'autres problèmes liés à la visualisation des solutions d'équations différentielles de la variable complexe : dans certains cas, pour être significative, la vision du problème doit être *globale*, alors que l'intégration numérique ou les théorèmes d'existence de solutions donnent plutôt des résultats *locaux*. Par exemple, le caractère local de ces théorèmes ne permet pas *a priori* de prendre en compte le caractère multiforme des solutions considérées.

Nous allons par la suite donner des méthodes de visualisation efficaces (au sens de la pertinence des données représentées) de solutions d'équations différentielles, non-linéaires notamment. Nous verrons que cela peut passer par une re-paramétrisation des valeurs d'une solution d'une équation différentielle, ou bien par des modes de visualisation rendant compte des phénomènes globaux qui se produisent avec les solutions d'équations différentielles. Pour cela, nous verrons un formalisme de représentation basé sur le principe des *portraits de phase* de systèmes réels de dimension deux. Pour commencer, nous définirons les portraits de phase des systèmes dynamiques, puis nous verrons comment cela peut s'appliquer aux problèmes des équations différentielles dans le plan complexe.

2.2.2 Les portraits de phase des systèmes réels de dimension 2

Si nous considérons un système dynamique à deux dimensions de la forme suivante

$$\begin{cases} \dfrac{dx}{dt} &= f_1(t,x,y) \\ \dfrac{dy}{dt} &= f_2(t,x,y) \end{cases} \tag{2.2}$$

qui vérifie les conditions du théorème de Cauchy-Lipschitz réel sur $I \times \Omega$, où $\Omega \subset \mathbb{R}^2$ est un ouvert de $\mathbb{R}^2$ et I un intervalle de $\mathbb{R}$, on a donc existence et unicité d'une solution possédant comme condition initiale $(t_0,(x_0,y_0))$ pour tout $t_0 \in I$ et $(x_0,y_0) \in \Omega$.

2.2.2.1 Définition (Flot du système dynamique [Dem89]) On définit le flot d'un système dynamique de la forme (2.2) par l'application

$$\begin{array}{rccc} \psi_t & : & \Omega & \to & \mathbb{R}^2 \\ & & \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} & \mapsto & \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} \end{array}$$

où $\begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$ est la solution du système (2.6) telle que $\begin{pmatrix} x(t_0) \\ y(t_0) \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$.

Cette solution vérifie la condition initiale $(t_0,(x_0,y_0))$.

Une *orbite* [BO78] (associée à une condition initiale $(t_0,(x_0,y_0))$) est alors l'ensemble défini par

$$o(x_0,y_0) = \{\psi_t(x_0,y_0), t \in I\}.$$

Une orbite $o(x_0,y_0)$ peut être vue comme une courbe paramétrée (par t) dans Ω. À partir des orbites du système, les *portraits de phase* et les *portraits de phase étendus* peuvent être définis.

2.2.2.2 Définition (Portrait de phase d'un système dynamique) On appelle *portrait de phase* associé au système (2.2) la partition de Ω formée par les différentes orbites du système.

D'une manière pratique, pour tracer le portrait de phase d'un système, nous considérons un nombre fini de conditions initiales dans $I \times \Omega$ (sur une grille régulière, par exemple), et nous traçons les orbites qui en sont issues, en tant que courbes orientées dans le sens croissant pour la variable t. La figure 2.1 montre un exemple ([BO78], p. 184) de portrait de phase associé au système

$$\begin{cases} \dfrac{dx}{dt} &= x + y \Leftrightarrow x(x^2+y^2) \\ \dfrac{dy}{dt} &= \Leftrightarrow x + y \Leftrightarrow y(x^2+y^2) \end{cases}$$

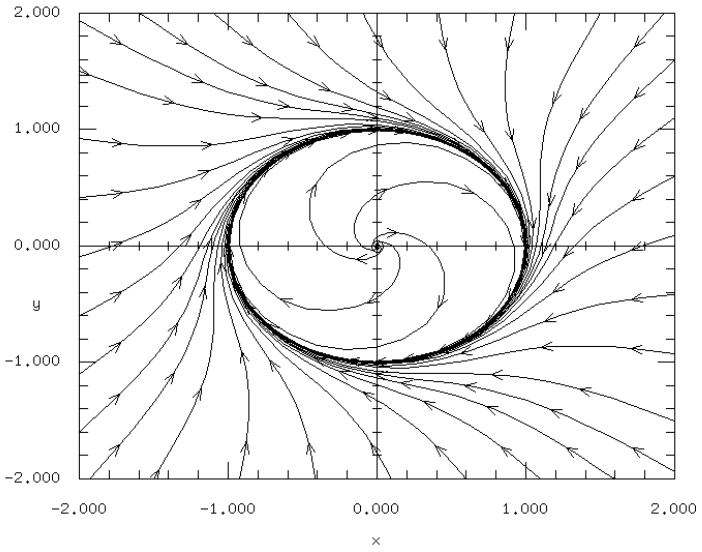


FIG. 2.1: Un exemple de portrait de phase

Sur ce portrait de phase, les points (x_0, y_0) sont pris régulièrement dans le carré $[\Leftrightarrow 2, 2] \times [\Leftrightarrow 2, 2]$ et t_0 est fixé à 0. Ce tracé peut être intéressant en ce qui concerne l'interprétation du comportement des solutions du système : en effet, la présence d'un cycle limite (le cercle de centre 0 de rayon 1) et d'un point spirale en 0 peut être détectée, et de plus l'évolution dans le temps d'une solution du système, à partir de sa condition initiale, peut être suivie sur le tracé.

Nous pouvons relever les orbites dans $\Omega \times I$ en considérant les courbes $(x(t), y(t), t)$ dont les deux premières composantes sont les solutions du système (2.2), et chaque point $(x(t), y(t))$ est placé dans l'espace à l'altitude t lui correspondant.

2.2.2.3 Définition (Orbites relevées et Portraits de Phase étendus) On appelle *orbite relevée* d'un système de type (2.2) une courbe de la forme

$$\mathcal{O}(x_0, y_0) = \{(\psi_t(x_0, y_0), t), t \in I\}$$

La partition de $\Omega \times I$ par ces courbes forme le *portrait de phase étendu* du système (l'espace txy dans [HW95]).

La figure 2.2 montre plusieurs orbites relevées, constituant un portrait de phase étendu :

La courbe tracée sur cette figure représente la solution $(x(t), y(t))$ telle que $(x(\tau_1), y(\tau_1)) = (x_0, y_0)$. Un portrait de phase étendu peut être vu comme le graphe de la fonction ψ_t dans $\Omega \times I$.

2.2.2.4 Définition (Systèmes autonomes) On appelle un *système autonome* un système dont le membre de droite est indépendant de t, c'est-à-dire lorsque le système peut s'écrire :

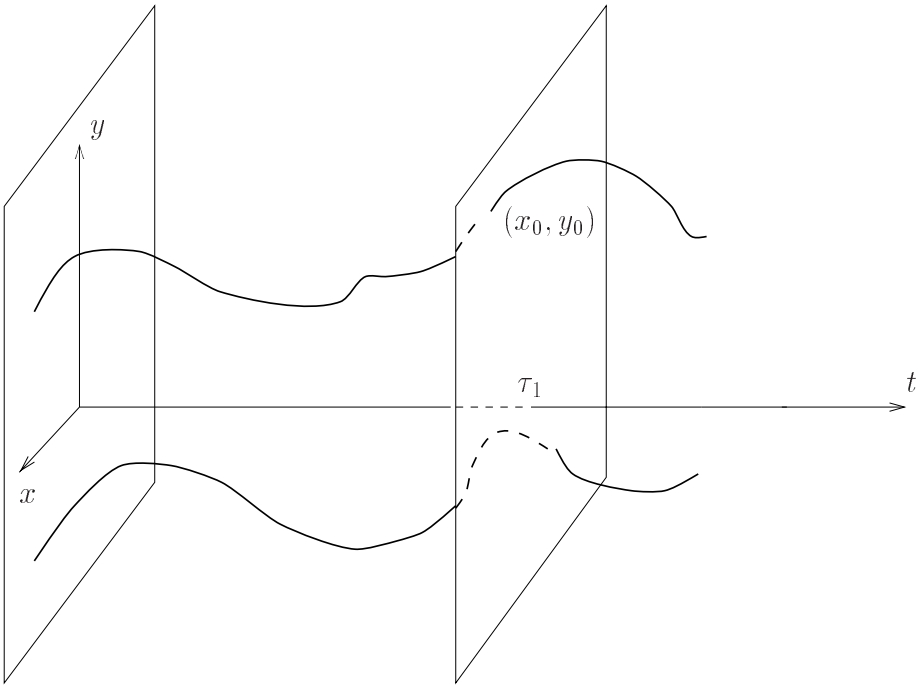


FIG. 2.2: Un portrait de phase étendu

$$\begin{cases} \dfrac{dx}{dt} &= g_1(x,y) \\ \dfrac{dy}{dt} &= g_2(x,y) \end{cases} \tag{2.3}$$

Ces systèmes présentent un intérêt lorsqu'ils sont visualisés sous forme de portraits de phase. En effet, si $g_1(x,y)$ et $g_2(x,y)$ vérifient les conditions du théorème de Cauchy-Lipschitz sur un ouvert $\Omega \subset \mathbb{R}^2$, alors deux orbites issues de deux conditions initiales différentes ont une intersection vide ([BO78], p. 173). Cela implique en particulier que deux courbes représentant deux orbites différentes ne peuvent pas se couper sur un portrait de phase.

Un système *non autonome* est un système dont le membre de droite dépend de t. Contrairement aux systèmes autonomes, pour un système non-autonome, rien n'empêche deux orbites issues de deux conditions initiales différentes d'avoir une intersection non vide, ce qui peut nuire à la lisibilité des portraits de phase de ces systèmes. Par contre, si f_1 et f_2 vérifient encore les conditions du théorème de Cauchy-Lipschitz sur Ω, alors deux orbites provenant de deux conditions initiales différentes ne peuvent pas se couper sur un portrait de phase étendu, ce qui peut améliorer la compréhension du comportement des solutions du système.

Un intérêt des portraits de phase est qu'ils peuvent permettre de donner des interprétations dynamiques des solutions qui y sont représentées : on peut suivre les positions d'un point dont le mouvement est déterminé par le système différentiel à

partir de sa position initiale (x_0, y_0), pour $t = t_0$.

D'une manière générale, on associe à un portrait de phase les renseignements suivants :

- un système dynamique
- un intervalle de variation de la variable t, $I = [t_1, t_2] \subset \mathbb{R}$
- un ensemble de points $(x_0, y_0) \in \mathbb{R}^2$
- pour t_0 fixé dans $[t_1, t_2]$, l'ensemble des orbites des solutions du système différentiel ayant comme conditon initiale $(t_0, (x_0, y_0))$.

Ce principe peut être naturellement étendu aux équations différentielles de la variable complexe.

2.2.3 Application aux équations différentielles de la variable complexe

Considérons une équation différentielle dans $\mathbb{C}$, de la forme suivante

$$\frac{dy}{dx} = f(x, y), \tag{2.4}$$

De plus, supposons que cette équation vérifie les conditions du théorème de Cauchy en version complexe (théorème 2.1.1.1) sur Ω ouvert de $\mathbb{C}^2$.

Considérons un chemin C de $\mathbb{C}$ et un ensemble $\gamma \subset \mathbb{C}$, tels que $C \times \gamma \subset \Omega$, ainsi qu'un point $x_0 \in C$. Nous pouvons alors nous poser la question suivante :

Si nous intégrons l'équation (2.4) sur le chemin d'intégration C à partir de x_0, quelle est la solution qui possède $y_0 \in \gamma$ comme valeur en x_0 ?

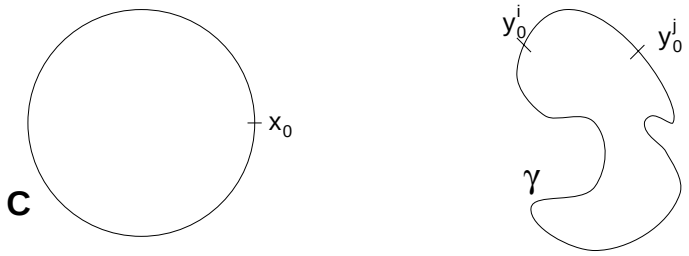


FIG. 2.3: Les chemins considérés

Supposons de plus que le chemin d'intégration C est un *chemin paramétré*, défini par une fonction Φ du paramètre t :

$$\begin{aligned} \Phi : [t_1, t_2] &\rightarrow \mathbb{C} \\ t &\mapsto \Phi(t) \end{aligned}$$

Si nous considérons de plus que le chemin C est *régulier*, c'est-à-dire

$$\forall t \in [t_1, t_2], \frac{d\Phi}{dt} \neq 0,$$

alors nous pouvons écrire, après avoir posé $x = \Phi(t)$,

$$dx = \frac{d\Phi}{dt}(t)dt$$

d'où

$$\frac{dy}{dx} = \frac{dy}{dt}\frac{dt}{dx}$$

et donc

$$\frac{dy}{dx} = \frac{dy}{dt}\frac{1}{\frac{d\Phi}{dt}(t)}$$

L'équation (2.4) peut donc s'exprimer de la manière suivante :

$$\frac{dy}{dt} = \frac{d\Phi}{dt}(t) f(\Phi(t), y) \tag{2.5}$$

où y est considérée comme une fonction à valeur complexe de la variable réelle t.

Si maintenant nous posons $y = y_1 + iy_2$, où y_1 et y_2 sont des fonctions à valeurs réelles de la variable réelle t, nous obtenons un système de la forme :

$$\left\{ \begin{array}{rcl} \frac{dy_1}{dt} &=& Re(\frac{d\Phi}{dt}(t) f(\Phi(t), y_1 + iy_2)) \\ \frac{dy_2}{dt} &=& Im(\frac{d\Phi}{dt}(t) f(\Phi(t), y_1 + iy_2)) \end{array} \right. \tag{2.6}$$

qui est un système dynamique de la forme (2.2).

Si nous considérons enfin $t_0 \in [t_1, t_2]$ tel que $\Phi(t_0) = x_0$, nous pouvons re-formuler la question posée précédemment de la manière suivante :

Si nous intégrons le système dynamique (2.6) pour des valeurs de t dans l'intervalle $[t_1, t_2]$, quelle est la solution qui possède comme condition initiale $(t_0, ((y_0^1, y_0^2))$, où $y_0 = y_0^1 + iy_0^2 \in \gamma$?

Nous pouvons noter que la notion de chemin d'intégration a disparu de la formulation ci-dessus, mais elle se trouve de manière implicite dans l'écriture du système. Il s'ensuit qu'avec les hypothèses précédentes, nous pouvons considérer la résolution d'une équation de la variable complexe de type (2.4) comme la résolution d'un système dynamique de dimension 2. En particulier, nous allons pouvoir appliquer le principe des portraits de phase à ce problème.

Notons cependant qu'en général le système (2.6) est *non autonome*, ce qui autorise des orbites issues de conditions initiales différentes à se couper, ce qui peut induire des difficultés de visualisation. De plus, le comportement souvent compliqué des fonctions complexes va encore rajouter des difficultés de visualisation à celles ci-dessus. Il s'ensuit la visualisation des solutions d'équations différentielles dans des portraits de phase ne sera exploitable qu'à la condition de trouver des formalismes de représentations adaptés, en particulier qui permettent de bien séparer graphiquement deux orbites de solutions différentes.

Le fait que le système ne soit pas autonome semble exclure la représentation des orbites sous forme de portrait de phase classique, au profit des portraits de phase étendus. En effet, les solutions du système (2.6) que nous voulons représenter sont des courbes paramétrées de $\mathbb{R}^2$:

$$\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} \tag{2.7}$$

lorsque t prend ses valeurs dans $[t_1, t_2]$. Pour éviter aux orbites correspondant à ces solutions de se croiser dans le plan, nous pouvons déplier ces courbes dans $\mathbb{R}^3$, en représentant chacun de leurs points par un point $P(t) \in \mathbb{R}^3$ ayant comme coordonnées :

$$\begin{pmatrix} y_1(t) \\ y_2(t) \\ t \end{pmatrix} \tag{2.8}$$

Ainsi, nous pouvons représenter les solutions de l'équation différentielle (2.4) comme un ensemble de courbes paramétrées dans l'espace, chacune étant issue d'une valeur sur le chemin γ, l'ensemble γ étant situé sur le plan d'équation $z = t_0$. Nous pouvons voir sur la figure 2.4 une courbe $\mathbf{R}$ issue d'une condition initiale.

Dans ce cas, $\mathbb{R}^3$ doit être considéré comme $\mathbb{C} \times \mathbb{R}$. Ainsi, la projection des courbes solutions sur les deux premières composantes de $\mathbb{R}^3$ (et d'un point de vue plus pratique la vue par dessus de la scène tridimensionnelle composée des courbes solutions) redonne les portraits de phase classiques.

Les courbes définies précédemment utilisent pleinement les trois dimensions réelles pour représenter la valeur des solutions en fonction de la valeur du paramètre t : cela laisse la possibilité de fournir d'autres informations en même temps que ces valeurs, par l'utilisation de la couleur notamment. Cela peut notamment permettre d'identifier la condition initiale d'où est issue une solution, ou de visualiser l'erreur numérique commise pendant l'intégration.

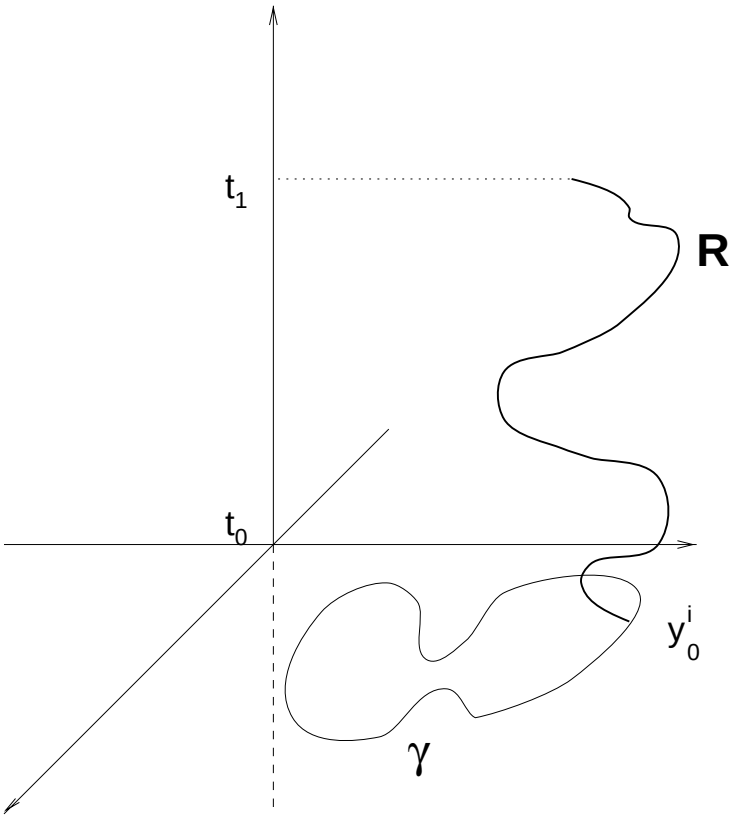


FIG. 2.4: Représentation 3D des solutions

2.2.4 Vision surfacique des résultats

Il peut être intéressant, par exemple pour évaluer la sensibilité des solutions de l'équation (2.4) aux conditions initiales, d'avoir une vision plus globale des solutions que le simple portrait de phase étendu. Une visualisation surfacique des solutions de l'équation peut être utile pour y arriver.

Considérons l'équation (2.4) comme un système dynamique de la forme (2.6). Le résultat suivant concernant le flot ψ_t du système dynamique (2.6) est alors valide :

2.2.4.1 Theorème Il existe $T \in \mathbb{R}_+$, tel que pour tout $t \in [t_0, t_0+T]$, le flot $\psi_t(\begin{pmatrix} y_0^1 \\ y_0^2 \end{pmatrix})$ est un difféormorphisme sur Ω.

Considérons l'ensemble γ du chapitre précédent, comme une courbe bornée et simple. Considérons , $\subset \mathbb{C}$ tel que $\gamma \subset$, . Nous pouvons alors définir la fonction $\Psi_{t,C}(y_0)$ de la manière suivante :

$$\begin{array}{rccl} \Psi_{t,C} & : & , & \to & \mathbb{C} \\ & & y_0 & \mapsto & \pi \circ \psi_t \circ \pi^{-1}(y_0) \end{array}$$

où π est la bijection qui à un vecteur de $\mathbb{R}^2$ fait correspondre son équivalent dans

$\mathbb{C}$, et ψ_t la fonction de flot du système dynamique associé. La fonction Ψ_t sera définie pour des valeurs de t dans l'intervalle défini par la paramétrisation du chemin C.

La fonction $\Psi_{t,C}$ transporte alors la courbe γ vers son image après intégration de l'équation différentielle sur une partie du chemin d'intégration C, paramétrée de t_0 à t à partir de la condition initiale (t_0, y_0).

D'après le résultat précédent, il existe $T \in \mathbb{R}_+$ tel que pour tout $t \in [t_0, t_0 + T]$ la fonction $\Psi_{t,C}(y_0)$ est un difféomorphisme sur , . De plus, l'image d'une courbe simple par un difféormorphisme est une courbe simple. Il existe alors $T \in \mathbb{R}_+, \forall t \in [t_0, t_0+T]$ tel que $\Psi_{t,C}(\gamma)$ est une courbe simple.

Nous pouvons donc composer une surface à partir de toutes les sections

$$(\Psi_{t,C}(\gamma))_{t\in[t_0,t_0+T]}$$

ces courbes étant dessinées, à t fixé, sur le plan d'équation $z = t$ pour $t \in [t_0, t_0 + T]$, comme sur la figure 2.5. Les courbes ascendantes composant cette surface sont paramétrées par le paramètre t, et chaque point aura les coordonnées suivantes dans $\mathbb{R}^3$:

$$\begin{pmatrix} Re(\Psi_{t,C}(y_0)) \\ Im(\Psi_{t,C}(y_0)) \\ t \end{pmatrix}$$

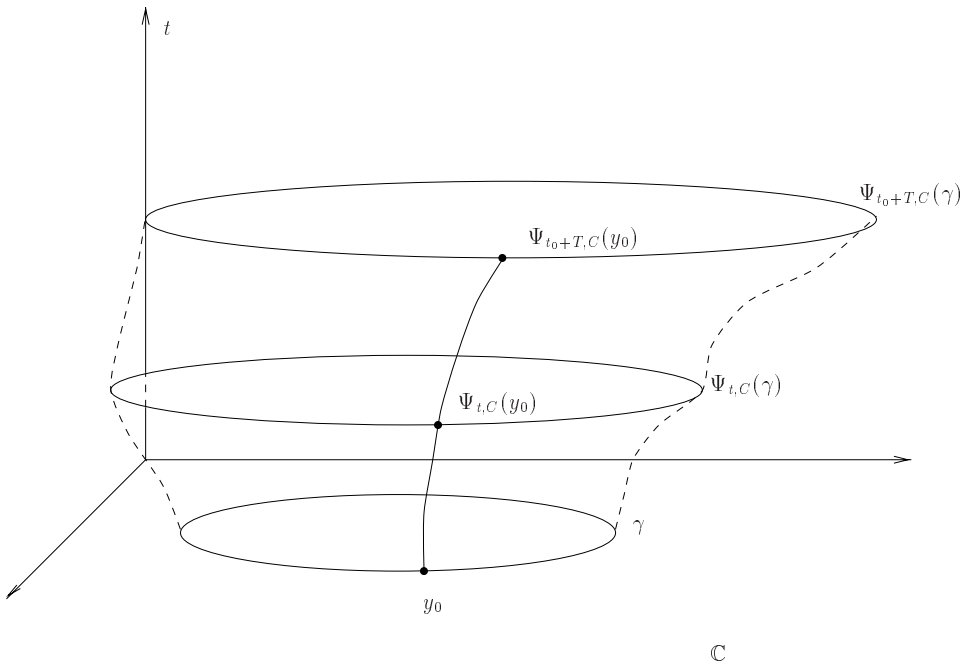


FIG. 2.5: La surface engendrée

Revenons à la formulation en systèmes dynamiques, et considérons τ_1 et τ_2 dans l'intervalle $[t_1, t_2]$, tel que $\tau_1 + \tau_2 \in [t_1, t_2]$. Le flot présente la particularité suivante :

$$\psi_{\tau_1} \circ \psi_{\tau_2} = \psi_{\tau_1+\tau_2}.$$

Considérons alors deux chemins complexes C_1 et C_2 paramétrés par Φ_1 et Φ_2 respectivement, chaque fonction étant définie sur les intervalles$[t_1, t_2]$ et $[t_2, t_3]$ respectivement.

Supposons de plus que C_1 et C_2 sont *consécutifs*, c'est-à-dire que

$$\Phi_1(t_2) = \Phi_2(t_2).$$

Dans ce cas, les chemins C_1 et C_2 peuvent être additionés, d'où

$$\begin{cases} \forall t \in [t_1, t_2] \quad , \quad \Psi_{t,C_1+C_2} = \Psi_{t,C_1} \\ \forall t \in [t_2, t_3] \quad , \quad \Psi_{t,C_1+C_2} = \Psi_{t,C_2} \circ \Psi_{t_2,C_1} \end{cases}$$

Cette relation va surtout être utile si Ψ_{t,C_1+C_2} est un difféomorphisme sur $[t_1, t_3]$, car dans ce cas cela signifie que deux surfaces peuvent être recollées pour obtenir la surface représentant l'intégration de l'équation différentielle sur la somme de deux chemins consécutifs.

Quelques renseignements intéressants pourront être tirés de ce mode de représentation :

- si nous considérons deux courbes issues de valeurs y_0 très proches dans γ, et si nous affectons à chacune de ces courbes une couleur identifiant la valeur de y_0 dont elle est issue, nous pourrons repérer sur la surface tracée la position de chacune de ces courbes. En particulier, si une couleur semble être concentrée sur une partie de la surface, cela signifiera que des conditions initiales différentes donneront sensiblement les mêmes résultats pendant l'intégration, alors qu'au contraire une diffusion de la couleur permettra d'identifier visuellement les zones sur γ où l'équation sera sensible aux conditions initiales.
- nous pourrons aussi repérer visuellement certaines valeurs de y_0 sur γ pour lesquelles la fonction $\Psi_{t,C}$ n'est pas un difféomorphisme sur l'intervalle de définition du paramètre t de C. En effet, pour de telles valeurs, nous verrons que l'ensemble $\Psi_{t,C}(\gamma)$ ne sera plus de même nature que γ, en particulier nous pourrons voir que la courbe $\Psi_{t,C}(\gamma)$ perd ses propriétés de continuité, et la surface en résultant en sera d'autant plus perturbée.

2.2.5 Complexité des résultats

Pour mesurer la quantité d'informations qui devraient être disponibles sur de tels portraits de phase étendus, considérons que nous voulons visualiser les solutions d'une équation différentielle de type (2.4), avec le formalisme décrit précédemment.

Considérons que nous voulons intégrer l'équation sur n chemins d'intégration (par exemple pour comparer les intégrations sur des chemins faisant le tour de singularités différentes), chacune étant échantillonnée avec p_i points, pour $0 \leq i < n$, et que l'ensemble γ est composé de m courbes différentes, chacune étant échantillonnée avec q_i points, pour i compris entre 0 et $m \Leftrightarrow 1$. Ainsi, si nous voulons représenter les données suivantes dans une même scène :

- les n courbes C,
- les m courbes composantes de γ,
- les solutions possédant comme condition initiale chacun des points de γ,

nous aurons alors à représenter

$$n + m + n \sum_{i=0}^{m-1} q_i$$

courbes solutions dans $\mathbb{R}^3$.

Considérons de plus que nous voulons une représentation surfacique de la visualisation. Dans ce cas, nous devrons tracer, pour chaque courbe solution un quadrangle par point de la courbe. Ainsi, nous aurons à tracer p_i quadrangles pour chaque courbe solution de l'intégration sur le chemin numéro i, soit

$$p_i \sum_{j=0}^{m-1} q_i$$

quadrangles pour le chemin d'intégration numéro i, et ainsi au total

$$\sum_{i=0}^{n} p_i \sum_{j=0}^{m-1} q_j$$

quadrangles pour l'ensemble de la scène en représentation surfacique.

Typiquement, si $n = 2$, $m = 2$ et $\forall i, p_i = 50$, 104 courbes dans $\mathbb{R}^3$ devront être représentées. De plus, si chaque chemin d'intégration est échantillonné avec 100 points, cela revient à tracer 10000 segments dans $\mathbb{R}^3$. Enfin, si nous traçons en même temps la représentation surfacique, cela reviendra à tracer dans ce cas environ 20000 quadrangles supplémentaires. C'est un nombre de données respectable, surtout si on veut les tracer sur un ordinateur général, non dédié à la représentation graphique tridimensionnelle.

De plus, vu la complexité de la scène à tracer (en particulier avec l'enchevêtrement prévisible des courbes résultats), nous devons pouvoir manipuler la scène afin de pouvoir en exploiter graphiquement les résultats. Enfin, le volume de quadrangles et de segments à tracer peut être facilement multiplié par deux si nous voulons représenter *en plus* de ces données d'autres données relatives à l'intégration, telle qu'une estimation de l'erreur commise pendant celle-ci.

2.3 Visualisation des solutions de systèmes différentiels complexes

2.3.1 Passage des équations aux systèmes

Un système différentiel d'ordre N dans le plan complexe est un système de la forme

$$\left\{\begin{array}{lcl} y_1' & = & f_1(x, y_1, \ldots, y_N) \\ \vdots & & \\ y_N' & = & f_N(x, y_1, \ldots, y_N) \end{array}\right. \tag{3.9}$$

où $x \in \mathbb{C}$, et $y_1, \ldots, y_N$ sont N fonctions de la variable complexe.

Les équations différentielles de degré N, c'est-à-dire les équations différentielles de la forme

$$y^{(N)} = f(x, y, \ldots, y^{(N-1)}), \tag{3.10}$$

peuvent s'exprimer sous forme de système de dimension N sur $\mathbb{C}$: en effet, pour une telle équation, on peut poser

$$\left\{\begin{array}{lcl} y_1 & = & y \\ \vdots & & \\ y_N & = & y^{(N-1)} \end{array}\right. \tag{3.11}$$

Dans ce cas, l'équation (3.10) se transforme en

$$\left\{\begin{array}{lcl} y_1' & = & y_2 \\ \vdots & & \\ y_{N-1}' & = & y_N \\ y_N' & = & f(x, y_1, \ldots, y_{N-1}) \end{array}\right. \tag{3.12}$$

Nous considérerons que les équations de type (3.10) s'écrivent sous forme de système de type (3.9) de manière naturelle, sans considérer les différents problèmes que peut poser cette réécriture.

Le problème auquel nous nous intéresserons est la transposition du problème détaillé dans le chapitre 2.2 à N dimensions :

- x est pris sur un chemin dans le plan complexe, C.
- les valeurs $(y_1^0, \ldots, y_N^0)$ telles que $(y_1(x_0), \ldots, y_N(x_0)) = (y_1^0, \ldots, y_N^0)$ seront prises sur un sous-ensemble γ_N de $\mathbb{C}^N$.
- nous étudierons différents modes de représentation pour la solution $(y_1(x), \ldots, y_N(x))$, afin d'en tirer un maximum de résultats sur les solutions de l'équation différentielle.

De la même manière que dans le chapitre précédent, considérons que le chemin C est paramétré par une fonction Φ du paramètre réel t :

$$\begin{array}{rccl} \Phi & : & [t_1, t_2] & \rightarrow & \mathbb{C} \\ & & t & \mapsto & \Phi(t), \end{array}$$

et que cette fonction Φ est telle que

$$\forall t \in [t_1, t_2], \frac{d\Phi}{dt} \neq 0$$

alors nous pouvons transformer l'équation i du système (3.9), pour $1 \leq i \leq N$:

$$\frac{dy_i}{dt} = \frac{d\Phi}{dt}.f_i(\Phi(t), y_1, \ldots, y_N) \tag{3.13}$$

et ainsi nous pouvons transformer le système (3.9) en un système dynamique de dimension $2N$ de la forme

$$\left\{ \begin{array}{rcl} \frac{dy_1^1}{dt} & = & Re(\frac{d\Phi}{dt}.f_1^1(\Phi(t), y_1^1 + iy_1^2, \ldots, y_N^1 + iy_N^2)) \\ \frac{dy_1^2}{dt} & = & Im(\frac{d\Phi}{dt}.f_1^2(\Phi(t), y_1^1 + iy_1^2, \ldots, y_N^1 + iy_N^2)) \\ \vdots & & \\ \frac{dy_i^1}{dt} & = & Re(\frac{d\Phi}{dt}.f_i^1(\Phi(t), y_1^1 + iy_1^2, \ldots, y_N^1 + iy_N^2)) \\ \frac{dy_i^2}{dt} & = & Im(\frac{d\Phi}{dt}.f_i^2(\Phi(t), y_1^1 + iy_1^2, \ldots, y_N^1 + iy_N^2)) \\ \vdots & & \\ \frac{dy_N^1}{dt} & = & Re(\frac{d\Phi}{dt}.f_N^1(\Phi(t), y_1^1 + iy_1^2, \ldots, y_N^1 + iy_N^2)) \\ \frac{dy_N^2}{dt} & = & Im(\frac{d\Phi}{dt}.f_N^2(\Phi(t), y_1^1 + iy_1^2, \ldots, y_N^1 + iy_N^2)) \end{array} \right. \tag{3.14}$$

en posant $y_i = y_i^1 + iy_i^2$ pour i dans l'intervalle $[1, N]$.

De même que dans le chapitre précédent, ce système dynamique n'a aucune raison d'être autonome, et donc sa représentation graphique sous forme de portrait de phase étendu sera nécessaire pour éviter à deux orbites issues de deux conditions initiales différentes de se couper dans l'espace de phase.

Si nous posons maintenant

$$Y(t) = \begin{pmatrix} y_1^1(t) \\ y_1^2(t) \\ \vdots \\ y_N^1(t) \\ y_N^2(t) \end{pmatrix}$$

et

$$Y_0 = \begin{pmatrix} y_1^{0,1} \\ y_1^{0,2} \\ \vdots \\ y_N^{0,1} \\ y_N^{0,2} \end{pmatrix}$$

les vecteurs de $\mathbb{R}^{2N}$ associés au problème, nous pouvons encore définir la fonction de flux sur $\Omega \subset \mathbb{R}^{2N}$ comme

$$\begin{array}{rccc} \psi_t & : & \Omega & \rightarrow & \mathbb{R}^{2N} \\ & & Y_0 & \mapsto & Y(t) \end{array}$$

où $Y(t)$ est la solution du système dynamique (3.14) telle que $Y(t_0) = Y_0$ (Y est la solution de ce système qui vérifie la condition initiale (t_0, Y_0)).

Nous pouvons encore définir la fonction Ψ_t sur un sous-ensemble , $_N$ de $\mathbb{C}^N$ par

$$\begin{array}{rccc} \Psi_{t,C} & : & , _N & \rightarrow & \mathbb{C}^N \\ & & Y_0 & \mapsto & \pi_N \circ \psi_t \circ \pi_N^{-1}(y_0) \end{array}$$

où π_N est la bijection de $\mathbb{C}^N$ dans $\mathbb{R}^{2N}$ qui fait correspondre à chaque composante du vecteur de $\mathbb{C}^N$ son équivalent dans $\mathbb{R}^2$, et ψ_t la fonction de flot du système dynamique de dimension $2N$ associé.

Le principe du chapitre précédent s'applique donc encore aux systèmes de dimension N, cependant, la complexité du problème a été multipliée par N : au lieu d'avoir à représenter un courbe solution dans $\mathbb{R}^3$ comme dans le chapitre précédent, il faut maintenant en représenter N. De manière classique [HLL97], la difficulté du problème est d'extraire des données pertinentes d'un amas de résultats bruts.

2.3.2 Différents modes de représentation

Pour limiter les difficultés de la visualisation des résultats, et ainsi maîtriser la complexité inhérente à l'étude de tels systèmes, il est impératif de trouver des méthodes de visualisation adaptées. La première qui vient à l'esprit est la généralisation en dimension N du principe décrit dans 2.2. Le portrait de phase associé au système dynamique (3.14) est l'ensemble des orbites du système dynamique, et existe naturellement dans $\mathbb{R}^{2N}$. Il se trouve donc que si $N > 1$, la visualisation directe ne sera pas forcément utile, à cause du nombre élevé de dimensions.

Considérons une solution Y du système dynamique (3.14) associé au système complexe (3.9) : cette solution s'écrit

$$Y(t) = \begin{pmatrix} y_1^1(t) \\ y_1^2(t) \\ \vdots \\ y_N^1(t) \\ y_N^2(t) \end{pmatrix} \in \mathbb{R}^{2N}$$

Visualisation "composante par composante"

Nous pouvons visualiser dans un premier temps, pour tout $i \in [0, N]$ et $j = 1, 2$, l'ensemble des courbes de $\mathbb{R}^2$:

$$t \mapsto y_i^j(t)$$

Cette visualisation composante par composante ne donne que des résultats partiellement intéressants : en effet, il est extrêmement difficile de pouvoir étudier le comportement *global* des composantes du système, en particulier les intéractions qui peuvent exister entre celles-ci.

Visualisation "composante par composante" complexe

De plus, pour tout $i \in [0, N]$, nous pouvons visualiser chaque courbe de $\mathbb{R}^3$:

$$\begin{pmatrix} y_i^1(t) \\ y_i^2(t) \\ t \end{pmatrix}$$

Cette représentation est la généralisation de la représentation des équations différentielles (*i.e. les systèmes différentiels de dimension* 1) définie dans le chapitre 2.2. La remarque précédente s'applique encore, c'est-à-dire que ce mode de visualisation ne donne qu'une petite partie des renseignements contenus dans la solution du système, car il ne prend en compte que les comportements d'une composante. Il restitue cependant mieux le comportement en tant que solution complexe du système, par le biais de l'identité qui existe entre un élément de $\mathbb{R}^2$ et $\mathbb{C}$.

Cas des équations d'ordre N

Si le système complexe a été obtenu à partir d'une équation d'ordre N, la première composante d'une solution du système peut être interprétée comme étant la solution de l'équation différentielle d'ordre N, et les autres composantes comme ses dérivées. Ainsi, le mode de représentation précédent peut s'appliquer naturellement à ce genre de systèmes, et la solution et ses dérivées peuvent être étudiées séparément de deux manières possibles :

– en tant que fonctions de $\mathbb{R}$ dans $\mathbb{C}$:

$$\begin{array}{rccc} y^{(j)} & : & [t1, t2] & \rightarrow & \mathbb{C} \\ & & t & \mapsto & y_{j+1}(\Phi(t)) \end{array}$$

– en tant que fonctions de $\mathbb{C}$ dans $\mathbb{C}$:

$$\begin{array}{rccc} y^{(j)} & : & C & \rightarrow & \mathbb{C} \\ & & x & \mapsto & y_{j+1}(x) \end{array}$$

Données primaires et secondaires

Les objets appartenant à un pavé de $\mathbb{R}^4$ peuvent être visualisés simplement grâce à l'utilisation de la couleur. Les objets considérés ici sont intrinsèquement des objets de $\mathbb{R}^3$, mais leur représentation en tant que tel fait disparaître une partie des informations qui pourraient en être extraites.

A titre d'exemple, si nous considérons les trois composantes d'une solution d'un système réel de dimension trois tel qu'un système de Lorenz, nous avons trois courbes de $\mathbb{R}$ dans $\mathbb{R}^3$, dont le tracé ne nous donne en fait que peu de renseignements sur la solution globale du système, et en particulier sur la géométrie des courbes de $\mathbb{R}^3$ que forme le portrait de phase de ce système.

Dans le cas des systèmes complexes, les courbes de $\mathbb{C}^2$ qui correspondent aux portraits de phase du système ne peuvent pas être tracées directement (sauf lorsque $N = 1$) , mais plusieurs informations à la fois peuvent être représentées sur une même courbe, à condition de pouvoir déterminer quelle information devra être visualisée de manière prioritaire.

Pour cela, considérons l'exemple d'une équation différentielle complexe d'ordre deux réécrite sous forme de système. Dans ce cas, il peut sembler naturel que cela soit la première composante de la solution du système qui est intéressante, mais il se trouve que l'influence de la dérivée de cette solution sur ses valeurs peut être étudiée, comme cela est possible sur un système de dimension deux réel. On peut dire à ce moment que la donnée primaire est la solution elle-même, et la donnée secondaire le module de la dérivée de cette solution (en fait de la deuxième composante d'une solution du système), ou l'argument de la dérivée de cette solution.

Une fois que l'information prioritaire est identifiée, elle peut être représentée dans $\mathbb{R}^3$, avec un formalisme de couleur qui donnera des renseignements sur la donnée secondaire. Dans l'exemple ci-dessus, chaque point de la courbe solution peut être tracé avec une couleur qui code le module de la dérivée de la solution. Pour cela, il faut quantifier les valeurs de la dérivée, comme cela à chaque valeur de la dérivée pourra être associée une couleur qui permettra d'estimer visuellement la valeur du module de cette dérivée, et ainsi de connaître partiellement les relations qui lient ces deux quantités.

Ce principe peut de plus s'appliquer à d'autres types de données que la dérivée dans le cas particulier d'une équation différentielle d'ordre N. A titre d'exemple, le principe précédent peut être appliqué aux expériences suivantes :

- étudier une composante de la solution du système, en fonction du module d'une autre composante.
- étudier une composante de la solution du système, en fonction du module de l'estimation de l'erreur globale numérique commise pendant l'intégration : cela permettra de voir si les résultats numériques calculés pendant l'intégration sont fiables, surtout lorsque le module de l'erreur est élevé (cf chapitre 11).
- étudier une composante de la solution du système, en fonction du logarithme du module de l'estimation de l'erreur relative commise pendant l'intégration.
- étudier une composante de la solution du système en fonction de la distance d'une autre composante par rapport à une singularité fixe du système.
- étudier le comportement de l'estimation de l'erreur relative commise pendant l'intégration sur la première composante, en fonction de la distance du chemin d'intégration à une singularité mobile du système pour une autre composante.

Cette liste d'expériences possibles n'est évidemment pas exhaustive : elle illustre la diversité des renseignements qui peuvent être tirés de l'étude graphique d'un système différentiel. Comme de plus ce système différentiel est de la variable complexe, cela induit des difficultés de visualisation qui ne sont pas focément rencontrées dans $\mathbb{R}$. Rappelons de plus que, pour le tracé d'un portrait de phase dans $\mathbb{C}$, environ 10000 segments et 20000 quadrangles peuvent être tracés pour en avoir une vision correcte. Ainsi, à chacune des expériences décrites ci-dessus, il faudra tracer environ $10000N$ segments et $20000N$ quadrangles. Cette complexité croissante peut ainsi motiver la conception d'un outil graphique adapté à ce genre de représentations massives.

Chapitre 3

Organisation logicielle

Nous avons vu dans les chapitres précédents quels types de données étaient susceptibles d'être mis en oeuvre et d'être visualisés dans une expérimentation sur une équation différentielle ou sur un système différentiel. Dans le cas particulier d'une implémentation logicielle, nous allons distinguer deux aspects dans la mise en oeuvre de moyens pratiques pour réaliser les expériences décrites dans le chapitre précédent : les *calculs* et la *visualisation graphique*.

3.1 Synthèse des chapitres précédents

Les expériences présentées dans les chapitres précédents peuvent être de natures très différentes : des intégrations dans le plan complexe, des évaluations de fonctions sur des chemins, des visualisations de fonctions et de solutions d'équations différentielles de la variable complexe. Les caractéristiques communes à ces expériences sont – elles aussi – de natures différentes :

- Ces expériences sont basées sur des environnements communs. Par exemple, une intégration dans le plan complexe et une évaluation de fonction sur un chemin complexe peuvent partager le même environnement de calculs permettant de gérer des nombres complexes, car ainsi les résultats des deux expériences peuvent être comparés. De la même façon, ces deux expériences peuvent partager le même outil de visualisation afin de pouvoir comparer visuellement les résultats, et d'en quantifier les différences.
- D'un point de vue plus pratique, ces expériences sont généralement très gourmandes, notamment en termes de puissance de calcul. Par exemple, bien que les calculs avec des nombres complexes consomment à peu près autant de puissance de calcul que les calculs avec des nombres réels, le fait de considérer des notions comme les indéterminations (définition 1.2.2.1) peut notablement augmenter la consommation en puissance de calcul. De même, l'exploitation graphique des résultats de certaines expériences peut se révéler très onéreuse en temps de calcul, surtout si les machines sur lesquelles sont réalisées les exploitations ne sont pas accélérées matériellement pour de telles tâches.

– Ces expériences doivent permettre la gestion de l'interactivité avec un utilisateur humain. Par exemple, les problèmes de résolutions d'indéterminations peuvent amener un utilisateur à spécifier certains résultats de calculs arithmétiques, de manière à ce que le résultat de ces calculs soit correct. De même, la spécification des éléments de base d'une expérience peut se faire de manière graphique, simplifiant ainsi la procédure des expériences.

Afin de définir précisément le rôle de chaque outil dans la mise en oeuvre des expériences, nous avons décidé d'adopter un modèle *modulaire*. Nous allons définir plusieurs types d'outils de calcul et de visualisation de manière indépendante. La nature des outils à intégrer pourrait être éventuellement complétée par la prise en compte de modules d'archivage, des liens avec des bases de données, etc ... Mais nous ne n'aborderons pas dans le cadre de cette thèse ces problèmes.

Ces outils peuvent s'insérer dans une plate-forme, c'est-à-dire un ensemble d'outils hétérogènes pouvant communiquer entre eux par le biais d'interfaces. Ainsi, un outil de calcul peut s'interfacer avec un outil de visualisation, par le biais de primitives de communication standards, mais aussi avec un autre outil de calcul, si ces outils ont été prévus pour une telle utilisation. Pour la mise en oeuvre de cette notion de plate-forme, nous avons utilisé les techniques et notions suivantes :

– La *programmation orientée objet*[Str95]. Cette technique de programmation permet, par opposition aux techniques de programmation usuelles, de considérer des objets auxquels sont associés certaines méthodes, plutôt que des fonctions prenant en compte un certain type d'objets. Ces objets peuvent être organisés de manière hiérarchique, c'est-à-dire que certains objets peuvent *hériter* des propriétés d'objets plus généraux, et les méthodes associées aux objets généraux peuvent être appliquées sans modifications aux objets hérités. L'utilisation de cette technique permet de gérer la généricité des objets implémentés, et en conséquence permet de définir *précisément* les différents modules qui entrent en jeu dans les expériences.

– La *répartition des ressources*, notamment sur des machines différentes et hétérogènes. Cela permet d'exploiter au mieux les ressources propres à chaque type de machines, et donc de bénéficier d'accélérations spécifiques à certains traitements. Par contre, cette notion passe par la définition précise du rôle de chacune des composantes de l'expérience, afin que les comportements des différentes composantes restent les mêmes quelle que soit la machine sur laquelle elles sont localisées.

– L'*interopérabilité* entre ces différentes composantes. Cela signifie que chaque composante peut être opérée par une autre, cela sans distinction de type de machine, de systèmes d'exploitation ou autres. À titre d'exemple, un outil de calcul doit pouvoir piloter un outil graphique, en générant des traces graphiques notamment. De même, un outil graphique doit pouvoir invoquer l'exécution d'un outil de calcul, à des fins de post-traitements prenant en compte une interaction avec l'utilisateur manipulant l'outil graphique.

Il est intéressant de remarquer que le transfert d'objets d'origine mathématiques entre plusieurs outils de calcul commence à être envisageable. Dans ce cadre, nous pouvons citer OpenMath [Ope97, AvLS96], qui présente une solution qui permet de donner une représentation générale d'objets par le biais de spécifications formelles de type SGML [SGM97]. OpenMath, en spécifiant les caractéristiques mathématiques des objets, permet à deux applications qui veulent échanger entre elles des résultats un codage standard des objets. Les concepteurs de ces applications ne doivent que prévoir des outils d'encodage et de décodage des objets vers les représentations internes de ceux-ci. En ce qui nous concerne (c'est-à-dire pour la conception d'outils de calculs et de visualisation intégrés dans une plate-forme) nous limiterons l'utilisation de ce principe à un échange *éventuel* d'objets mathématiques entre des outils de calcul. Cela signifie en particulier que nous définirons des outils de visualisation génériques, qui n'ont *a priori* aucune méthode pour visualiser des objets mathématiques, ces méthodes étant par contre implémentées dans les outils de calcul. En effet, notre objectif n'est pas de spécifier encore une fois un nouveau standard, mais d'obtenir une solution souple pour mettre en oeuvre les expériences que nous avons montrées dans les chapitres précédents. Cependant, la communication de données mathématiques entre différents outils de calcul n'a été qu'envisagée (c'est-à-dire qu'elle n'a pas encore donné lieu à un travail d'implémentation) dans le cadre de la plate-forme décrite ici.

Évidemment, les éléments d'une plate-forme sont succeptibles d'y être intégrés de manière dynamique, c'est-à-dire que pendant le déroulement d'une expérience, des nouveaux outils (de calcul, de visualisation) peuvent être créés, utilisés et détruits. Cette intégration de nouveaux outils peut être soit commandée par un utilisateur, soit effectuée de manière automatique.

À titre d'exemple, nous pouvons considérer la plate-forme décrite sur la figure 3.1. Les flèches représentent les actions que les différents modules peuvent être amenés à effectuer. Par exemple, une double flèche entre un outil de Calcul Formel et un outil graphique en deux dimensions peut représenter une plate-forme comme celle constituée par LAMEX[Ber97], qui permet de piloter Maple de manière interactive à partir d'un environnement Microsoft Windows, ou COMPASS [DJ95]. Ces outils peuvent eux-mêmes s'interfacer avec d'autres modules de calculs, ou d'autres modules graphiques.

Cette conception modulaire a déjà fait l'objet de nombreuses études. Nous pouvons en particulier noter les environnements suivants :

Izic : Cet environnement permet de bénéficier d'un modèle Client/Serveur pour la visualisation d'objets issus notamment des principaux systèmes de Calcul Formel. Le client ici est l'application qui s'exécute dans le système de Calcul Formel, qui émet des requêtes de type graphique à un serveur, qui lui-même peut être localisé sur une machine différente. Nous verrons par la suite que le modèle GANJ que nous avons implémenté possède à la base le même modèle, bien que nous avons étendu ce modèle pour qu'il soit plus puissant.

Oorange [GOP+96] : Cette plate-forme est elle aussi basée sur un environnement réparti, mais cette fois à base de transferts d'objets entre les différentes com-

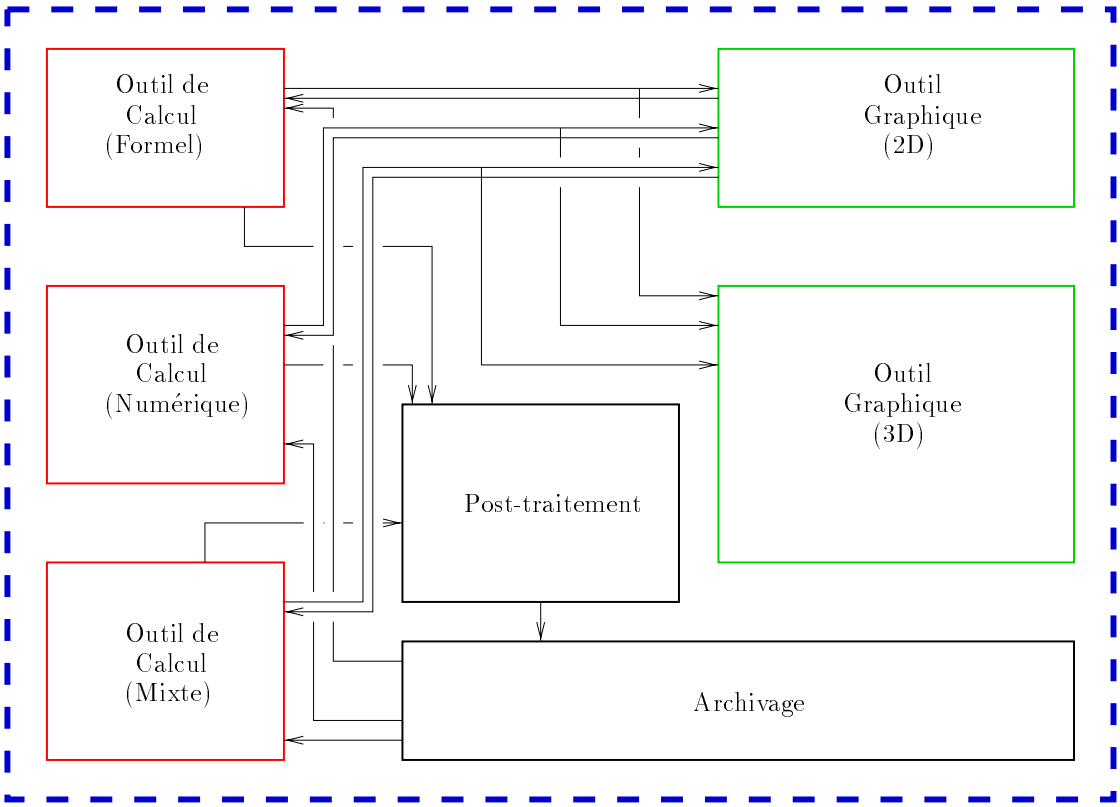


FIG. 3.1: Un exemple de plate-forme

posantes de l'environnement. Ainsi, un objet peut être soumis à un module de calcul afin d'y effectuer un traitement, puis le même objet peut être soumis à un module de visualisation afin d'être exploité graphiquement.

La solution que nous avons retenue est mixte, c'est-à-dire que nous avons retenu de la première démarche l'idée du Client/Serveur (pour la séparation des calculs de la visualisation), puis de la deuxième l'aspect dynamique de l'environnement.

3.2 Idée générale d'un outil de calculs

Les intérêts de la découpe en modules des outils de calculs sont multiples :

- Cette découpe permet de ne pas avoir à gérer des outils trop importants : en termes de volume de données à traiter ou en terme de puissance de calcul requise.
- La conception des ces outils peut être simplifiée par leur modularité. En effet, chaque module ne traite qu'un nombre limité de classes d'objets. Plutôt que d'effectuer un traitement spécifique à un objet par un appel de type procédural, donc interne à un programme, le traitement considéré peut être *commandé* à un autre outil, et donc pas forcément dans le même programme. Cette démarche

simplifie la conception d'un outil plus général, car elle est effectuée sur la base des résultats des autres outils.

- La conception de plusieurs outils de manière modulaire permet aussi de s'affranchir des contraintes de langages de programmation. Par exemple, un module de calcul basé sur des algorithmes récursifs pourra être programmé dans un langage permettant de gérer facilement et efficacement la récursivité, alors qu'un module basé sur des calculs intensifs pourra exploiter les optimisations d'un langage de plus bas niveau, Fortran par exemple.
- Dans le cas particulier d'une exploitation graphique des résultats, cette modularité permet de prévoir un traitement graphique adapté aux objets calculés, et donc d'avoir une démarche de type incrémentale dans la conception des traitements graphiques d'un objet : cette démarche sera utilisée pour la réalisation des méthodes de visualisation spécifiques aux chemins (chapitre 9).

Nous pouvons considérer l'exemple d'un outil de type LAMEX [Ber97]. Cet outil est prévu à la base pour effectuer des traitements spécifiques aux systèmes dynamiques, comme des détections de cycles limites, et d'en exploiter graphiquement les résultats. Cette application reçoit en entrée de la part d'un utilisateur les équations d'un système dynamique, ou les récupère dans un fichier. L'adjonction de méthodes particulières d'acquisition des expressions à traiter, par exemple par une description de type SGML, permettrait à cet outil de s'interfacer avec d'autres programmes, de manipulation symbolique d'équations par exemple, et donc de s'intégrer dans une plate-forme d'expérimentation, sans pour autant en compliquer la conception.

Nous pourrons examiner, notamment dans la partie II avec quelques programmes de calcul en nombres complexes, ainsi que lorsque nous aborderons les applications dans la partie IV, comment de tels outils peuvent être réalisés. La réalisation de ce type d'outils passe avant tout par la conception de primitives standard de communications, que les communications soient effectuées avec d'autres outils de calcul (cas de partage des données), soit avec des outils graphiques (cas de l'exploitation graphique). Ceci étant fait, chaque module pourra exploiter ses propres données, ainsi que celles des autres.

3.3 Premières spécifications d'un outil graphique

Les intérêts de l'intégration d'un outil graphique dans une plate-forme plus générale sont eux aussi multiples. La conception de méthodes de visualisation d'objets mathématiques notamment peut tirer plusieurs bénéfices d'un environnement modulaire :

- La séparation du calcul des graphiques permet de concevoir des outils graphiques généraux, et l'exploitation graphique des résultats de certains modules est indépendante des outils utilisés pour cette exploitation.
- La découpe en modules permet encore une fois de s'affranchir des contraintes de langages de programmation et des problèmes spécifiques à un outil graphique

particulier. Par exemple, cela peut permettre d'implémenter des algorithmes de calculs dans un langage spécialement adapté à cela (par exemple, Maple), et de visualiser leur résultat avec un outil écrit dans un autre langage de programmation, plus adapté à la programmation graphique (en C ou en C++).

– L'écriture des primitives graphiques s'en trouve elle aussi simplifiée, par la définition d'interfaces de niveau suffisamment bas. De plus, ces différentes primitives peuvent être adaptées pour chaque outil de visualisation, et ainsi permettre à différents outils graphiques de réaliser des exploitations des mêmes résultats, chacun exploitant ses ressources graphiques au mieux.

La visualisation mathématique n'a généralement pas de besoins aussi importants en termes de puissance graphique que d'autres domaines de l'informatique graphique : ce n'est pas tant le réalisme des images produites qui est cherché que celui d'une vision pertinente des objets mathématiques. Cette dernière remarque est particulièrement importante dans la démarche de spécification d'une plate-forme adaptée à la visualisation mathématique : en effet, nous ne sommes généralement pas limités par une puissance minimale pour exploiter graphiquement des données d'origine mathématique. En conséquence, des implémentations sur des machines *génériques*, c'est-à-dire des stations de travail ne possédant pas de cartes accélérant les opérations géométriques en trois dimensions aussi bien que des PC du commerce, doivent être prévues. De manière évidente, les outils de visualisation que nous prenons en compte doivent aussi bénéficier des accélérations matérielles proposées par certains types de matériels.

Nous verrons dans la partie III de ce document l'exemple d'un outil graphique, Ganj. Cet outil, destiné à la visualisation en trois dimensions, et particulièrement adapté à des machines non accélérées matériellement, est basé sur un modèle Client/Serveur. Cela permet d'offrir un jeu de primitives graphiques standard, que les outils de calculs pourront utiliser afin de générer des résultats graphiques de leurs résultats, ainsi que d'effectuer des traitements tels que des interactions avec l'utilisateur afin de re-spécifier leurs données de départ. Cet outil a fait l'objet d'une implémentation portable sous UNIX, sur des matériels tels que des stations de travail génériques (SUN, IBM) et à terme sur des PC du commerce.

En ce qui concerne une implémentation sur du matériel accéléré, celle-ci devrait être possible à peu de frais de programmation : cet outil est basé sur une implémentation logicielle du standard OpenGL [NDW93]. Ainsi, la partie du code du serveur Ganj appelant des primitives graphiques doit fonctionner de la même manière sur la totalité des machines sur lesquelles ce standard a été implémenté. Une première conséquence est que le travail de portage de ce serveur sur du matériel tel que des stations de travail possédant des cartes graphiques dédiées à l'accélération matérielles des opérations géométriques en trois dimensions (Silicon Graphics notamment) ne devrait pas poser de problèmes particuliers (autres que les différences de comportement des différentes stations de travail sous UNIX). Cependant, nous ne sommes pas convaincus qu'un travail de portage sur de telles stations soit nécessaire : en effet, une première implémentation de GLX [Kil96] devrait bientôt voir le jour *pour des machines génériques*. C'est-à-dire qu'un programme s'exécutant sur une machine peut

effectuer directement des requêtes OpenGL sur la console d'une autre machine, et les avantages matériels de cette dernière machine peuvent être utilisés. Jusqu'à présent, les bibliothèques permettant de générer des appels GLX étaient fournies uniquement par les constructeurs de matériel accéléré, mais une implémentation gratuite pour de nombreux types de machines est actuellement à l'étude. Ce type d'appels est évidemment moins efficace que des appels au niveau d'une machine possédant du matériel accéléré, mais représente un compromis en ce qui concerne l'exploitation graphique de scènes de complexité moyenne, telles que les scènes propres à la visualisation mathématique.

Deuxième partie

Un environnement de calculs en nombres complexes

Chapitre 4

Calculs en nombres complexes

Nous appellerons *calculs en nombres complexes* dans ce chapitre les calculs avec des nombres dans $\mathbb{C}$, mais aussi les calculs avec des éléments de surfaces de Riemann. Le fait d'étendre les calculs en nombres complexes à ces surfaces ne relève pas que d'une volonté de généralisation, mais représente bien une solution pour mieux calculer avec les nombres complexes.

4.1 Notations

Afin de différencier dans ce chapitre la nature des nombres manipulés (c'est-à-dire les nombres complexes usuels ou les éléments de surfaces de Riemann), il faut définir précisément quelques fonctions de base.

Nous proposons pour cela les notations dans le tableau suivant pour les fonctions complexes de base :

– l'exponentielle réelle :

$$\begin{array}{rccc} e & : & \mathbb{R} & \leftrightarrows & \mathbb{R}^{+*} \\ & & x & \leftrightarrows & e^x \end{array}$$

– le logarithme réel :

$$\begin{array}{rccc} \ln & : & \mathbb{R}^{+*} & \leftrightarrows & \mathbb{R} \\ & & x & \leftrightarrows & \ln(x) \end{array}$$

– la détermination principale de l'argument $\arg(x)$ d'un nombre complexe x non nul, tel que

$$\forall x \in \mathbb{C}^*, \arg(x) \in]\Leftrightarrow\pi, \pi]$$

– l'exponentielle complexe :

$$\begin{array}{rccc} \exp & : & \mathbb{C} & \leftrightarrows & \mathbb{C}^* \\ & & x+iy & \leftrightarrows & e^x(\cos y + i \sin y) \end{array}$$

La fonction exp n'est pas bijective dans le plan complexe, car elle n'est pas injective. Nous noterons $\mathbb{C}_k$, pour $k \in \mathbb{Z}$, les bandes horizontales du plan complexe, définies de la manière suivante :

$$\mathbb{C}_k = \{x + iy \in \mathbb{C}, y \in]\Leftrightarrow\pi + 2k\pi, \pi + 2k\pi]\}$$

La fonction $\exp_k : \mathbb{C}_k \Leftrightarrow\!\!\rightarrow \mathbb{C}^*$ est injective. On peut donc définir la fonction $\log_k$ comme fonction réciproque de la fonction $\exp_k$. On retrouve en particulier la *détermination prinicipale du logarithme*, définie sur $\mathbb{C}^*$, en considérant la fonction réciproque de la fonction $\exp_0$:

$$\begin{array}{rccl} \log & : & \mathbb{C}^* & \Leftrightarrow\!\!\rightarrow & \mathbb{C}_0 \\ & & z & \Leftrightarrow\!\!\rightarrow & \ln|z| + i\arg(z) \end{array}$$

4.2 Position du problème

Un problème classique avec les fonctions multiformes est celui de la détermination continue. Une fonction multiforme sur un domaine D est avant tout une fonction *holomorphe* en tout point de D, à l'exception des singularités de la fonction. Calculer une *détermination continue* d'une telle fonction sur un chemin C inclus dans D, et ne contenant *aucune* des singularités de la fonction, revient à calculer l'image point par point du chemin C. D'un point de vue théorique, cet ensemble image est un chemin inclus dans le plan complexe, mais d'un point de vue logiciel, le calcul d'une détermination continue de certaines fonctions multiformes n'est pas possible, à cause de la représentation interne des nombres complexes, qui ne permet de considérer que des nombres dont l'argument est restreint à $]\Leftrightarrow\pi, \pi]$. Les deux exemples suivants, concernant la racine et le logarithme, montrent que cette restriction empêche le calcul d'une détermination continue des deux fonctions.

Par exemple, le calcul d'une détermination continue de la fonction $\sqrt{z}$ présente de telles difficultés : la figure 4.1 montre le résultat d'un tel calcul, obtenu en calculant pour tout point z du chemin de départ C l'image de z. Le chemin C est parcouru dans le sens trigonométrique, en partant du point 1. La fonction utilisée effectue le calcul

$$\sqrt{z} = \sqrt{|z|}e^{i\arg(z)/2} \tag{2.1}$$

L'image du chemin C n'est pas un chemin, et donc le calcul n'a pas mené à une détermination continue de la fonction $\sqrt{z}$. Malheureusement, la fonction $\sqrt{z}$ est implémentée de telle manière (formule (2.1)) dans la totalité des systèmes de calcul formel et numérique, et donc cette fonction n'est pas utilisable simplement pour sa détermination continue. Le problème ici vient de la représentation interne des nombres complexes comme une paire de réels, codant la partie réelle et la partie imaginaire du nombre. Avec une telle représentation qui utilise implicitement la détermination principale de l'argument, l'argument des points du cercle passe brutalement de π à

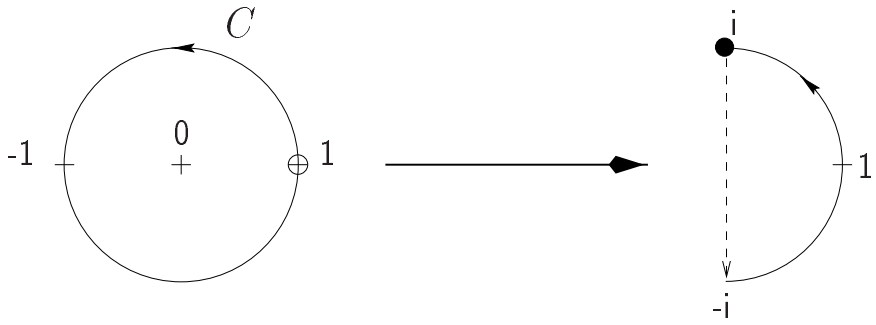


FIG. 4.1: Un essai de détermination continue de la racine carrée

$\Leftrightarrow\pi$ lorsque le cercle C coupe l'axe réel négatif. Une conséquence de ce phénomène est que, calculée de cette façon, la fonction $\sqrt{z}$ n'est pas continue sur le cercle C.

Nous pouvons également chercher à calculer la valeur du logarithme d'un élément de C en appliquant la formule ci-dessous :

$$\log(z) = \ln|z| + i\theta \ .$$

De même que dans l'exemple précédent, cette formule empêche le calcul d'une détermination continue sur le cercle C de la fonction log, car sur le même cercle C que précédemment (et parcouru de la même façon) , $|z| = 1$ et $\log(z) = i\theta$, et donc l'image par la fonction log de C n'est toujours pas un chemin.

De plus, les manipulations symboliques classiques, telles que $\log(a.b) = \log(a) + \log(b)$, ne sont même plus justifiées dans le plan complexe, ou alors au prix d'un effort supplémentaire, comme la considération de fonctions "unln" [Pat96], qui n'est pas spécialement adaptée aux calculs numériques.

Ces exemples illustrent les problèmes causés par le fait de prendre les arguments des nombres complexes dans l'intervalle $]\Leftrightarrow\pi, \pi]$, qui ne permettent pas de différencier les nombres selon la façon dont ils sont obtenus (c'est-à-dire pendant le parcours d'un chemin), et donc ne permettent pas de calculer naturellement les déterminations continues des fonctions multiformes. Nous allons considérer dans ce chapitre des *calculs en nombres complexes* faisant intervenir ce genre d'expressions. Cela signifie qu'étant donnés des nombres complexes de départ, des expressions (contenant éventuellement des fonctions multiformes) seront calculées à partir de ces nombres de départ, tous ces résultats étant éventuellement recombinés pour effectuer de nouveaux calculs. Il existe deux méthodes pour traiter le problème du calcul des valeurs d'une fonction multiforme sur un chemin :

- choisir une valeur de référence y_0 pour la fonction multiforme en un point particulier z_0 (i. e. une *détermination*), et calculer la valeur de la fonction en n'importe quel point z de $\mathbb{C}^*$ en effectuant un prolongement analytique (chapitre 1.2) sur un chemin reliant z_0 à z. Cette technique présente l'inconvénient d'être lourde à mettre en oeuvre (le prolongement analytique correspond à une vision locale du problème), et surtout *d'être dépendante du chemin considéré.*

– choisir une détermination particulière de la fonction, définie dans le plan complexe coupé (ce qui correspond, pour l'exemple du logarithme, à choisir la *détermination principale*), et s'y tenir tout au long du calcul. Cette solution présente l'avantage d'être simple à mettre en oeuvre, mais interdit tout espoir de considérer une détermination continue.

Ce chapitre présente un environnement de calcul (qui a été réalisé, sous forme de bibliothèques C++) et qui permet d'apporter une vision plus globale à ce genre de problèmes, de donner effectivement des résultats à certains calculs en nombres complexes.

4.3 Traitements classiques du problème

Le calcul de la fonction F (présenté dans [Asl96]) n'est en général pas traité correctement, que ce soit avec un système de calcul formel ou un système de calcul numérique, avec le modèle usuel des nombres complexes :

$$F(z,w) = \sqrt{zw} \Leftrightarrow \sqrt{z}.\sqrt{w} \tag{3.2}$$

En effet, l'évaluation de cette fonction par un système de calcul formel ou de calcul numérique sur un chemin ne mène pas à une détermination continue de la fonction. La représentation interne des nombres complexes comme paire de deux réels (la partie réelle et la partie imaginaire) entraîne des indéterminations dans le calcul des valeurs de F. Certaines de ces indéterminations peuvent être résolues en considérant les nombres complexes non plus comme un couple de réels, mais comme un couple de réels ainsi qu'un numéro qui permet de considérer des arguments sur $\mathbb{R}$ et non plus sur $]\Leftrightarrow\pi,\pi]$. Cette manipulation, classique au demeurant, peut avoir des avantages dans certains cas (pour la résolution des indéterminations sur les puissances notamment), mais peut aussi présenter des inconvénients, en particulier sur des opérations élémentaires telles que les additions de nombres complexes (voir [CJ96], p. 30).

Considérons l'évaluation numérique de l'expression (3.2), dans le cas particulier où $w = 1 + i$. D'après [Asl96], le *principal product excess* de z et w appartenant à $\mathbb{C}^*$ peut être défini par :

$$ppe(z,w) = \frac{\arg(zw) \Leftrightarrow \arg(z) \Leftrightarrow \arg(w)}{2\pi} \tag{3.3}$$

La fonction *ppe* ne peut prendre que les trois valeurs $\Leftrightarrow 1, 0, 1$ dans $\mathbb{C}$. De plus, si sa définition est étendue aux nombres complexes dont l'argument est pris dans $\mathbb{R}$, cette fonction est identiquement nulle. Son unique intérêt est qu'elle est non nulle lorsque l'argument du produit de deux nombres *renormalisé* dans $]\Leftrightarrow\pi,\pi]$ est différent de la somme des deux arguments des nombres de départ.

L'auteur en déduit l'égalité (3.4), qui prend en compte les manipulations à effectuer sur les arguments des nombres complexes considérés.

$$\sqrt{zw} = (\Leftrightarrow 1)^{ppe(z,w)}\sqrt{z}\sqrt{w} \tag{3.4}$$

Le tableau suivant présente les résultats d'une évaluation numérique de l'expression (3.2), avec $w = 1 + i$:

	ppe(z,1+i)	F(z,1+i)
$\Leftrightarrow\pi < arg(z) \leq 0$	0	0
$0 < arg(z) \leq \frac{3\pi}{4}$	0	0
$\frac{3\pi}{4} < arg(z) \leq \pi$	1	$\neq 0$ $\leftarrow\Leftrightarrow$

Ces résultats montrent les limitations du calcul des racines carrées avec des éléments de $\mathbb{C}$, c'est-à-dire des nombres dont l'argument est compris entre $\Leftrightarrow\pi$ et π. En effet, la normalisation des arguments sur les nombres dont l'argument est compris entre $\frac{3\pi}{4}$ et π, lorsque leur produit est effectué, entraîne que l'argument du résultat est supérieur à π et donc renormalisé pour appartenir à nouveau à l'intervalle $]\Leftrightarrow\pi, \pi]$. Cela introduit des discontinuités sur la fonction $\sqrt{z(1+i)}$. En particulier, si cette fonction doit être étudiée sur un cercle centré en 0, paramétrisé de la même manière qu'en 4.2, les valeurs obtenues ne sont pas cohérentes avec le fait que cette fonction est holomorphe sur $\mathbb{C}^*$, et donc interdisent une détermination continue.

Dans l'article [Asl96], l'auteur propose un jeu de tests pour les systèmes de calculs formels. Ces tests permettent, lors de manipulations symboliques d'expressions faisant intervenir des nombres complexes usuels, de gérer de manière formelle ces indéterminations.

L'auteur propose des règles de simplification de la forme suivante, où $P(z, w)$ représente une condition portant sur les positions respectives de z et w dans le plan complexe :

$$F_1(z, w) \Leftrightarrow F_2(z, w) = 0 \text{ si } P(z, w) \text{ est vraie.}$$

Les fonctions F_1 et F_2 sont toutes des combinaisons de logarithmes et d'exponentielles. Il s'ensuit que le membre de gauche dans l'expression ci-dessus va présenter des indéterminations, qui seront résolues de la manière usuelle dans $\mathbb{C}$, et l'égalité ci-dessus ne sera pas toujours vérifiée. Les résultats de ces tests peuvent être visualisés sur la planche 4.2, page 73 où les expressions $F_1(z, w)$ et $F_2(z, w)$ ont été évaluées pour différentes valeurs de z et w sur $\mathbb{C}$, et en dessinant un point à chaque valeur pour laquelle l'identité n'est pas satisfaite. Les valeurs seront prises régulièrement sur une grille de $\mathbb{C}$, typiquement pour $z = x + iy$ où

$$\begin{cases} x_{min} \leq x \leq x_{max} \\ y_{min} \leq y \leq y_{max} \end{cases}$$

Pour évaluer les expressions où des logarithmes et des fonctions puissances apparaissent, la détermination principale du logarithme a été utilisée, c'est-à-dire qu'étant donné $z \in \mathbb{C}$, $z^\alpha = \exp(\alpha \log(z))$. De plus, lorsque le test fait intervenir des fonctions de deux variables complexes, une valeur particulière est affectée à la deuxième variable (valeur donnée dans le tableau 4.1).

Ces tests montrent que les manipulations classiques sur les fonctions combinaisons de logarithmes et d'exponentielles ne sont en général plus autorisées avec des nombres

test	figure	F_1	F_2
test 1	4.2	$\sqrt{z(1+i)}$	$\sqrt{z}\sqrt{1+i}$
test 2	4.3	$\sqrt{z^2}$	z
test 3	4.4	$\sqrt{\exp(z)}$	$\exp(\frac{z}{2})$
test 4	4.5	$\log(z^2)$	$2\log(z)$
test 5	4.6	$\log(\exp(z))$	z

TAB. 4.1: Les tests d'Aslaksen

complexes. Cependant, la connaissance de ces divers phénomènes (comme notamment la connaissance du domaine de $\mathbb{C}$ où l'égalité $\sqrt{z^2} = z$) est particulièrement précieuse dans le cadre des systèmes de calculs formels en particulier. En effet, des informations complémentaires sur z peuvent servir à simplifier des expressions de la forme ci-dessus. De plus, la connaissance de ce type de domaines permet aussi de déterminer, dans des calculs de nature numérique, des zones "admissibles" pour les valeurs considérées.

Il est à noter cependant que ce principe n'a pas donné lieu à une implémentation, et donc ne se prête guère à des tests, afin d'en vérifier la validité. De plus, il revient dans tous les cas à une détermination "à la main" (et non pas de manière automatique) des ensembles de validité des règles de simplification, et n'est donc que peu adapté à l'évaluation automatique d'expressions arithmétiques.

Ces résultats montrent de manière plus générale que la représentation des nombres complexes comme éléments de $\mathbb{R}^2$, en particulier avec l'interprétation de l'argument comme un angle entre l'axe réel et le vecteur représentant le nombre, possède ses propres limitations, notamment pour les calculs d'expressions contenant des exponentielles et des logarithmes.

4.4 La surface de Riemann du logarithme

Le surface de Riemann du logarithme est définie comme étant un recouvrement de $\mathbb{C}$ (voir [Bea84], p.46 pour plus de détails) par la fonction

$$z \Leftrightarrow\!\!\rightarrow \exp(z)$$

Ainsi, cette surface est un ensemble sur lequel la fonction exp est bijective et analytique. Topologiquement, cette surface peut être vue comme une infinité dénombrable de copies de $\mathbb{C}$, les *feuillets*, dont deux exemplaires consécutifs sont "recollés" par l'axe réel négatif (les *coupures*). Cette surface sera notée $\mathcal{M}_{\log}$ dans la suite.

Nous utiliserons par la suite la fonction

$$\begin{array}{rccc} \mathrm{Arg} & : & \mathcal{M}_{\log} & \Leftrightarrow\!\!\rightarrow & \mathbb{R} \\ & & z = (p,k) & \Leftrightarrow\!\!\rightarrow & \mathrm{Arg}(z) = \arg(p) + 2k\pi \end{array}$$

Un élément z de cette surface est donc parfaitement défini par la donnée de (p,k), où $p \in \mathbb{C}$ et $k \in \mathbb{Z}$, k représentant le numéro de feuillet de z.

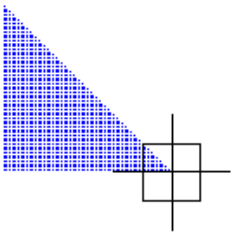


FIG. 4.2: Le test 1 (Aslaksen)

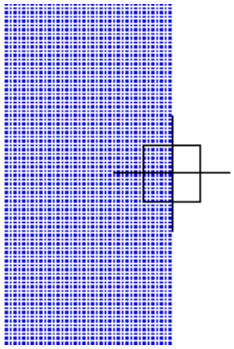


FIG. 4.3: Le test 2 (Aslaksen)

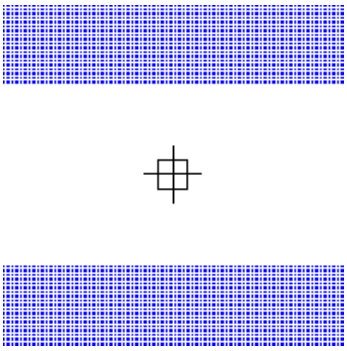


FIG. 4.4: Le test 3 (Aslaksen)

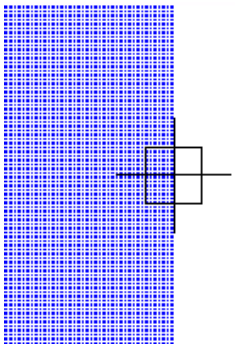


FIG. 4.5: Le test 4 (Aslaksen)

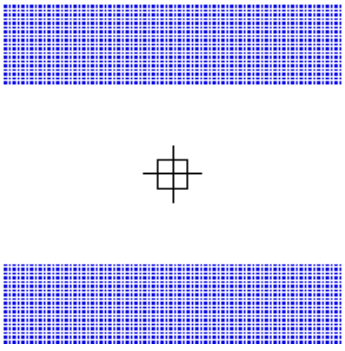


FIG. 4.6: Le test 5 (Aslaksen)

TAB. 4.2: Les tests d'Aslaksen

Ainsi le fait de considérer des arguments *modulo* 2π revient à considérer des nombres complexes de manière classique. Nous pouvons alors définir, pour tout $k \in \mathbb{Z}$,

$$U_k = \{z \in \mathcal{M}_{\log} \Leftrightarrow \pi + 2k\pi < \mathrm{Arg}(z) \leq \pi + 2k\pi\}$$

et

$$\Pi_k(z) = p$$

La surface du logarithme définie de la manière précédente est bien une surface de Riemann. Dans la suite, nous identifierons $\mathbb{C}$ à U_0.

Le choix du nombre π dans la détermination des arguments des éléments de la surface de Riemann du logarithme est arbitraire : de manière classique en analyse complexe, il est choisi comme valeur limite de la détermination principale de l'argument (la direction correspondante est la direction de coupure). Cela implique que la définition des ouverts U_k est liée à la détermination principale de l'argument d'un nombre complexe. De manière plus générale, les ouverts U_k^θ peuvent être considérés, pour tout $k \in \mathbb{Z}$, et pour $\theta \in]\Leftrightarrow\pi, \pi]$:

$$U_k^\theta = \{z \in \mathcal{M}_{\log}, \theta + 2(k \Leftrightarrow 1)\pi < \mathrm{Arg}(z) \leq \theta + 2k\pi\}$$

pour définir la surface de Riemann $\mathcal{M}_{\log}$, mais ce choix n'est pas spécialement adapté au calcul des arguments. Par contre, tous ces ouverts doivent être d'ouverture *strictement inférieure* à 2π.

4.5 Intérêt des surfaces de Riemann pour les calculs en nombres complexes

Nous allons exploiter les propriétés inhérentes à cette surface de Riemann particulière qu'est la surface du logarithme pour faire des calculs en nombres complexes. Le logarithme peut être défini de manière *univalente* (c'est-à-dire non multiforme) sur cette surface, ce qui peut être utile pour avoir une détermination continue de cette fonction, et donc pour avoir des résultats cohérents à certains calculs.

Une des difficultés rencontrée dans ce chapitre réside dans le fait que, puisque $\mathcal{M}_{\log}$ est une surface de Riemann, il est difficile de définir de manière claire ne serait-ce que l'addition de deux éléments de cette surface. Mais ce problème peut être résolu dans certains cas, soit de manière *intrinsèque*, soit en faisant intervenir des connaissances mathématiques complémentaires sur les nombres considérés.

Dans la suite, nous noterons $z \in \mathcal{M}_{\log}$ comme un couple (p, k) où $p \in \mathbb{C}$ et $k \in \mathbb{Z}$. p sera la projection de z sur $\mathbb{C}$ par le biais des projections Π_k associées à la surface de Riemann $\mathcal{M}_{\log}$, et k étant son numéro de feuillet.

4.5.1 Définition du logarithme et de l'exponentielle

Le logarithme sur $\mathcal{M}_{\log}$, noté Log, peut être défini de manière holomorphe :

$$\begin{array}{rccc} \mathrm{Log} & : & \mathcal{M}_{\log} & \leftrightarrows & \mathbb{C} \\ & & z & \leftrightarrows & \log(p) + 2ik\pi, \end{array}$$

L'exponentielle à valeurs dans $\mathcal{M}_{\log}$, notée Exp, peut être définie comme fonction inverse de la fonction logarithme, de la manière suivante :

$$\begin{array}{rccc} \mathrm{Exp} & : & \mathbb{C} & \leftrightarrows & \mathcal{M}_{\log} \\ & & p = x + iy & \leftrightarrows & (\exp(p), \lceil \frac{y-\pi}{2\pi} \rceil) \end{array}$$

On a ainsi

$$\mathrm{Arg}(\mathrm{Exp}(p)) = y = \arg(\exp(p)) + 2k\pi$$

De cette manière, la fonction exponentielle est parfaitement définie sur $\mathbb{C}$. Pour déterminer le numéro de feuillet de l'image d'un complexe z par cette fonction, il suffit de déterminer dans quelle bande horizontale se trouve ce nombre z, comme précisé sur la figure 4.7.

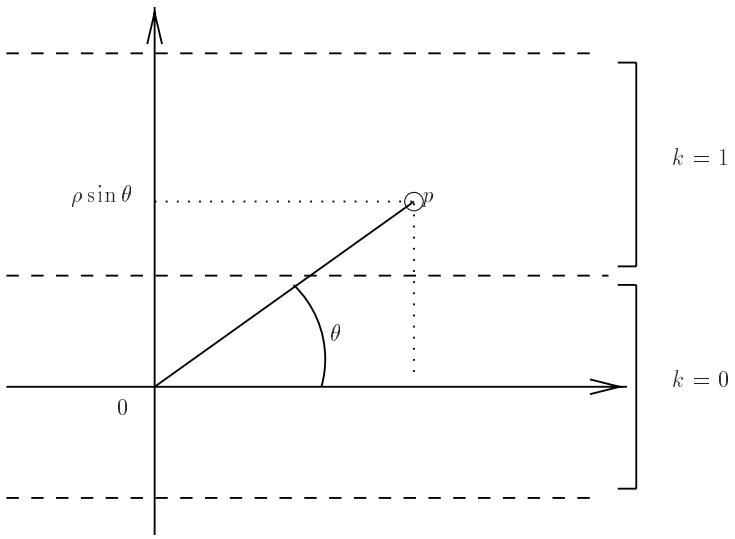


FIG. 4.7: Les bandes horizontales de la fonction exp

De plus, avec les définitions des fonctions Log et Exp ci-dessus, les deux identités suivantes sont vérifiées :

$$\begin{array}{rcl} \forall p \in \mathbb{C} & , & \mathrm{Log}(\mathrm{Exp}(p)) = p \\ \forall z \in \mathcal{M}_{\log} & , & \mathrm{Exp}(\mathrm{Log}(z)) = z \end{array}$$

Nous voyons ainsi que, d'un point de vue mathématique, nous avons sur $\mathcal{M}_{\log}$ une définition des fonctions exponentielle et logarithme. Mais les ensembles de définition de ces fonctions peuvent poser des problèmes de calcul. En effet, comment définir la fonction $e_2(z) = \mathrm{Exp}(\mathrm{Exp}(z))$ sur $\mathbb{C}$? Lorsque z est situé dans une bande autre que celle correspondant à $k = 0$, $\mathrm{Exp}(z)$ n'est plus un élément de $\mathbb{C}$, et mathématiquement $e_2(z)$ n'est pas définie. Or il se trouve que dans un calcul en nombres complexes usuels, rien n'interdit de considérer cette valeur. C'est un cas typique d'*indétermination*, c'est-à-dire que nous ne savons pas donner *a priori* une valeur à ce calcul, bien que ce calcul soit justifié dans $\mathbb{C}$. Ce paradoxe peut être gênant pour conduire un calcul avec des éléments de surfaces de Riemann, et il est donc particulièrement important de détecter ces indéterminations pendant le déroulement d'un calcul. Pour cela, il faut bien cerner le problème pour voir à quel moment d'un calcul une indétermination peut se produire.

4.5.2 Opérations arithmétiques usuelles

La multiplication de $z_1 = (p_1, k_1)$ et $z_2 = (p_2, k_2)$ est parfaitement définie de la manière suivante : soit $z = z_1 * z_2$, $z = (p, k)$. Nous avons alors

$$\begin{aligned} p &= p_1 * p_2 \\ \\ \mathrm{Arg}(z) &= \mathrm{Arg}(z_1) + \mathrm{Arg}(z_2) \\ &= \arg(p_1) + \arg(p_2) + 2(k_1 + k_2)\pi \\ &= \arg(p) + 2k\pi \end{aligned}$$

Dans ce cas, le nombre k associé à z sera $k_1 + k_2$, éventuellement augmenté ou diminué de 1 dans le cas où $|\arg(p_1) + \arg(p_2)| > 2\pi$.

Par contre, l'addition peut poser des problèmes dans certains cas. Considérons

$$z_1 = (p_1, k_1)$$

et z_2 sur le même feuillet que z_1 :

$$z_2 = (p_2, k_1)$$

Le principe de l'addition de deux nombres complexes usuels s'applique alors ici. En effet, si nous notons $z = z_1 + z_2$, $z = (p, k)$, nous pouvons aisément calculer

$$p = p_1 + p_2$$

et choisir k de telle manière que

$$\mathrm{Min}(\mathrm{Arg}(z_1), \mathrm{Arg}(z_2)) < \mathrm{Arg}(z) < \mathrm{Max}(\mathrm{Arg}(z_1), \mathrm{Arg}(z_2))$$

car la détermination de $\mathrm{Arg}(z)$ est unique dans ce cas : cela revient à choisir $k = k_1$. Ce choix est présenté sur la figure 4.8. Dans ce cas, l'addition est *locale*, car z_1 et z_2 appartiennent au même feuillet k_1, et z peut être vu comme $\Pi_{k_1}^{-1}(p_1 + p_2)$.

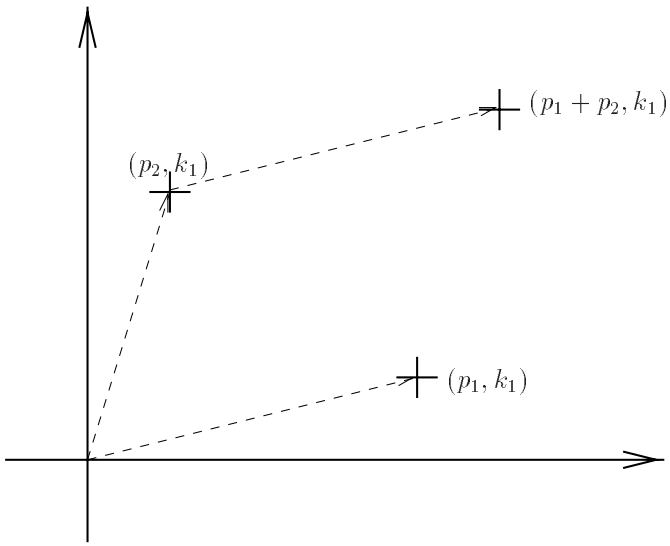


FIG. 4.8: Addition de deux complexes, sur le même feuillet

Grâce à un argument de continuité sur la surface, l'addition de deux nombres sur deux feuillets consécutifs peut être encore justifiée dans le cas particulier où

$$|\mathrm{Arg}(z_2) \Leftrightarrow \mathrm{Arg}(z_1)| < \pi$$

En effet, la règle décrite sur la figure 4.8 fonctionne encore, car il suffit de considérer l'addition au sens usuel des deux nombres complexes, et de lui affecter un numéro de feuillet en fonction de l'argument du nombre complexe résultant de l'addition des deux nombres : un exemple est donné sur la figure 4.9. Sur cette figure, l'addition usuelle des deux nombres z_1 et z_2 donne un nombre complexe dont l'argument est plus proche de celui de z_2 que de celui de z_1 : l'indétermination peut être facilement résolue en imposant comme numéro de feuillet pour la somme $z_1 + z_2$ le numéro de feuillet de z_2.

Par contre, dans le cas où

$$|\mathrm{Arg}(z_2) \Leftrightarrow \mathrm{Arg}(z_1)| \geq \pi$$

nous avons une indétermination sur le choix de l'argument correspondant à $z_1 + z_2$. Ce cas est présenté sur la figure 4.10. Si nous supposons que, sur cette figure, $k_2 = k_1 + 1$, la différence entre les arguments de z_1 et de z_2 est effectivement supérieure à π. Le choix entre k_1 ou k_2 (ou entre ces deux valeurs) est complètement arbitraire pour le numéro de feuillet de $z_1 + z_2$.

Ce cas de figure peut être détecté à tout moment pendant un calcul, ce qui fait que cette indétermination n'est pas forcément pénalisante. Lorsqu'une telle indétermination est détectée, il n'y a en général pas de solution intrinsèque (c'est-à-dire faisant appel à des connaissances sur les surfaces de Riemann) au calcul. Mais dans des cas particuliers, nous pouvons avoir des renseignements *a priori* sur les résultats du calcul, que nous pouvons exploiter pour résoudre ces indéterminations : par exemple,

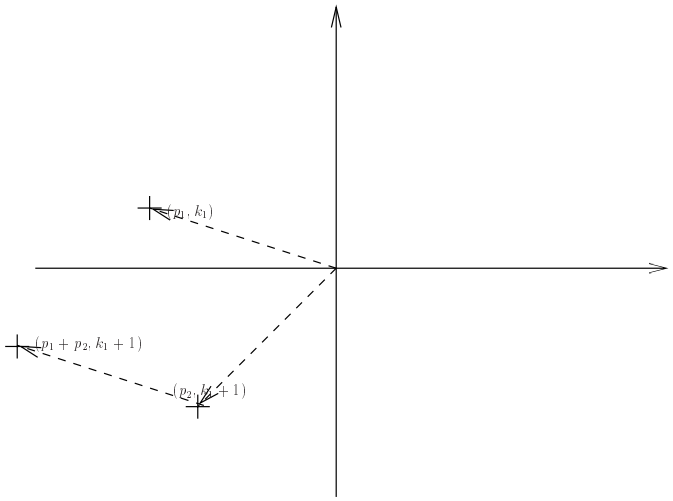


FIG. 4.9: Addition de deux complexes, sur des feuillets différents (premier cas)

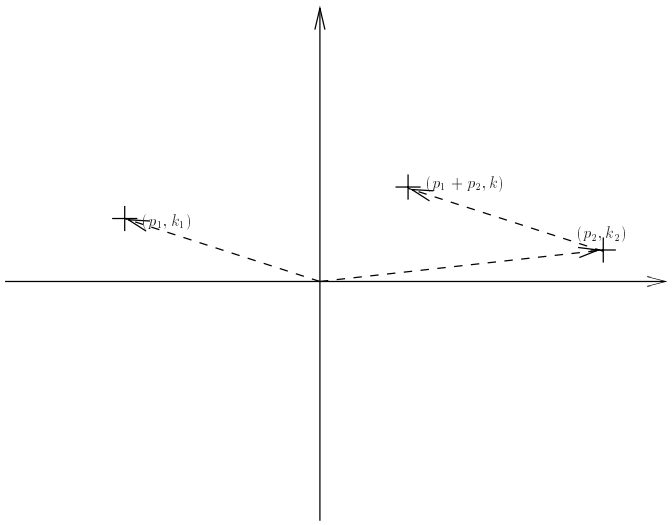


FIG. 4.10: Addition de deux complexes, sur des feuillets différents (deuxième cas)

lorsque la valeur résultat de l'addition est sensée représenter la valeur d'une fonction continue en un point, la valeur de la fonction aux points caclulés précédemment peut aider à déterminer la valeur du résultat indéterminé.

La *résolution* de ces indéterminations (c'est-à-dire la détermination d'une valeur cohérente avec le reste du calcul) peut être traitée de manière logicielle.

4.6 Un modèle d'environnement de calculs

D'un point de vue logiciel, deux classes de nombres complexes ont été effectivement implémentées :

- la première (*i.e.* la classe `Complex`) permet de traiter les nombres complexes usuels : en effet, les méthodes s'appliquant aux nombres complexes usuels ne sont pas forcément adaptables aux éléments de la surface de Riemann du logarithme, et les nombres complexes usuels servent donc de solution de secours en cas d'incompatibilité.
- la seconde (*i.e.* la classe `Riemann`) permet de traiter les éléments des surfaces de Riemann décrits précédemment, ainsi que d'intercepter[1] les éventuelles indéterminations.

Lorsque les nombres considérés sont des éléments de la surface de Riemann du logarithme, deux types de traitements sont possibles :

- les calculs directement avec ces nombres : cela permet de bien définir les fonctions Log, Exp, ... à condition de bien spécifier les nombres de départ (et en particulier leur numéro de feuillet).
- les calculs avec des nombres complexes usuels, mais en effectuant des traitements *a posteriori* pour transformer les résultats comme des éléments de surfaces de Riemann. Ceci peut être une solution de remplacement lorsque certains codes ne peuvent pas prendre en compte les éléments de la surface de Riemann du logarithme.

A chaque phase d'un calcul (spécification, opérations arithmétiques et exploitation des résultats), peut être associé un traitement des indéterminations, qui en cas d'échec peut demander à l'utilisateur une aide, voire même interrompre le calcul. Le principe est montré sur la figure 4.11.

Sur cette figure, les doubles flèches indiquent les phases où une consultation de l'utilisateur est requise. Le calcul commence par une phase de spécification, puis les calculs proprement dits sont effectués. Il peut y avoir une première source d'indétermination, qui nécessite un premier traitement : soit le mécanisme de résolution des indéterminations trouve la solution (par le biais de renseignements externes au programme, comme notamment des connaissances complémentaires sur les données du problème, comme par exemple la continuité d'une fonction), soit il consulte l'utilisateur, soit il sort du programme. Ensuite, lorsque le calcul est achevé, la phase de post-traitement peut servir à remettre en forme les données, afin de relancer un nouveau calcul notamment. Le mécanisme de résolution des indéterminations est encore présent dans cette phase.

La sortie du calcul est appelée dans les cas extrêmes : lorsque l'utilisateur ne sait pas résoudre lui-même l'indétermination. Dans ce cas, considérant que le calcul n'a plus aucun sens, l'exécution peut être interrompue.

Il est important de laisser le maximum de flexibilité à l'utilisateur, car il lui est souhaitable de ne pas interrompre un calcul s'il est capable de le corriger par la suite, en particulier à l'aide de raisonnements mathématiques. De plus, l'utilisateur doit savoir à quel endroit dans son programme se trouve l'indétermination. La solution idéale serait qu'il puisse connaître le numéro de la ligne où le calcul est indéterminé.

[1] Ici, intercepter est à considérer de la même manière qu'une exception en C++.

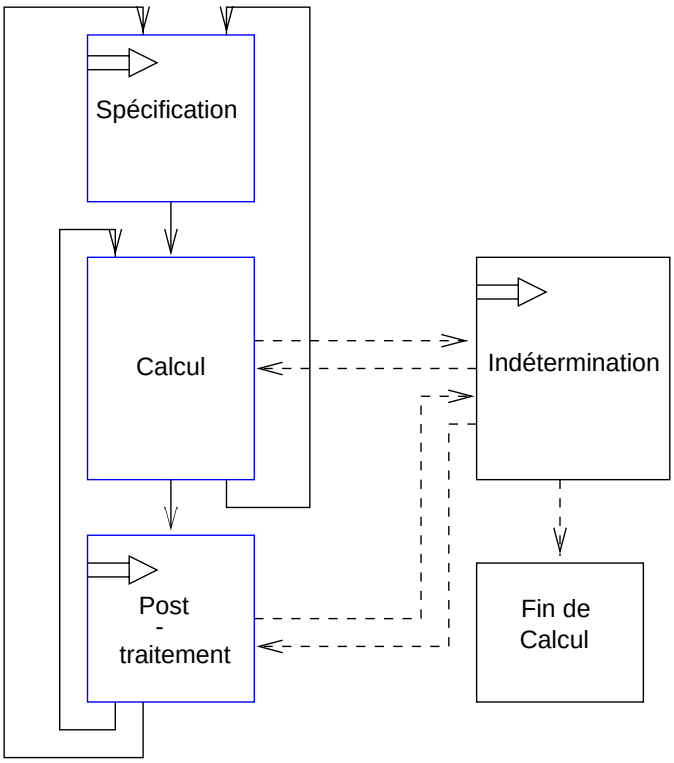


FIG. 4.11: Le principe du calcul en nombres complexes

Dans l'environnement présenté ici, l'utilisateur doit repérer les zones sensibles de son code, à savoir celles où des indéterminations peuvent se produire (typiquement, des évaluations de fonctions exponentielles, ou des additions de deux nombres complexes quelconques). Lorsqu'une indétermination est détectée (au niveau des opérateurs redéfinis dans la classe `Riemann`), le mécanisme de résolution de l'indétermination est alors activé : le calcul est interrompu, mais les valeurs ayant provoqué l'indétermination sont stockées, de manière à pouvoir être réutilisées par la suite. Ce mécanisme de résolution a été implémenté grâce aux exceptions en C++. La section "critique" du code doit être encapsulée dans un bloc **try**, alors que la méthode de résolution est encapsulée dans un bloc **catch**. L'exemple suivant illustre ce principe :

```
try {
   z = x + y ;
} catch (PlusIndetermination& E) {
   z = E.Solve ( Indermination::ARBITRARY) ;
}
```

La méthode de résolution (`PlusIndetermination::Solve`) est appelée avec un paramètre : ce paramètre permet de déterminer de quelle manière doit être résolue l'indétermination. Dans l'exemple ci-dessus, l'indétermination doit être résolue de

manière purement arbitraire : le numéro de feuillet du premier nombre (`x`) est affecté au résultat. Deux autres méthodes sont proposées par défaut : la méthode `FATAL`, qui interrompt l'exécution du programme, et la méthode `INTERACTIVE`, qui interrompt l'exécution, et demande de manière interactive à l'utilisateur quel numéro de feuillet doit être affecté au résultat, puis reprend l'exécution du programme.

Si l'utilisateur a des connaissances complémentaires sur la nature mathématique du problème, il peut intégrer ses connaissances à son calcul en programmant son propre mécanisme de résolution d'indéterminations :

```
try {
   z = x + y ;
} catch (PlusIndetermination& E) {
   z = E.Solve ( Indermination::ARBITRARY) ;
   z.F_ = MySheetNumberDetermination ( x , y , z ) ;
}
```

Ainsi, ce traitement particulier n'est invoqué que lorsque des indéterminations ont été détectées. Le chapitre 5 présente un exemple d'intégrateur dans le plan complexe utilisant ce mécanisme d'exceptions, avec une méthode de résolution des indéterminations adaptée aux problèmes d'intégrations d'équations différentielles.

4.7 Tests et applications : les "clearcut regions"

La "clearcut region" d'une fonction de la variable complexe f est un ensemble noté $clearcut(f)$ et défini dans [CJ96] de la manière suivante :

$$clearcut(f) = \{z \in \mathbb{C}, \log(\exp(f(z))) = f(z)\}$$

D'après l'expression ci-dessus, une telle région du plan complexe contient les points du plan complexe dont l'image par f appartient à la bande $\mathbb{C}_0$: en effet, si pour $z \in \mathbb{C}$, $f(z) \in \mathbb{C}_k$ avec $k \neq 0$, alors $\log(\exp(f(z))) \in \mathbb{C}_0$, car log est la détermination principale du logarithme (les deux valeurs seront égales à $2ik\pi$ près). Par suite, $\log(\exp(f(z))) \neq f(z)$.

Dans ce paragraphe, nous allons chercher à étendre cette notion de "clearcut region" à des fonctions multiformes, c'est-à-dire à des fonctions définies de $\mathcal{M}_{\log}$ dans $\mathcal{M}_{\log}$. Soit donc f une telle fonction. Nous proposons d'évaluer f en tout point de $\mathbb{C}$ (ou plus précisément d'une grille discrétisée) en le considérant comme élément de U_0 (défini page 74) et en utilisant l'arithmétique présentée précédemment. Puis nous présenterons les résultats sous forme graphique en affichant le point z en couleur de la manière suivante :

	$f(z) \in U_0$	$f(z) \notin U_0$
$\Pi_k(f(z)) \in \mathbb{C}_0$	Noir	Vert
$\Pi_k(f(z)) \notin \mathbb{C}_0$	Bleu	Rouge

Ici, k est défini par le numéro de feuillet de $f(z)$ pour le point z considéré. L'algorithme utilisé pour la détermination d'une couleur d'un point est présenté page 83. Les tests présentés dans le tableau 4.4 peuvent être visualisés sur la planche 4.5, page 85. En particulier, les fonctions puissances de la forme z^α sont calculées en utilisant la formule

$$z^\alpha = \mathrm{Exp}(\alpha \mathrm{Log}(z))$$

Sur les sorties graphiques présentées sur la planche 4.5, les quatre couleurs (noir, bleu, vert, rouge) sont toujours représentées, par contre la couleur indéfinie (vert vif) n'apparaît que sur les figures 4.14, 4.15, 4.17 et 4.18. Pour l'exemple de la fonction $\mathrm{Exp}(z)$ (test 1) et pour l'exemple de la fonction z^3 (test 4), cela est normal : ces fonctions sont parfaitement déterminées pour $z \in \mathbb{C}$. De plus, il est facile de retrouver à partir de ces figures les "clearcut regions" des fonctions $\exp(z)$ et z^3, considérées comme fonctions de $\mathbb{C}$ dans $\mathbb{C}$, telles qu'elles sont présentées dans [CJ96] : il suffit pour cela de ne pas considérer le numéro de feuillet sur lequel se trouve $f(z)$, ce qui revient à considérer l'ensemble des points verts et noirs. Par contre, pour les autres tests, il existe une région de $\mathbb{C}$ pour laquelle des indéterminations se produisent pendant le calcul des fonctions puissances. Les points z appartenant à de telles régions sont caractérisés par le fait que $\mathrm{Log}(z)$ est proche de l'axe réel négatif, et donc le produit $\alpha \mathrm{Log}(z)$ peut ne plus appartenir à U_0 : dans ce cas, le calcul de l'exponentielle d'un tel nombre provoque des indéterminations.

La figure 4.12 présente un agrandissement d'une telle région, pour $\alpha = 3 + i$. Ici, puisque $\mathrm{Arg}(3 + i) > 0$, le produit $\alpha \mathrm{Log}(z)$ peut donner comme résultat un élément de U_1, ce qui provoque une indétermination sur le calcul de $\mathrm{Exp}(\alpha \mathrm{Log}(z))$.

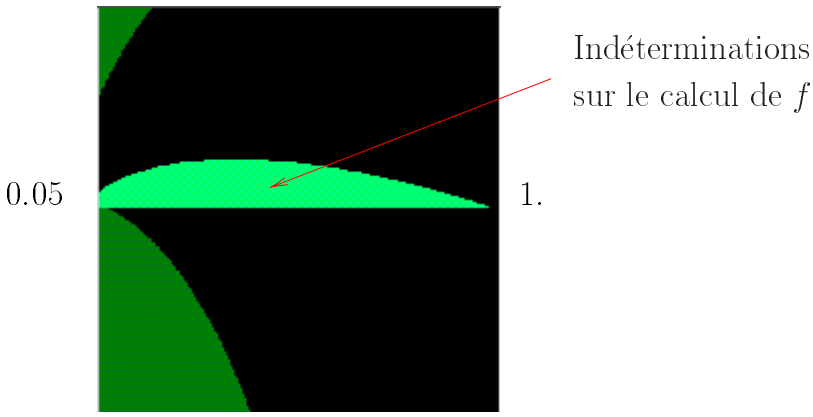


FIG. 4.12: Agrandissement d'une région où se produisent des indéterminations

D'une manière pratique, ces tests permettent de déterminer les régions de $\mathbb{C}$ où le calcul avec des nombres complexes usuels est possible, et celles où il devient impératif de calculer avec les éléments de la surface de Riemann du logarithme : ce sont les zones vertes et rouges. Sur ces zones, il sera impossible ne serait-ce que d'avoir une

```
indet = 0 ;

try {

   val = f ( z ) ;
   // si ce calcul n'a pas provoqué d'indéterminations
   try {

      nval = val.expR () ;

   } catch (Indetermination& E) { //val n'est pas sur U0

      nval = E.Solve ( Indetermination::ARBITRARY ) ;
      indet = 1 ;

   }

   nval = nval.logR () ;

   if ( val == nval ) { // ce point appartient à la clearcut

    if ( indet == 0 )
        couleur = Noir ; // cas déterminé
      else
        couleur = Vert ; // cas non déterminé

   } else { // le point n'appartient pas à la clearcut

    if ( indet == 0 )
        couleur = Bleu ; // cas déterminé
      else
        couleur = Rouge ; // cas non déterminé

   }
} catch (Indetermination& E) { //Indetermination dans le calcul de f

   couleur = INDEFINI ;
}
```

TAB. 4.3: Algorithme d'affectation de couleurs des points du plan complexe

test	figure	f
test 1	4.13	$\mathrm{Exp}(z)$
test 2	4.14	z^{3-2i}
test 3	4.15	z^{3-i}
test 4	4.16	z^{3}
test 5	4.17	z^{3+i}
test 6	4.18	z^{3+2i}

TAB. 4.4: Les tests de "clearcut regions"

détermination continue des fonctions puissances. De plus, la donnée de ces zones peut se révéler précieuse pour l'évaluation rapide de fonctions dans des problèmes plus généraux, tels que des intégrations sur des chemins, car cette carte permet de décider si une fonction doit être calculée naïvement avec des nombres complexes usuels, ou s'il faut utiliser des éléments de surface de Riemann, qui peuvent être délicats à gérer, surtout si des opérations telles que des additions doivent être effectuées.

4.8 Première conclusion

Le principe exposé dans ce chapitre ne résout évidemment pas les indéterminations de tous les problèmes de calculs en nombres complexes. Il offre cependant l'avantage de détecter ces indéterminations, et de laisser l'utilisateur choisir quelle valeur sera la mieux adaptée à son calcul. L'utilisateur peut le faire de manière dynamique, c'est-à-dire pendant l'exécution de son programme, ou même pendant la conception du programme, c'est-à-dire que les indéterminations détectées peuvent être résolues par un traitement spécifique, programmé par l'utilisateur.

De plus, cet environnement de calcul en nombres complexes permet d'agrandir le plan complexe : en effet, le calcul se déroule à partir de la spécification initiale des nombres, et ainsi le résultat sera cohérent avec les données de départ s'il ne s'est pas produit d'indéterminations. Par exemple, la multiplication de deux nombres prend en compte les feuillets de départ des nombres, et propage cette information tout au long du calcul. Cela permet entre autres de passer les tests d'Aslaksen de manière naturelle, c'est-à-dire que dans ce cas le programmeur n'a pas à se soucier de la région de $\mathbb{C}$ dans laquelle se passe le calcul pour avoir des valeurs correctes pour ses évaluations de fonctions puissances.

Enfin, ce principe de calcul permet d'avoir une vision plus globale pour des fonctions dont le calcul des valeurs est généralement effectué de manière locale. En effet, si nous ne considérons pas les nombres complexes comme des éléments de surfaces de Riemann du logarithme, nous sommes obligés de calculer les fonctions puissances par des procédés de type *prolongement analytique*, ce qui peut nuire aux performances de programmes utilisant des nombres complexes.

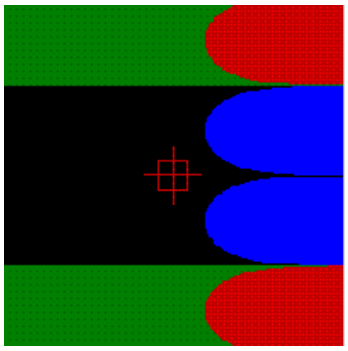


FIG. 4.13: "clearcut region" pour $f(z) = \mathrm{Exp}(z)$

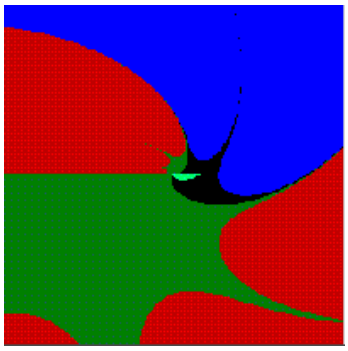


FIG. 4.14: "clearcut region" pour $f(z) = z^{3-2i}$

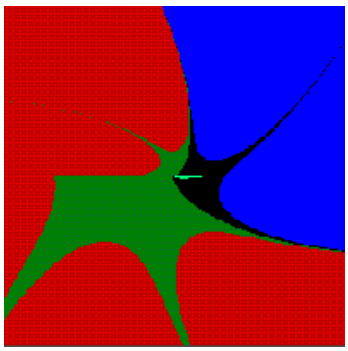


FIG. 4.15: clearcut region pour $f(z) = z^{3-i}$

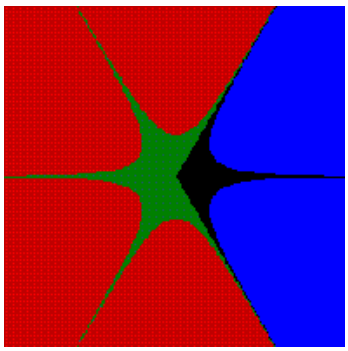


FIG. 4.16: "clearcut region" pour $f(z) = z^3$

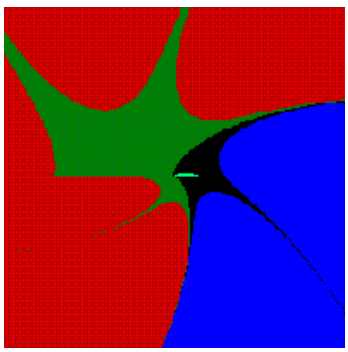


FIG. 4.17: "clearcut region" pour $f(z) = z^{3+i}$

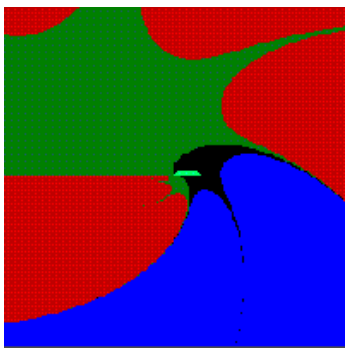


FIG. 4.18: "clearcut region" pour $f(z) = z^{3+2i}$

TAB. 4.5: Les "clearcut regions"

Chapitre 5

Les intégrateurs dans le champ complexe

Un intégrateur dans le champ complexe est un algorithme qui permet de réaliser l'intégration d'une équation différentielle de la variable complexe, c'est-à-dire de donner la valeur de la solution d'une équation différentielle (lorsqu'elle existe) en un point du plan complexe. Nous nous intéresserons ici aux intégrateurs *numériques*, c'est-à-dire à ceux qui renvoient un résultat obtenu par une méthode numérique, par opposition aux intégrateurs de type formel qui eux utilisent des méthodes prenant en compte des séries formelles notamment. De nombreux types d'intégrateurs ont déjà été étudiés dans la littérature, à commencer par le plus simple d'entre eux, utilisant des itérations d'Euler, puis des raffinements de ces méthodes ont été développés, par la considération de méthodes avancées, comme des méthodes à pas variables, voire même avec des itérations de type plus évolués.

Nous nous intéresserons ici non pas aux méthodes numériques en relation avec les équations différentielles, mais plutôt à l'application des notions introduites dans le chapitre 4 concernant les calculs en nombre complexes. Nous étudierons en effet l'influence que peuvent avoir les déterminations principales des fonctions définissant certaines équations différentielles, puis comment la notion d'indétermination introduite au chapitre 4 peut être appliquée aux calculs sur les équations différentielles.

5.1 Méthodes numériques d'intégration dans le plan complexe

Considérons une équation différentielle de la forme

$$y' = f(x, y) \tag{1.1}$$

vérifiant les conditions du théorème de Cauchy (chapitre 2.1.1.1, partie I) sur un ouvert $\Omega = \Omega_1 \times \Omega_2$ de $\mathbb{C}^2$. Considérons de plus une condition initiale $(x_0, y_0) \in \Omega$. Un *intégrateur* est un outil qui calcule la valeur en un point x_f du plan complexe de la solution y de l'équation différentielle (1.1) telle que $y(x_0) = y_0$.

Cas réel

Dans le cas des intégrations avec des variables réelles, un intégrateur doit, d'une manière générale (c'est-à-dire quel que soit la méthode numérique utilisée pour réaliser les calculs), disposer en entrée des données suivantes (cf [Ise96], p.74) :

1. Une équation différentielle à intégrer, c'est-à-dire la donnée de f,
2. Une condition initiale (x_0, y_0),
3. Un point x_f où une valeur de la solution doit être calculée,
4. Une tolérance δ.

L'intégrateur renvoie en sortie une valeur approchée de $y(x_f)$, dans le cas où l'estimation de l'erreur commise est dans les limites définies par la tolérance δ.

Cas complexe

Dans le cas réel, la donnée de x_0 et de x_f est suffisante pour réaliser une intégration *entre* x_0 et x_f. Dans le cas complexe, ceci n'est plus vrai car il n'y a pas qu'un seul chemin pour aller de x_0 à x_f. De plus, la valeur $y_f = y(x_f)$ peut varier en fonction du chemin choisi pour réaliser l'intégration, en particulier si la solution de l'équation différentielle est multiforme. Il faut donc rajouter *a priori* la donnée du chemin d'intégration, $C \subset \Omega_1$, qui a pour origine x_0 et pour extrémité x_f. L'intégration de l'équation (1.1) se fera donc sur le chemin C, c'est-à-dire que les valeurs de x ne seront prises que sur le chemin C. Cela permet, toujours dans le cas où la solution de l'équation différentielle est multiforme, d'avoir une détermination particulière de cette solution.

La méthode d'Euler dans le plan complexe

Nous étudierons dans ce chapitre uniquement la méthode d'Euler explicite dans le plan complexe, sachant que les autres méthodes numériques classiques (de type Runge-Kutta notamment) suivent le même principe.

La méthode d'Euler est la plus simple des méthodes d'intégration numérique. Le principe de cette méthode est, étant donné une condition initiale (x_0, y_0), de trouver la valeur de la solution de manière itérative en x_f.

Étant donnée la condition initiale (x_0, y_0), le couple (x_1, y_1) peut être calculé de la manière suivante :

$$\left\{ \begin{array}{rcl} x_1 & = & x_0 + h_0 \\ y_1 & = & y_0 + h_0 f(x_0, y_0) \end{array} \right.$$

où h_0 est un nombre complexe.

Ainsi, en itérant le procédé, nous pouvons calculer

$$\left\{ \begin{array}{rcl} x_{i+1} & = & x_i + h_i \\ y_{i+1} & = & y_i + h_i f(x_i, y_i) \end{array} \right.$$

Si les différents pas $h_0, h_1, \ldots, h_N$ sont choisis de telle manière que, pour un certain N,

$$x_{N+1} = x_f$$

alors la valeur y_{N+1} représentera une valeur approchée de $y(x_f)$ [Ise96]. En général, le choix des h_i est fait de telle manière que la solution numérique vérifie des conditions préalables de précision [SH96], par le biais du paramètre δ.

La méthode décrite ci-dessus réalise l'intégration de l'équation (1.1) sur un chemin qui dépend fortement du choix des h_i. En général ces nombres complexes ne sont pas choisis arbitrairement. En effet, considérons que nous voulons intégrer l'équation (1.1) sur un chemin C. Alors, si le chemin C est donné sous forme discrète,

$$(\chi_0, \chi_1, \ldots, \chi_n),$$

avec $\chi_n = x_f$, le choix des h_i peut être effectué de deux manières différentes :

- Si nous considérons que la discrétisation du chemin est adaptée à l'intégration de l'équation (1.1), nous pouvons prendre, pour tout i,

  $$h_i = \chi_{i+1} \Leftrightarrow \chi_i$$

  Ainsi, les points issus de la discrétisation du chemin C seront choisis comme points intermédiaires pour la méthode d'Euler. En général, lorsqu'un chemin sous forme discrète est considéré, il n'est pas facile de savoir *a priori* si sa discrétisation est adaptée à l'intégration de l'équation différentielle.
- Nous pouvons aussi considérer que le chemin C est un ensemble d'arêtes, et que l'intégration de l'équation (1.1) doit obligatoirement passer par tous les points de la discrétisation. Ainsi, l'intégration sera effectuée dans un premier temps sur le segment (χ_0, χ_1), avec (χ_0, y_0) comme condition initiale. Notons y^1 la résultat de cette intégration. L'intégration peut être poursuivie sur le segment (χ_1, χ_2) avec comme condition initiale (χ_1, y^1), et continuer ainsi jusqu'au segment (χ_{n-1}, χ_n), pour obtenir une approximation de la solution y au point x_f. Sur chaque segment, le pas peut être soit constant, soit variable.

L'intégration d'une équation différentielle sur un chemin du plan complexe se fait donc en n étapes correspondant aux n segments composant le chemin C discrétisé, et en n_i sous-étapes, correspondant au nombre d'itérations à effectuer pour atteindre le point χ_{i+1} à partir du point χ_i.

5.2 Évaluations de fonctions

Un paramètre important, pour l'intégration numérique d'une équation différentielle de type (1.1), est l'évaluation de la fonction f qui définit l'équation. En effet, si l'intégration est réalisée grâce à une méthode d'Euler explicite à pas constant,

l'intégrateur doit, à chaque fois qu'il calcule la valeur de la solution y en un point intermédiaire, réaliser une évaluation de la fonction f. Ainsi, pendant une intégration par la méthode d'Euler, il y a N_e évaluations de fonctions, N_e étant défini par

$$\sum_{i=0}^{n-1} n_i$$

où n_i a été défini précédemment. L'influence de ces évaluations peut se faire sentir à plusieurs niveaux :

- au niveau de la performance de l'intégrateur. En effet, plus l'évaluation de la fonction est coûteuse en temps de calcul, plus l'intégration sera coûteuse en temps de calcul, et de fait interdira les méthodes d'ordre élevé, qui nécessitent beaucoup d'évaluations de fonctions.
- au niveau de la précision du calcul. En effet, l'erreur commise lors de l'évaluation de la fonction f se transmet directement à la solution de l'équation, et se propage à toutes les valeurs calculées postérieurement.

L'exemple suivant permet de mesurer l'importance du soin à apporter aux évaluations de fonctions, lors de l'intégration numérique d'équations différentielles.

5.3 Exemple

Considérons à titre d'exemple l'équation différentielle suivante de la variable complexe x, pour $c \in \mathbb{C}$, $\alpha \in \mathbb{C}\backslash\{1\}$ et $\beta \in \mathbb{C}\backslash\{\Leftrightarrow 1\}$:

$$y' = cx^{\beta}y^{\alpha} = f(x,y) \tag{3.2}$$

que nous voulons intégrer le long du cercle C centré en 0, à partir de la condition initiale (x_0, y_0), pour $x_0 \in C$. Bien que la fonction f n'est en général pas holomorphe en 0, elle est holomorphe en tout point de $\mathbb{C}^{*\,2}$, et donc nous pouvons intégrer l'équation (3.2) sur C, puisque C ne passe pas par 0.

Afin d'effectuer une intégration par la méthode d'Euler, il faudra évaluer à chaque itération une expression de la forme :

$$y_i + c.h_i x_i^{\beta} y_i^{\alpha}.$$

Cette expression pose plusieurs problèmes au niveau du calcul, à cause de la présence des termes en $x^{\beta}y^{\alpha}$ notamment. L'étude du chapitre 4 montre que des expressions de ce genre peuvent dans certains cas être calculées efficacement, de deux manières possibles :

- En réalisant le prolongement analytique de la fonction de deux variables $x^{\beta}y^{\alpha}$, le long d'un chemin partant de (x_{i-1}, y_{i-1}) allant jusqu'à (x_i, y_i). En plus de la dépendance au chemin et de la difficulté algorithmique de spécification automatique d'un chemin, cette méthode présente l'inconvénient d'être lourde à

mettre en oeuvre, au niveau de la performance en temps de calcul notamment. En effet, ce travail devra être fait à chaque fois qu'une évaluation de fonctions sera nécessaire, c'est-à-dire N_e fois, ce qui peut être pénalisant pour l'efficacité de la méthode.

- Les valeurs de la fonction $x^\beta y^\alpha$ peuvent aussi être calculées de manière globale sur la surface de Riemann du logarithme, après avoir fixé les numéros de feuillet des points x_0 et y_0. Le principal inconvénient est de laisser subsister une indétermination lors de certaines opérations. Dans certains cas, ces indéterminations peuvent être résolues, à condition que les valeurs de x et de y calculées soient cohérentes, par le calcul de leur numéro de feuillet.

Une condition nécessaire pour avoir un calcul correct de la solution de cette équation est bien d'avoir une détermination continue de la fonction $x^\beta y^\alpha$. En effet, supposons que la fonction ci-dessus est calculée avec des nombres complexes usuels, avec les formules

$$\begin{aligned} y^\alpha &= e^{\alpha \log(y)} \\ x^\beta &= e^{\beta \log(x)} \end{aligned}$$

Si les arguments de x et de y sont considérés dans l'intervalle $]\Leftrightarrow\pi, \pi]$, les formules ci-dessus introduisent des problèmes de continuité, notamment lorsque l'argument de x varie. Si en calculant x_i et y_i lors de l'itération numéro i une erreur de détermination est commise, cette erreur va se reporter sur l'itération suivante et par suite sur la totalité des valeurs calculées postérieurement. Cela peut donner une situation comme celle décrite sur les figures 5.1c et 5.1d.

La figure 5.1c a été obtenue en intégrant l'équation (3.2) avec une méthode d'Euler sur le plan complexe, avec comme condition initiale le couple $(1, 1)$, et avec les paramètres $c = 5$, $\alpha = \beta = \frac{1}{2}$. La figure 5.1c montre que les problèmes de détermination continue de la fonction f ont une influence importante sur la solution calculée par la méthode d'Euler : en effet, celle-ci n'est pas dérivable pour certaines valeurs de x, et donc non-holomorphe en ces points. Le théorème de Cauchy-Lipshitz n'est plus vérifié.

La figure 5.1d a été obtenue en traçant la solution (calculée dans le chapitre 5.7) de l'équation (3.2) le long du cercle C. Cette fonction étant multiforme, le principe décrit dans le chapitre 4 a été appliqué pour tracer cette figure. Cette courbe coincide bien jusqu'à ce que $\mathrm{Arg}(y) = \pi$ avec la courbe 5.1c, et à ce niveau la solution calculée avec les nombres complexes usuels présente une discontinuité sur sa dérivée. En effet, la dérivée en ce point a été obtenue par évaluation de la fonction f en un point précédent, et la mauvaise évaluation de f a provoqué l'apparition de cette discontinuité. Les images du cercle C par la fonction $f(x, y_0)$, soit calculée avec des nombres complexes usuels, soit en tenant compte des numéros de feuillets, permettent de vérifier ce point, comme montré sur les figures 5.1a et 5.1b.

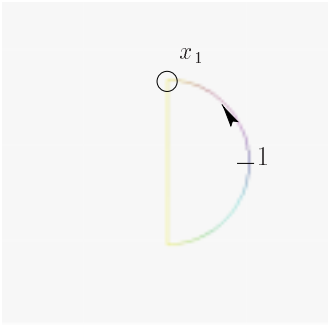


fig. 5.1a

Évaluation de la fonction f sur le cercle centré en 0, de rayon 1, avec les nombres complexes usuels

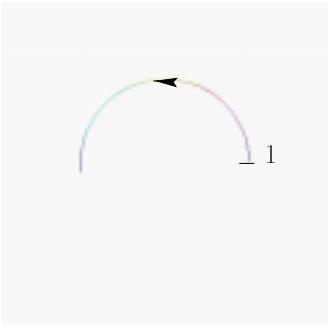


fig. 5.1b

Évaluation de la fonction f sur le cercle centré en 0, de rayon 1, avec les éléments de $\mathcal{M}_{\log}$

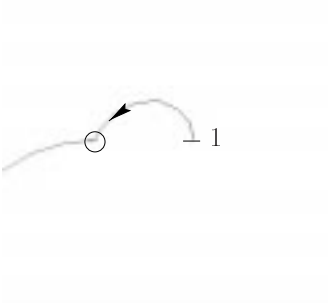


fig. 5.1c

Solution de l'équation différentielle obtenue en évaluant la fonction f avec les nombres complexes usuels

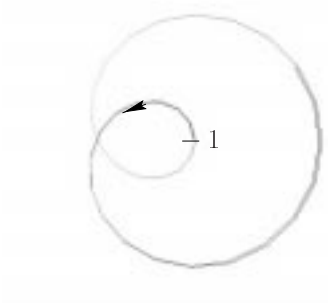


fig. 5.1d

"Vraie solution", calculée à la main et évaluée avec les éléments de $\mathcal{M}_{\log}$

FIG. 5.1: Influence de l'évaluation des fonctions pour l'intégration

Le calcul avec des nombres complexes usuels ne donne pas de détermination continue de la fonction $f(x, y_0)$, notamment au point x_1, ce qui pose des problèmes de discontinuités de la dérivée de la solution calculée. Par contre, la figure 5.1 permet de vérifier que la détermination de la fonction $f(x, y_0)$ est bien continue sur le cercle C, en particulier au point x_1, dès lors que les numéros de feuillet des variables sont pris en compte. Le prochain chapitre montre comment adapter une méthode numérique d'intégration (la méthode d'Euler) à ce cas de figure.

5.4 La méthode d'Euler modifiée pour la prise en compte des numéros de feuillet

Nous allons supposer que la fonction f possède une seule singularité, en 0. Elle peut être éventuellement multiforme, mais nous allons supposer que *nous sommes capables de donner une détermination continue de la fonction f de manière globale*

sur tout chemin de $\mathbb{C}^2$ ne passant pas par 0 . Le but de ce chapitre n'est pas d'intégrer de manière générale une équation différentielle sur une surface de Riemann, bien que dans [For81] il est montré *de manière théorique* que l'on peut intégrer une équation différentielle sur un revêtement universel[1]. Les méthodes de calcul exposées en 4 seront appliquées au problème de l'intégration numérique des équations différentielles.

Considérons C, un chemin de $\mathbb{C}^*$, et (x_0, y_0) une condition initiale telle que $x_0 \in C$. Nous supposerons par la suite que le chemin C est paramétré par une fonction Φ, et que de plus nous avons une discrétisation de C par l'intermédiaire des points

$$(\chi_0, \chi_1, \ldots, \chi_n)$$

Soit enfin $\Omega = \Omega_1 \times \Omega_2 \subset \mathbb{C}^2$ un ouvert dans lequel f est holomorphe, et tel que $C \subset \Omega_1$.

La méthode d'Euler peut être appliquée à partir de (x_0, y_0), sur chaque segment (χ_j, χ_{j+1}) pour $0 \leq j < n$, avec comme condition initiale pour ces différentes intégrations le couple (χ_j, y_j) où y_j est la valeur calculée lors de l'intégration précédente si $j > 0$, ou y_0 sinon. A chaque itération, les valeurs calculées de manière intermédiaires seront notées $x_1, x_2, \ldots, x_{n_j}$ et $y_1, y_2, \ldots, y_{n_j}$:

$$x_{i+1} = x_i + h_i \tag{4.3}$$
$$y_{i+1} = y_i + h_i f(x_i, y_i) \tag{4.4}$$

Lorsque les nombres x_i, y_i, x_{i+1} et y_{i+1} sont affectés de leur numéros de feuillet, des indéterminations de plusieurs types peuvent se produire, dans le calcul de f, mais aussi lorsque les additions de l'équation (4.3) et celles de l'équation (4.4) sont effectuées.

Calcul de f

Puisque nous avons supposé que le calcul d'une détermination continue globale de f sur Ω était possible, il est clair que le calcul de f en (x_i, y_i) ne pose aucun problème de détermination à partir du moment où les deux nombres x_i et y_i sont bien déterminés, *ce que nous supposerons être le cas pour l'indice* i.

Indéterminations : premier cas

Par contre, les additions dans les deux formules ci-dessus peuvent soulever des indéterminations : cependant, dans le cas particulier de l'équation (4.3), il n'est pas difficile de résoudre l'indétermination qui a pu être éventuellement soulevée. En effet, si cette équation est réécrite de la manière suivante

$$x_{i+1} \Leftrightarrow x_i = h_i$$

[1]de plus, cette démonstration ne concerne que des équations différentielles linéaires, bien qu'elle s'applique aussi à des équations différentielles non linéaires en y introduisant le théorème de Cauchy-Lipschitz.

alors le cas où l'addition présente dans l'équation (4.3) est indéterminée correspond au cas où x_{i+1} et x_i ne sont pas sur le même feuillet. Nous avons supposé que le chemin C est paramétré par une fonction Φ :

$$\begin{array}{rcccc} \Phi & : & [0,1] & \leftrightarrows & \mathbb{C} \\ & & t & \leftrightarrows & \Phi(t) \end{array}$$

Notons tout de suite que $\forall t \in [0,1]$, $\Phi(t) \neq 0$. Alors

$$\mathrm{Arg}(\Phi(t)) = \frac{1}{i}[\mathrm{Log}(\Phi(t)) \leftrightarrows \ln(|\Phi(t)|)]$$

La fonction Log ci-dessus représente une détermination continue de la fonction logarithme sur le chemin C, alors que la fonction ln représente la fonction logarithme réel. Puisque, pour tout $t \in [0,1], |\Phi(t)| > 0$, et que cette fonction est continue, la fonction

$$\begin{array}{rcccc} \ln(|\Phi|) & : & [0,1] & \leftrightarrows & \mathbb{R} \\ & & t & \leftrightarrows & \ln(|\Phi(t)|) \end{array}$$

est continue sur l'intervalle $[0,1]$.

De plus, la fonction logarithme définie au chapitre 4 est continue sur le chemin C, donc la fonction

$$\begin{array}{rcccc} \mathrm{Log}(\Phi) & : & [0,1] & \leftrightarrows & \mathcal{M}_{\log} \\ & & t & \leftrightarrows & \mathrm{Log}(\Phi(t)) \end{array}$$

est elle aussi continue, et par conséquent la fonction $\mathrm{Arg}(\Phi(t))$ est continue sur le segment $[0,1]$. En conséquence,

$$\exists n \in \mathbb{N},\ \forall j \in [0,n[,\ |\mathrm{Arg}(\chi_{j+1}) \leftrightarrows \mathrm{Arg}(\chi_j)| < \pi \tag{4.5}$$

Cela signifie que, puisque le chemin C considéré est continu, il peut être choisi de telle manière que deux points consécutifs de la discrétisation de ce chemin soient, soit sur le même feuillet, soit sur des feuillets adjacents, mais d'écart angulaire inférieur à π. Il s'ensuit que l'opération

$$\chi_{j+1} \leftrightarrows \chi_j$$

pourra toujours être correctement déterminée pour des valeurs de n suffisamment importantes. Donc, si les points x_{i+1} et x_i sont situés sur le segment (χ_j, χ_{j+1}), ou sur l'arc de la courbe C défini par ces deux points(cela dépendra du choix de h_i), l'indétermination éventuellement présente dans l'évaluation de (4.3) peut être résolue par le choix d'un bon n et d'un bon h_i.

Indéterminations : deuxième cas

La deuxième possibilité d'indéterminations se trouve dans la formule (4.4), elle est de plus beaucoup plus délicate à résoudre, bien que le principe soit le même que

dans le cas précédent : au lieu de garantir la continuité de $x(t)$, nous allons garantir l'holomorphie de $y(x)$ sur C.

La fonction $y(x)$ est, d'après le théorème de Cauchy-Lipshitz, holomorphe sur tout ouvert inclus dans $\mathbb{C}^*$, et plus précisément sur le chemin C. Si une indétermination se produit pendant l'itération i, c'est-à-dire si y_i et $h_i f(x_i, y_i)$ ont un écart angulaire supérieur à π, alors il faut déterminer un numéro de feuillet cohérent qui doit être affecté à y_{i+1}. Posons $y_i = (p_i, k_i)$, $y_{i+1} = (p_{i+1}, k_{i+1})$ et $h_i f(x_i, y_i) = (P_i, K_i)$. Dans ce cas, la relation suivante

$$p_{i+1} = p_i + P_i$$

s'applique aux parties complexes des nombres y_i, y_{i+1} et $h_i f(x_i, y_i)$. Ce nombre complexe est parfaitement déterminé par la relation ci-dessus, de telle manière que la projection de la fonction $y(x)$ sur $\mathbb{C}$ soit holomorphe.

La détermination de k_{i+1} peut se faire par holomorphie de la fonction $y(x)$. En effet, l'indétermination de y_{i+1} tient à l'impossibilité de déterminer son numéro de feuillet de manière arbitraire, sans connaissances complémentaires sur $y(x)$. Ici, nous possédons une telle connaissance sur y, qui est celle de la continuité de son argument. En effet, la relation

$$\mathrm{Arg}(y(x)) = \frac{1}{i}[\mathrm{Log}(y(x)) \Leftrightarrow \ln(|y(x)|)]$$

s'applique encore à $y(x)$, et puisque $\forall x \in C, y(x) \neq 0$, alors la fonction ci-dessus est continue.

De plus, si nous écrivons

$$\begin{aligned} \mathrm{Arg}(y_{i+1}) &= \arg(p_{i+1}) + 2k_{i+1}\pi \\ \mathrm{Arg}(y_i) &= \arg(p_i) + 2k_i\pi \end{aligned}$$

alors, pour tout $k \in \mathbb{Z}$,

$$\begin{aligned} |\mathrm{Arg}(y_{i+1}) \Leftrightarrow \mathrm{Arg}(y_i)| &= |\arg(p_{i+1}) \Leftrightarrow \arg(p_i) + 2(k_{i+1} \Leftrightarrow k_i)\pi| \\ &= |\arg(p_{i+1}) \Leftrightarrow \arg(p_i) + 2k\pi + 2(k_{i+1} \Leftrightarrow (k_i \Leftrightarrow k))\pi| \\ &\leq |\arg(p_{i+1}) \Leftrightarrow \arg(p_i) + 2k\pi| + 2|k_{i+1} \Leftrightarrow (k_i \Leftrightarrow k)|\pi \end{aligned}$$

Ainsi, afin de résoudre l'indétermination qui a pu éventuellement se produire pendant le calcul de la formule (4.4), nous sommes confrontés à trois possibilités. Soit δ_a un nombre réel, représentant une tolérance angulaire, qui peut être pris aussi petit que l'on veut, et examinons les trois cas :

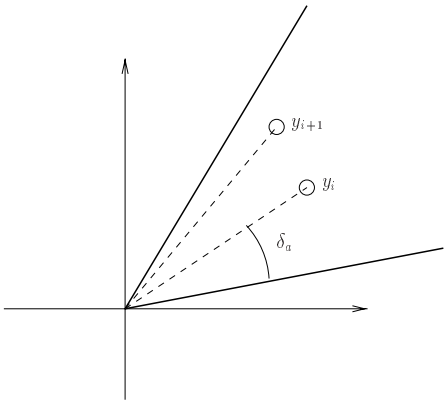


FIG. 5.2: Premier cas

premier cas : $|\arg(p_{i+1}) \Leftrightarrow \arg(p_i)| < \delta_a \Leftrightarrow k = 0$

Ce cas (illustré sur la figure 5.2) est le plus simple, car les deux nombres y_i et y_{i+1} sont sur le même feuillet. Il suffira alors de choisir $k_{i+1} = k_i$ pour satisfaire la continuité de la fonction $\mathrm{Arg}(y(x))$.

deuxième cas : $|\arg(p_{i+1}) \Leftrightarrow \arg(p_i)| > \delta_a$ et $\exists \epsilon \in \{\Leftrightarrow 1, 1\}, |\arg(y_{i+1}) \Leftrightarrow \arg(y_i) + 2\epsilon\pi| < \delta_a$

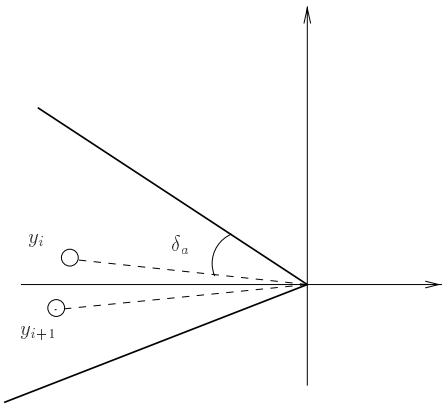


FIG. 5.3: Second cas

La figure 5.3 montre que $\arg(p_i)$ est proche de π, alors que $\arg(p_{i+1})$ est proche de $\Leftrightarrow\pi$. Ce cas est celui où la continuité de l'argument *principal* de $y(x)$ ne peut être satisfaite, mais où la continuité de $\mathrm{Arg}(y(x))$ peut l'être, à δ_a près. Dans ce cas, il suffit de choisir $k_{i+1} = k_i \Leftrightarrow \epsilon$. En effet, cela correspond au cas où il y a changement de feuillet pour les valeurs de la fonction $y(x)$. Le fait de choisir un tel ϵ revient à assurer la continuité de $\mathrm{Arg}(y(x))$ à la tolérance δ_a près.

dernier cas : $\forall k \in \mathbb{Z}, |\mathrm{Arg}(y_{i+1}) \Leftrightarrow \mathrm{Arg}(y_i) + 2k\pi| > \delta_a$

Le dernier cas (illustré par la figure 5.4) est celui où, pour le δ_a donné, la continuité de la fonction $\mathrm{Arg}(y(x))$ ne peut pas être assurée : dans ce cas, il est

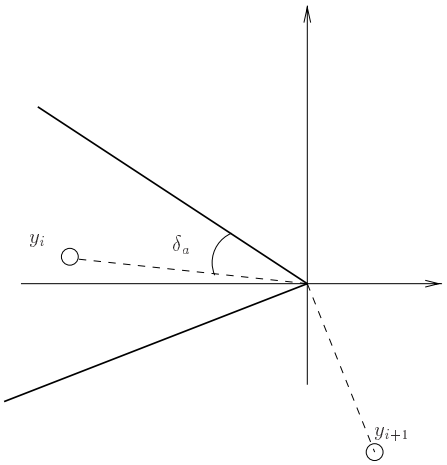


FIG. 5.4: Dernier cas

impossible de décider *a priori* quelle est la valeur de k_{i+1} qui donne les meilleurs résultats pour la continuité de la fonction. Ne pouvant pas le déterminer avec de tels écarts angulaires, nous pouvons essayer de refaire une intégration entre x_i et x_{i+1}, mais avec un pas plus petit afin de limiter les variations (en argument) des nombres considérés.

En conclusion, avec les hypothèses posées en début de chapitre, la continuité de la solution y peut être dans certains cas assurée, même lorsque des indéterminations se produisent pendant les calculs.

5.5 Algorithme d'intégration

Nous pouvons maintenant proposer l'algorithme 5.5.0.1 d'intégration par la méthode d'Euler, adapté à la prise en compte des numéros de feuillet : cet algorithme est une implémentation directe des méthodes numériques d'intégration classiques, les particularités se trouvent dans les fonctions *GetNextX, GetNextY* détaillées respectivement page 99 et page 100.

Comme exposé précédemment, la fonction *GetNextX* doit retourner x_{i+1} à partir de x_i, et cette nouvelle valeur doit être cohérente (en particulier au niveau des numéros de feuillet) avec les valeurs précédentes de x_i. Cette méthode peut être de deux natures différentes : si le chemin est donné sous forme paramétrée, avec prise en compte des numéros de feuillets (c'est-à-dire que la fonction de paramétrisation est définie de $[0,1]$ dans $\mathcal{M}_{\log}$), cette méthode revient à une simple discrétisation temporelle du segment $[t_j, t_{j+1}]$, alors que si le chemin est juste donné sous forme de points dans le plan complexe, il faut que les x intermédiaires soient bien cohérents au niveau des numéros de feuillets.

Si nous disposons de la fonction Φ qui paramétrise le chemin C, ces points peuvent être calculés directement : le moyen le plus efficace de calculer cette valeur est d'effectuer une discrétisation en temps du segment $[t_j, t_{j+1}]$:

5.5.0.1 Algorithme: Intégration entre χ_j et χ_{j+1}

```
i = 0 ;
x_0 = χ_j ;
y_0 = y_j ;
pas_fini = 1 ;

while (pas_fini == 1) {

        x_{i+1} = GetNextX ( x_i ) ;
        h_i = x_{i+1} ⇔ x_i ;
        y_{i+1} = GetNextY ( x_i, y_i, x_{i+1}, h_i ) ;

        if ( x_{i+1} == χ_{j+1} )
                pas_fini = 0 ;
}

return  y_{i+1} ;
```

$$(t_j^0, t_j^1, \ldots, t_j^{n_j})$$

telle que $t_j^0 = t_j$ et $t_j^{n_j} = t_{j+1}$, et considérer la discrétisation de l'arc (χ_j, χ_{j+1})

$$(\Phi(t_j^0), \Phi(t_j^1), \ldots, \Phi(t_j^{n_j}))$$

Le paramètre n_j peut être choisi de telle manière que cette discrétisation satisfasse les conditions énoncées en (4.5) sur les éléments de cette discrétisation : cela évitera les indéterminations lors de cette étape.

Si par contre le chemin C n'est pas paramétré par une fonction Φ (c'est-à-dire qu'il a été spécifié point par point), alors il faudra faire attention à ce que les points intermédiaires forment une courbe continue (notamment en argument), ce qui ne pourra se faire que si un grand soin est apporté au traitement des problèmes de coupures selon l'axe réel négatif : cela peut se faire par des considérations de continuité, comme montré dans l'algorithme 5.5.0.2.

La fonction *GetNextY* est la plus importante (algorithme 5.5.0.3, page 100) : elle calcule les valeurs de y. Cet algorithme est de type récursif, et les appels récursifs sont générés lorsque la précision angulaire n'est pas suffisante, et qu'il faut effectuer une nouvelle intégration entre deux points intermédiaires, x_i et x_{i+1}, avec une valeur initiale précédemment calculée, y_i.

5.5.0.2 Algorithme: GetNextX : cas non paramétré

```
try {
        x_{i+1} = x_i + (χ_{j+1} − χ_j) / n_j ;
} catch (PlusIndetermination& Ex) {
        x_{i+1} = Ex.Solve ( Indetermination::ARBITRARY ) ;
        if ( Arg(x_i) ⇔Arg(x_{i−1}) > 0 )
                k_{i+1} = k_i + 1 ;
        else
                k_{i+1} = k_i ⇔1 ;
}
return x_{i+1} ;
```

L'algorithme 5.5.0.3 est constitué essentiellement de deux blocs **try/catch**. Le premier bloc revient à effectuer l'itération d'Euler, le second permet d'examiner le résultat précédemment trouvé et de décider de sa validité, et notamment de sa cohérence au niveau des numéros de feuillets.

Le second bloc essaie plusieurs valeurs possibles pour le numéro de feuillet de y_{i+1}, lorsque celui-ci n'est pas cohérent avec les valeurs précédemment calculées. Pour cela, les deux arguments consécutifs sont comparés, et si le résultat de la comparaison est supérieur à la tolérance angulaire, plusieurs valeurs sont essayées : dans un premier temps, la valeur du numéro de feuillet précédemment trouvée est augmentée de 1, puis si cela ne convient pas, diminuée de 1. Si un de ces différents essais est concluant, le résultat est renvoyé, sinon une exception est lancée. Cela correspond au cas où les arguments de deux nombres consécutifs sont d'écart trop important, et donc il faut effectuer une nouvelle intégration plus précise afin de pouvoir décider de la validité des nombres calculés. Ce traitement est effectué dans le dernier bloc **catch**, où l'intégration est relancée entre x_i et x_{i+1}, avec comme condition initiale y_i.

Les critères d'arrêt de l'algorithme peuvent être de deux natures différentes :

- $|x_{i+1} \Leftrightarrow x_i| < h_m$, où h_m est un réel fixé en fonction de la précision machine. Dans ce cas, les erreurs numériques dues aux calculs ne sont plus négligeables devant les valeurs trouvées.
- le niveau de récursivité est trop important : cela se produit lorsque la fonction $f(x, y)$a des variations en arguments trop importantes par rapport à celles de

5.5.0.3 Algorithme: GetNextY

```
try {
        y_{i+1} = y_i + h_i f(x_i, y_i) ;
} catch (PlusIndetermination& Ex) {
        y_{i+1} = Ex.Solve ( Indetermination::ARBITRARY ) ;
}
try {
      if ( |arg(y_{i+1}) ⇔arg(y_i)| > δ_a ) {
            if ( arg(y_i) ⇔arg(y_{i-1}) > 0 ) {
                  y_{i+1} = y_{i+1}.e^{2iπ} ;
                  if ( |Arg(y_{i+1}) ⇔Arg(y_i)| < δ_a )
                        return y_{i+1} ;
                  else
                        throw Euler_Exception () ;
            } else {
            y_{i+1} = y_{i+1}.e^{-2iπ} ;
            if ( |Arg(y_{i+1}) ⇔Arg(y_i)| < δ_a )
                  return y_{i+1} ;
            else
                  throw Euler_Exception () ;
            }
      }
} catch (Euler_Exception& EE) {
        y_{i+1} = Integrate ( x_i, x_{i+1}, y_i ) ;
}
return y_{i+1} ;
```

$y(x)$. Pour éviter une saturation de la machine, il peut être préférable d'arrêter le calcul.

Nous allons voir dans le chapitre suivant comment ceci a été implémenté en C++, pour ensuite passer à des tests.

5.6 Implémentation

La classe `Riemann_Euler`

Les objets intégrateurs définis précédemment sont des objets dérivés d'une classe, la classe `Riemann_Euler`. Cette classe `Riemann_Euler` est une classe abstraite (virtuelle pure), c'est-à-dire qu'il est impossible de construire un objet de cette classe.

Pour construire un objet réalisant l'intégration d'une équation de type

$$y' = F(x, y)$$

entre deux points du plan complexe avec une condition initiale, il faut passer par la définition d'une autre classe, par exemple la classe `Mon_Integrateur`, qui doit respecter les points suivants :

- la classe `Mon_Integrateur` doit dériver de la classe `Riemann_Euler`. En effet, la classe générique `Riemann_Euler` définit l'algorithme général d'intégration défini précédemment, et cet algorithme sera basé sur des classes implémentées dans la classe `Mon_Integrateur`.
- elle doit surcharger les fonctions virtuelles suivantes :
  - la méthode `Mon_Integrateur::F`, qui permet de calculer des évaluations de $F(x, y)$, pour x et y des nombres complexes affectés de leurs numéros de feuillet :

    ```
    Riemann Mon_Integrateur::F ( const Riemann x ,
                                 const Riemann y ) ;
    ```

  - les pseudo-constructeurs et pseudo-destructeurs virtuels utilisés pour construire les objets en cas de récursivité :

    ```
    Riemann_Euler
         *Mon_Integrateur::NewInteg ( const Riemann& x0 ,
                                      const  Riemann& x1,
                                      const  Riemann& y0,
                                      int n , double d) ;

    void Mon_Integrateur::Destroy ( Riemann_Euler *RE ) ;
    ```

Les classes d'exceptions

La classe `Euler_Exception` est utilisée uniquement comme classe d'exceptions, c'est à dire que les objets de cette classe sont construits uniquement pour interrompre l'intégration, et pour cette classe particulière, relancer une intégration plus précise.

Deux autres classes ont été définies, afin d'interrompre de manière définitive l'intégration :

- la classe `Little_Step`, dont un objet est construit lorsque le pas d'intégration est trop petit relativement à la tolérance machine.
- la classe `Big_Level`, dont un objet est construit lorsque le niveau de récursivité est trop important.

5.7 Exemples

Nous allons intégrer numériquement l'équation (3.2), pour différentes valeurs des paramètres c, α, β.

5.7.1 Résolution exacte

La solution y de l'équation différentielle (3.2) vérifiant $y(x_0) = y_0$ peut être calculée directement. En effet, l'équation (3.2) peut se transformer en

$$\frac{dy}{y^\alpha} = cx^\beta dx$$

d'où

$$\frac{1}{1 \Leftrightarrow \alpha}(y^{1-\alpha} \Leftrightarrow y_0^{1-\alpha}) = \frac{c}{1+\beta}(x^{1+\beta} \Leftrightarrow x_0^{1+\beta})$$

et ainsi nous obtenons la solution de l'équation (3.2) vérifiant la condition initiale (x_0, y_0) :

$$y(x) = [\frac{c(1 \Leftrightarrow \alpha)}{1+\beta}(x^{1+\beta} \Leftrightarrow x_0^{1+\beta}) + y_0^{1-\alpha}]^{\frac{1}{1-\alpha}} \tag{7.6}$$

Cette fonction est multiforme, et de plus présente deux singularités : la première est située en 0 et provient du terme $x^{1+\beta}$, la seconde (singularité mobile) est située au point

$$\sigma(x_0, y_0) = [x_0^{1+\beta} \Leftrightarrow \frac{1+\beta}{c(1 \Leftrightarrow \alpha)} y_0^{1-\alpha}]^{\frac{1}{1+\beta}}$$

qui provient du terme mis à la puissance $\frac{1}{1-\alpha}$ dans l'expression (7.6). La singularité présente en 0 est exclue, car la fonction $cx^\beta y^\alpha$ n'est pas holomorphe en 0. Ceci dit, la forme de cette expression nous oblige à faire des considérations de détermination pour pouvoir calculer la valeur de cette fonction en un point x quelconque dans le plan complexe.

5.7.2 Résolution numérique

Pour résoudre numériquement cette équation, il faut d'abord programmer un intégrateur, qui contiendra la définition de la fonction f. Considérons alors la définition de la classe `Pow_Euler` :

```
class Pow_Euler : public Riemann_Euler {

  public :
  double coeff ;
  double alpha , beta ;
  static int level ;

  // constructors
  Pow_Euler () ;
  Pow_Euler ( const Riemann& x0,const  Riemann& x1 ,
              const Riemann& y0 , int n , double a ,
              double b, double d ) ;
  ~Pow_Euler () ;


  //specific functions
  Riemann_Euler *NewInteg ( const Riemann& x0 ,
                            const Riemann& x1, const  Riemann& y,
                            int n , double d) const ;
  void Destroy ( Riemann_Euler *RE ) const ;

  Riemann Sol ( const Riemann x, const Riemann x0 ,
                const Riemann yi ) const ;
  Riemann F ( const Riemann x ,const  Riemann y ) const ;

  void Identify ( ostream& os ) const {
    os << "Pow_Euler created \n"
      << "xd = " << xd << ", xf = " << xf
      <<  ", y0 = " << y0 << "\n"
      << "N = " << N << " coeff = " << coeff
      << "level = " << level << "\n" ;
  }


};
```

Le constructeur des objets de la classe `Pow_Euler` prend en entrée les bornes de l'intervalle d'intégration, x_0 et x_1, la valeur initiale de y en x_0, y_0, puis différents paramètres requis pour l'intégration : n représente le nombre de sous-étapes qui seront effectuées sur le segment $[x_0, x_1]$, puis a et b qui représentent les nombres α et β vus

précédemment, puis enfin la tolérance angulaire, d. Puisque cette classe dérive de la classe abstraite `Riemann_Euler`, elle bénéficie de la méthode `Integrate ()`, ainsi que de toutes les méthodes de tracé des résultats définies par défaut dans la classe générique.

La méthode de tracé par défaut utilise celle de la classe `Riemann`, c'est-à-dire que pour tout point de coordonnée complexe z, de numéro de feuillet k, le point de l'espace défini par

$$\begin{bmatrix} \Re(z) \\ \Im(z) \\ \mathrm{Arg}(z) \end{bmatrix}$$

où $\mathrm{Arg}(z) = \arg(z) + 2k\pi$. De plus, le formalisme de couleur consistant à affecter à chaque point une couleur correspondant à l'argument de la valeur de la variable d'intégration correspondant à ce point a été adopté.

5.7.3 Cohérence des numéros de feuillet des valeurs calculées

La planche 5.1 montre plusieurs exemples développés pour illustrer l'usage de l'algorithme exposé en 5.4. Cette planche montre l'usage des numéros de feuillet afin de garantir l'holomorphie de la solution calculée. Aucune des intégrations effectuées pour obtenir les figures présentes sur cette planche n'ont effectué des appels récursifs. Les trois premières figures de cette planche (5.5, 5.6, 5.7) reprennent l'exemple exposé précédemment.

Figure 5.5 : L'intégration complète (c'est-à-dire sur la totalité du cercle d'intégration, le cercle centré en 0 de rayon 1) de l'équation (3.2) est présentée sur cette figure. Les paramètres de l'équation sont $\alpha = \beta = \frac{1}{2}$. La condition initiale a pour valeur $(1, 1)$. La courbe de la solution calculée avec les nombres complexes usuels possède quatre points anguleux.

Figures 5.6 et 5.7 : La première figure montre l'intégration complète sur le cercle en utilisant l'algorithme 5.5.0.1, avec les mêmes paramètres que précédemment, la valeur de la tolérance angulaire étant fixée à 0.5. La seconde figure superpose les deux courbes 5.6 et 5.5. Aucun appel récursif ne s'est produit durant le calcul, ce qui semble indiquer que la fonction définissant l'équation différentielle (3.2) n'a pas eu de variations importantes de son argument pendant le déroulement de l'intégration. La solution dessinée est bien lisse, et correspond à la vraie solution de la même équation (sur la figure 5.1d).

Les figures 5.9 et 5.10 reprennent l'intégration de cette même équation, mais en faisant varier les conditions initiales.

FIG. 5.5: Solution obtenue avec les nombres complexes usuels

FIG. 5.6: Cohérence des numéros de feuillet

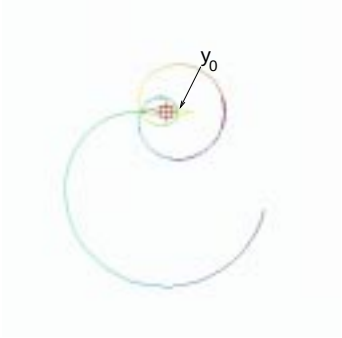


FIG. 5.7: Superposition des deux courbes

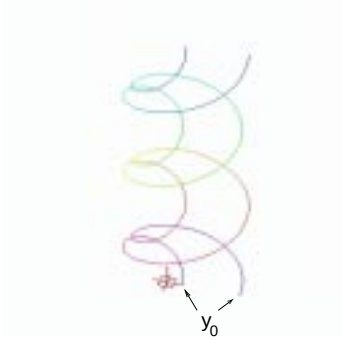


FIG. 5.8: Cas $y_0 > 0$

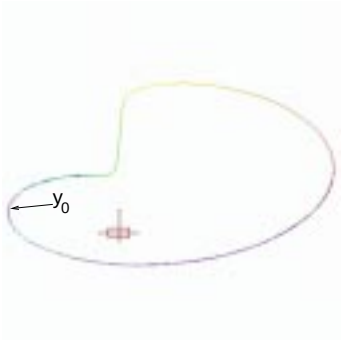


FIG. 5.9: Cas $y_0 = \Leftrightarrow 1$

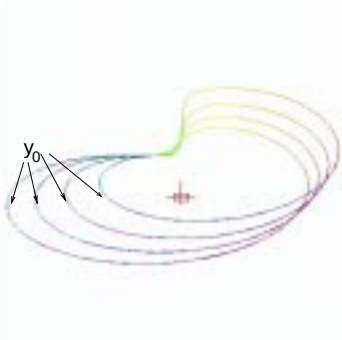


FIG. 5.10: Cas $y_0 < 0$

TAB. 5.1: Résultats de l'intégrateur 5.5.0.1

Figure 5.9 et 5.10 : La figure 5.9 montre l'intégration de l'équation à partir de la condition initiale $(1, \Leftrightarrow 1)$. Aucun appel récursif ne s'est produit durant l'intégration, et de plus les valeurs de la solution ont toujours un numéro de feuillet qui reste cohérent. Cela peut être lu sur la figure, par la continuité des arguments de la solution. Le passage délicat pendant cette intégration se présente lorsque la courbe s'approche de 0, où l'argument principal de la solution passe brusquement de π à $\Leftrightarrow\pi$. La discrétisation du cercle d'intégration ($N = 50$) suffit dans ce cas pour passer les tests de tolérance angulaire et donc les critères de continuités de l'argument de la solution sont vérifiés. La figure 5.10 montre d'autres intégrations, pour des valeurs de y_0 allant de $\Leftrightarrow 1$ à $\Leftrightarrow 4$. Les différentes solutions ainsi calculées reviennent toutes à leur point de départ, ce qui n'est pas le cas pour l'exemple suivant.

Figure 5.8 : Cette figure reprend la courbe présentée en 5.6, mais cette fois dépliée dans l'espace, en utilisant la méthode de visualisation par défaut de la classe `Riemann`. La seconde courbe tracée correspond à une autre condition initiale, $(1, 4)$, et les deux solutions ont le même comportement. Les courbes ne sont pas fermées, bien qu'en projection sur $\mathbb{C}$, les valeurs des extrémités de la courbe sont égales. Cette différence tient au fait que pour avoir des résultats corrects pour les déterminations de la fonction à évaluer, les valeurs de la solution ont été affectées de numéros de feuillet allant de 0 à 2, et donc les courbes ne sont pas fermées sur la figure. Cette figure est intéressante, car elle pourrait motiver une étude poussée de la solution permettant de considérer les numéros de feuillet de ses valeurs *modulo* 2 : en effet, la solution exacte de l'équation différentielle, bien que nous la considérions sur la surface de Riemann du logarithme, est en fait bien définie sur une surface compacte de genre topologique égal à deux. Cela signifie en particulier que deux éléments de cette surface dont les numéros de feuillets diffèrent de deux peuvent être identifiés. Cependant, connaître *a priori* le genre topologique de la surface de définition de la solution d'une équation différentielle est un problème difficile, et cela explique pourquoi nous nous sommes restreints à l'étude de ce genre de fonctions sur le recouvrement universel de leur surface de définition.

5.7.4 Illustration des appels récursifs

La planche 5.2 montre une deuxième série d'expériences illustrant les appels récursifs dans l'algorithme 5.5.0.1, et leur intérêt pour éviter d'avoir à prévoir une discrétisation trop fine du chemin d'intégration. Pour cela, l'équation (3.2) a été considérée, avec $\frac{5}{2}$ comme valeur pour les deux paramètres α et β. Le but derrière cette affectation est de permettre à la fonction définissant l'équation différentielle d'avoir d'importantes variations de son argument, pour que la continuité de l'argument des solutions calculées ne soit pas assurée, à la valeur de la tolérance angulaire près, et donc de provoquer des appels récursifs de l'intégrateur.

Toutes les figures sur cette planche sont orientées de la même façon, c'est-à-dire sur le plan $(0x, 0z)$. Les courbes obtenues partent toutes d'une condition initiale y_0, d'argument négatif (et de numéro de feuillet fixé égal à 0), et leur altitude décroit au

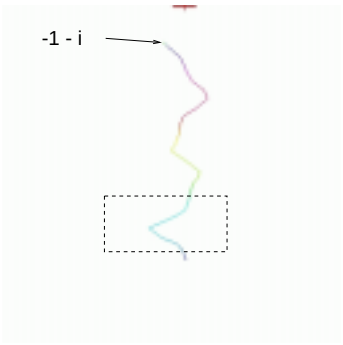


FIG. 5.11: $N = 100$

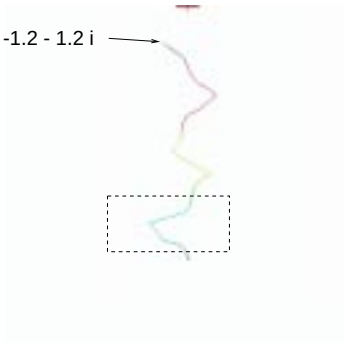


FIG. 5.12: $N = 100$

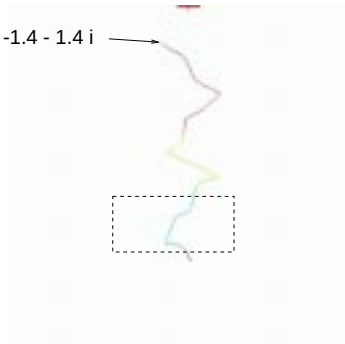


FIG. 5.13: $N = 100$: deux appels récursifs

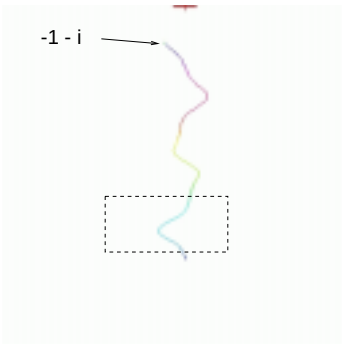


FIG. 5.14: $N = 200$

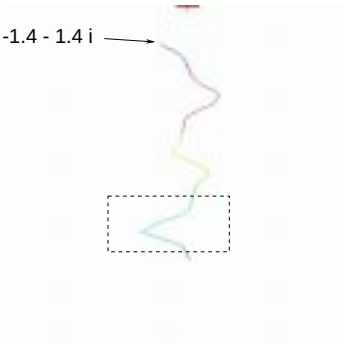


FIG. 5.15: $N = 200$: pas d'appels récursifs

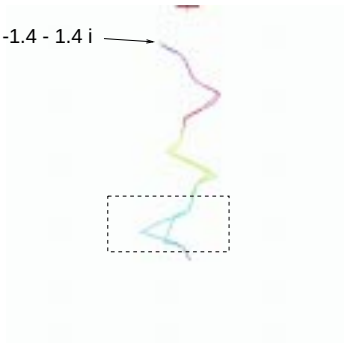


FIG. 5.16: Superposition des deux courbes

TAB. 5.2: Test des appels récursifs de l'intégrateur 5.5.0.1

cours de l'intégration. Cela signifie que les numéros de feuillet seront négatifs, voire strictement négatifs.

Figures 5.11, 5.12 et 5.13 : Cette première série d'expériences a été obtenue en considérant une discrétisation à 100 points du chemin d'intégration. Les conditions initiales ont été prises à $(1, \Leftrightarrow 1 \Leftrightarrow i)$ pour la figure 5.11, à $(\Leftrightarrow 1.2, \Leftrightarrow 1.2i)$ pour la figure 5.12, et à $(\Leftrightarrow 1.4, \Leftrightarrow 1.4i)$ pour la figure 5.13. La troisième figure a été obtenue par une intégration ayant effectué deux appels récursifs consécutifs (et non pas imbriqués). Cela se produit lorsque la continuité de l'argument de la solution calculée ne peut pas être assurée, à la tolérance angulaire près. La zone de la courbe correspondant à ce phénomène a été entourée d'un rectangle en pointillés sur les figures. Lorsque le module de la valeur initiale de y_0 augmente, la partie de la courbe présente dans cette zone a tendance a devenir de plus en plus anguleuse. Ainsi, sur la dernière figure, un certain nombre de valeurs ont été calculées de manière récursives, et ces valeurs intermédiaires n'apparaissent pas sur la figure. La correction de ces valeurs peut néanmoins être vérifiée si la discrétisation du chemin d'intégration est rendue plus fine.

Figures 5.14, 5.15 et 5.16 : La discrétisation du chemin d'intégration a été fixée à 200 points pour les trois intégrations correspondant aux trois figures. Sur la figure 5.14, ce raffinement de la discrétisation donne comme résultat une courbe plus lisse, ce qui est normal car il a été utilisé deux fois plus de points pour l'obtenir. L'intégration menant à la figure 5.15 n'a pas effectué d'appels récursifs : la zone plate de la figure 5.13 a ici été remplacé par deux segments, et l'intégrateur a pu assurer la continuité de l'argument de la solution à la tolérance près grâce à ce raffinement de la discrétisation. La figure 5.16 permet de vérifier que les valeurs obtenues en cas d'appels récursifs et celles obtenues en cas de raffinement global de la discrétisation sont bien cohérentes.

5.7.5 Conclusion

Nous avons cherché à montrer dans ce chapitre que, bien que la prise en compte des numéros de feuillet durant les calculs numériques peut paraître fastidieuse à première vue (avec la considération des indéterminations notamment), elle présente néanmoins de nombreux avantages sur des applications précises, comme ici sur des intégrations numériques d'équations différentielles. De plus, la prise en compte des indéterminations comme exceptions permet d'implémenter de tels algorithmes de manière efficace, sans recourir à des techniques de prolongement analytique, coûteuses en temps de calcul, et sans trop de difficultés techniques. Une double généralisation de ce principe pourrait être envisagée :

- Une extension de ce principe à des méthodes numériques plus évoluées que la méthode d'Euler, qui a le mérite d'être extrêmement simple à programmer, mais qui n'est qu'une méthode d'ordre 1, et donc ne donne pas des résultats

numériques de bonne qualité. Une éventuelle extension à des méthodes telles que Runge-Kutta serait probablement intéressante à mettre en oeuvre, bien qu'elle multiplierait les sources possibles d'indéterminations, et donc la complexité de programmation de la méthode.

– Une extension de ce principe à la détection non plus des discontinuités de l'argument de la solution calculée, mais aussi à des discontinuités d'ordres plus élevés. Par exemple, lorsqu'une courbe représentant une solution se présente *tangentiellement* à une coupure, l'intégrateur n'est pas en mesure de détecter une erreur de détermination à cette étape. Il pourrait être intéressant d'étudier le problème de la détection de ces comportements, de manière rapide, c'est-à-dire sans avoir à examiner le développement en série de Taylor de la solution de l'équation différentielle.

Troisième partie

L'environnement graphique GANJ

Chapitre 6

Le principe général

L'environnement graphique GANJ est prévu pour fournir des moyens simples de visualiser des objets mathématiques, à partir de données calculées dans des programmes qui ne possèdent pas de moyens pour réaliser eux-mêmes cette visualisation. L'environnement graphique GANJ peut donc être mis en commun entre plusieurs programmes, et donc permettre de comparer facilement des résultats issus de programmes différents. Cet envitonnement est basé sur un modèle Client/Serveur, que nous allons détailler dans ce chapitre. Les deux chapitres suivants détailleront une implémentation du serveur GANJ , puis deux interfaces de programmation (en C++ et en Maple).

Notons cependant que le chapitre 7 décrit une implémentation *particulière* du modèle, et que cette implémentation peut être réalisée de plusieurs manières différentes. Ainsi, celle qui sera décrite par la suite correspond à la version de développement 2 du serveur, alors qu'une troisième version plus puissante et permettant de mieux tirer parti du matériel accéléré a été mise au point plus tard. Cette dernière implémentation est décrite sur le site `http://www-lmc.imag.fr/~testard/GII`. Ces deux versions sont néanmoins conforme au même modèle, qui est l'objet de ce chapitre.

6.1 Introduction

Le modèle Client/Serveur dans le domaine des applications distribuées est utilisé dans le développement d'applications où un ou plusieurs proccesus (les clients), éventuellement répartis sur des sites distants, doivent communiquer ou exécuter un traitement au niveau d'un processus dédié (le serveur). Dans ce type d'application, les processus clients interagissent avec le processus serveur soit par le biais de primitives de communications par échange de messages (MPI), soit en utilisant un modèle de programmation de type procédural (RPC). Dans le cas de cette dernière forme d'interaction, les clients réalisent des appels de procédure à distance pour réaliser un traitement au niveau du serveur. L'échange d'informations s'établit alors pendant le transfert des paramètres des procédures et de leur résultat.

Dans ces deux modes de programmation, les clients émettent des requêtes à destination du serveur, requêtes donnant lieu à un traitement spécifique de la part du

serveur. Si un client peut transmettre plusieurs requêtes de manière concurente au niveau du serveur, on parle alors d'un mode de fonctionnement *asynchrone*, alors que si l'exécution du client est suspendue jusqu'à la complétion d'une requête, on parle alors de mode de fonctionnement *synchrone*.

La mise en oeuvre d'une application Client/Serveur peut présenter plusieurs difficultés lors de son implémentation. En effet, la gestion de la distribution des différents processus clients et serveurs répartis sur des sites potentiellement hétérogènes nécessite la prise en compte de problèmes tels que les différences de représentation interne des données sur des machines différentes, la fiabilité et l'intégrité du transfert d'informations.

Un tel modèle a de plus déjà été utilisé pour Izic, afin de permettre l'exploitation graphique des résultats de programmes écrits dans des systèmes de Calcul Formel. Cette solution ne nous a pas paru satisfaisante, essentiellement à cause des nouveaux standards et outils qui ont émergé depuis sa conception. Nous donnerons dans la section 6.4 une idée de ces nouveaux outils.

Nous allons maintenant étudier l'intérêt de ce modèle pour l'exploitation graphique de données en provenance d'applications clientes.

6.2 Le modèle GANJ

Pour appliquer le principe ci-dessus à l'environnement graphique spécifié dans le chapitre 3, les rôles respectifs du serveur et des clients doivent être précisément définis.

6.2.1 Le serveur

Le serveur reçoit des requêtes sous forme de *messages* de la part d'un client : ces messages transitent généralement par le réseau. Dans le cas particulier d'un environnement graphique Client/Serveur, ces requêtes sont généralement composées d'une ou de plusieurs *instructions* graphiques, instructions spécifiées (c'est-à-dire qu'elles y sont définies, ainsi que leur utilisation) dans un protocole.

La figure 6.1 montre l'architecture générale du serveur graphique. Lorsque le serveur reçoit un message, par l'intermédiaire d'une interface de communication, ce message doit être décodé afin d'obtenir la requête y correspondant. L'encodage des requêtes présente ici deux intérêts :

- Il permet d'assurer la compatibilité entre plusieurs types de machines. En effet, les nombres manipulés (par exemple les sommets d'un triangle dans l'espace) sont souvent de type *flottant*, et donc sont soumis à diverses méthodes de codage de manière interne à une architecture. Un format standard pour les nombres flottants doit donc être considéré, et chaque implémentation (d'un serveur ou d'un client) dispose des primitives d'encodage ou de décodage lui permettant de se conformer à ce format standard.

– Il permet de regrouper plusieurs instructions dans la même requête (*bufferization*). Le décodage consiste dans ce cas à retrouver à partir d'une requête l'ensemble des instructions qui la composent.

Les instructions graphiques spécifiées par le client obtenues après décodage sont adressées ensuite à l'*interpréteur graphique*, qui les transforme en requêtes vers le matériel, et donc, du point de vue de l'utilisateur, en résultats graphiques. Le décodage des messages, l'interprétation et la génération des résultats graphiques correspondant à la requête du client sera désigné dans la suite par *traitement graphique*.

Le serveur peut de plus activer certains mécanismes de sélection afin de prendre en compte des interactions avec un utilisateur (par exemple le résultat d'un pointage à la souris effectué par un utilisateur). Dans ce cas, une fois que ces mécanismes de sélection ont récupéré une valeur, le serveur peut renvoyer cette valeur au client qui l'a requise, après l'avoir encodée.

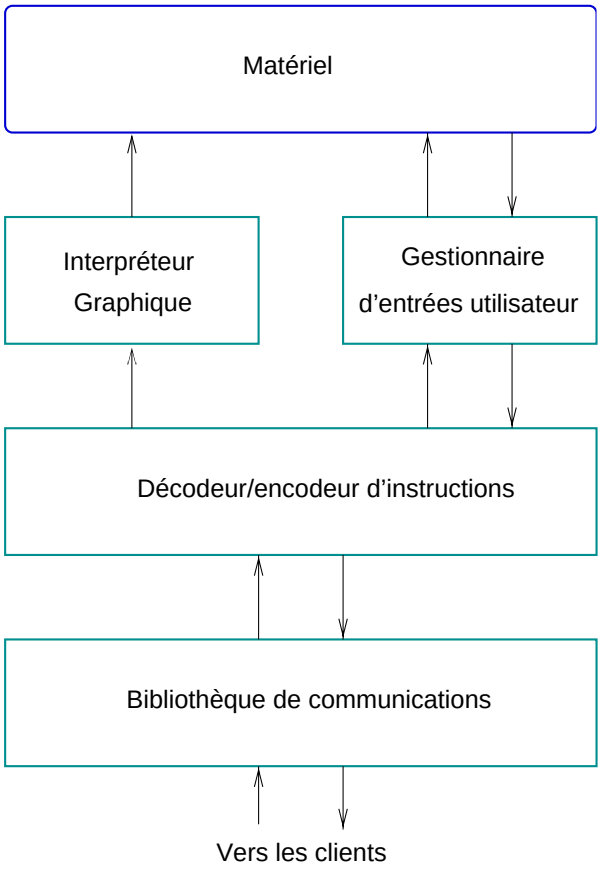


FIG. 6.1: Les différentes composantes du serveur

L'intérêt du découpage en couche montré sur la figure 6.1 permet de modulariser la différentes actions du serveur, et par là même de spécialiser certaines parties du serveur : par exemple, le décodage de nombres codés en double précision peut être optimisé sur certains types de matériel, la partie graphique peut aussi être adaptée à

la prise en compte d'accélérateurs matériels, et la partie de décodage des instructions peut être parallélisée pour un décodage plus efficace des instructions graphiques.

6.2.2 Le client

La figure 6.2 montre l'architecture générale d'une application cliente. Le client envoie des requêtes au serveur, qui sont encodées avant d'être transmises à la bibliothèque de communication. Ces requêtes sont composées d'instructions graphiques (les primitives géométriques standards pour la modélisation de scènes 3D, des primitives permettant de définir des options) et de requêtes autorisant les interactions avec un utilisateur. Ce dernier type de requêtes est prévu lorsque l'exécution d'une application cliente est soumise à un choix de l'utilisateur (par exemple, la spécification graphique d'une valeur). Dans ce cas particulier, le client transmet une telle requête au serveur qui doit lui renvoyer un résultat. Ce résultat peut être le résultat attendu par l'utilisateur (des valeurs numériques), ou une annulation du processus de sélection. Ce résultat sera ensuite transmis au client *via* la bibliothèque de communications.

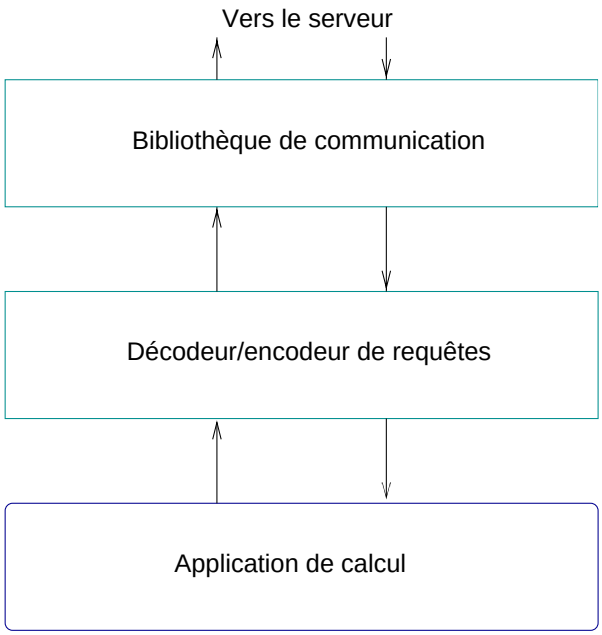


FIG. 6.2: Architecture générale des clients

Le protocole spécifiant les différentes instructions graphiques est commun au serveur et aux clients. De manière générale, un client effectue un calcul, puis envoie au serveur les requêtes qui permettront d'exploiter graphiquement ce calcul. Ces requêtes peuvent être émises pendant le calcul (on parle notamment de la génération de *traces* graphiques dans le programme), ou après que le calcul soit fini (les requêtes sont bufferisées). Dans les exemples de la partie IV, le principe est d'abord de calculer un premier ensemble de valeurs, puis d'exploiter graphiquement ces valeurs, d'attendre

une réponse d'un utilisateur et de recommencer depuis le début avec les nouvelles valeurs spécifiées par l'utilisateur.

6.3 Intérêt du modèle

Les figures 6.1 et 6.2 montrent que les clients et le serveur sont deux entités indépendantes, pouvant communiquer entre elles. Par conséquent, rien n'oblige ces deux entités à coexister (physiquement) sur la même machine. En effet, de nombreuses bibliothèques de communication permettent d'envoyer et de recevoir des données par un réseau, tout en garantissant leur intégrité. Cela permet d'envisager une situation comme celle décrite sur la figure 6.3. Cette situation peut se révéler avantageuse, à condition que les communications par le réseau soient suffisamment efficaces (latences, etc ...).

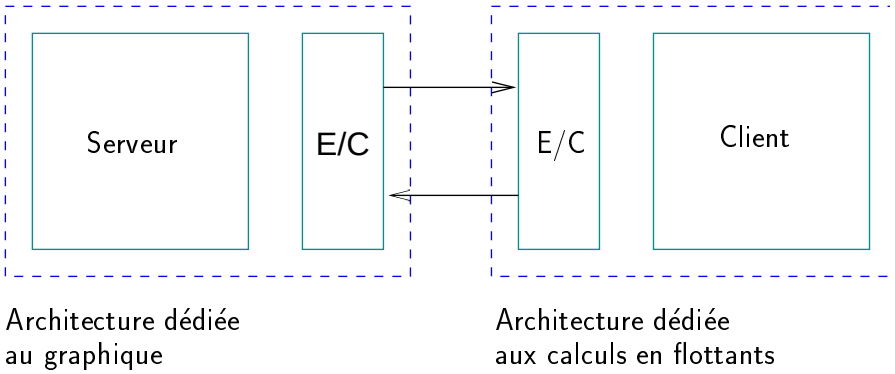


FIG. 6.3: Un exemple d'utilisation

La configuration décrite sur la figure 6.3 permet de localiser les opérations graphiques (généralement coûteuses en temps de calcul si elles ne sont pas accélérées matériellement) sur une architecture dédiée à ce type d'opérations, mais aussi de localiser les calculs proprement dits sur des architectures optimisées pour les calculs en nombres flottants. Cela permet d'espérer une exploitation efficace des resources propres à chaque machine.

Le codage des instructions que le client adresse au serveur permet de s'affranchir de contraintes de *langage* en considérant plutôt une *grammaire*, qui sera décrite ultérieurement en 6.5. Cette grammaire s'exprime de manière sous-jacente dans la définition du protocole de communication entre client et serveur, et se trouve être indépendante non seulement du langage de programmation du client, mais aussi du type de machine sur lesquelles sont basées les différentes composantes du modèle. Cela permet notamment de pouvoir choisir de manière indépendante les langages de programmation, les systèmes d'exploitation et le type de chaque machine pour chaque composante du modèle. Ainsi, les clients peuvent être implémentés en utilisant un langage de programmation adaptés à leurs caractéristiques. Par exemple, un client peut être écrit en utilisant l'interface de programmation écrite sous Maple,

alors que le serveur peut être programmé en C, sans que cela ait une importance sur les résultats graphiques générés.

La centralisation au niveau du serveur des appels à des primitives graphiques permet de faire assumer au serveur l'aspect de *manipulation* de scènes graphiques, ainsi que l'aspect *interactivité*, et permet de donner une interface standard et unique à toutes les expériences puisque seuls les clients changent, pas le serveur. Ce principe permet de fournir une certaine facilité d'utilisation aux utilisateurs de l'environnement graphique.

Afin de permettre à un utilisateur de contrôler au maximum la validité de l'exploitation graphique de ses données, une grammaire assez rigide a été mise en place, associée à un mécanisme de contrôle des instructions émises par un client. Ainsi, une instruction hors de son contexte sera détectée et une erreur sera renvoyée au client. Nous verrons d'ailleurs en 7.3 que le traitement des erreurs peut être de deux natures différentes : soit exhaustif, avec un mécanisme de compte-rendus au client pour toutes les requêtes, soit simplifié au profit d'un gain important en termes de puissance.

6.4 Choix d'implémentation

En ce qui concerne les traitements graphiques et réseaux au niveau du serveur, il est particulièrement important de bien choisir les outils qui vont être utilisés. Ces outils doivent être :

conformes à des standards : en effet, le serveur GANJ, ainsi que ses clients, ne doivent pas être dépendants des changements de version d'une bibliothèque ou d'un outil particuliers. De plus, il est souhaitable que le serveur puisse évoluer de manière conjointe avec les outils utilisés, et cela sans prévoir des versions particulières.

résidents sur de nombreux types de machines : dans l'optique d'expérimentations massives, il est intéressant de pouvoir s'affranchir d'un type de machines particulier, car cela permet une plus grande souplesse d'utilisation.

Ces deux points ont donc particulièrement motivé les deux choix suivants :

6.4.1 Graphique

OpenGL [NDW93, Kil96, SA96] est un standard graphique pour le rendu en trois dimensions, implémenté sur plusieurs sortes de machines : il en existe des implémentations matérielles (sur des machines comme des Silicon Graphics, IBM–RS6000, SUN[1]), mais aussi des implémentations logicielles, fonctionnant sur de nombreux types de machines et de systèmes d'exploitation (comme l'implémentation gratuite Mesa, conforme au standard OpenGL1.1). Généralement, chaque implémentation propre à un type de machines exploite au mieux les ressources de la machine : ce

[1]bien que l'implémentation de SUN ne respecte pas tout à fait le standard.

point est particulièrement sensible en ce qui concerne les accélérations matérielles [Kil97].

OpenGL est associé à GLX dans le système de fenêtrage X Window, qui est un protocole permettant à OpenGL de fonctionner sur un réseau, par le biais du serveur X [Kil96].

6.4.2 Communications

TCP/IP[Ste90, CS91, CS93] est un protocole de communications composé de deux parties :

- IP est un protocole d'envoi de données sur le réseau,
- TCP est un protocole qui gère les communications entre deux machines. Il permet de plus de s'assurer l'intégrité des données et de communiquer par *sockets*.

Le protocole TCP/IP permet de communiquer simplement sur un réseau, par l'envoi de paquets de données entre deux machines, en particulier UNIX, ce protocole étant implémenté sur la quasi-totalité des machines UNIX existant à ce jour. De plus, le parc de machines utilisées pouvant être de nature hétérogène, l'échange de données entre deux programmes s'exécutant sur des machines différentes est rendu particulièrement facile par l'utilisation de ce standard.

6.5 La grammaire de Ganj

Une interface de programmation permet de faire l'interface entre un langage de programmation et les structures de données propres à une implémentation : elle est composée d'un ensemble de primitives permettant de manipuler ces structures de données. Nous noterons par la suite *API* pour *Application Programming Interface*. Notons d'abord qu'une API est associée à un langage et à un concept de programmation, alors que l'architecture générale des clients décrite dans le paragraphe 6.2.2 ne l'est pas. Plutôt que de décrire une API, nous décrirons une *grammaire*, qui sera appliquée à chaque langage ou concept de programmation.

Les API qui ont été implémentées sont des spécialisations à certains langages de programmation ou à certains concepts de la grammaire que nous allons décrire. Pendant l'exécution d'un programme, les appels aux primitives composant l'API construisent des messages qui sont envoyés au serveur, messages en accord avec cette grammaire.

Cette grammaire est décrite dans la table 6.1, page 120, dans le formalisme BNF, avec les éléments du vocabulaire terminal en **gras**.

GANJProgram	::=	**Inits** *PackedInstructions* **Close**
PackedInstructions	::=	*Instruction PackedInstructions*
Instruction	::=	*UserInput*
		| **ConfigRequest**[a]
		| *GlobalOption*
		| *BeginEndPair*
UserInput	::=	**MouseInput**
		| **KeyboardInput**
		| **MixedInput**
GlobalOption	::=	**Xmin**
		| **Xmax**
		| **Ymin**
		| **Ymax**
		| **Zmin**
		| **Zmax**
		| **QueryDL**
		| **NeedLib**
		| **NeddFunc**
		| **CustomGlobalOption**
BeginEndPair	::=	**Begin** *BeginEndPairInstructions* **End**
BeginEndPairInstructions	::=	*GraphicInstruction BeginEndPairInstructions*
GraphicInstruction	::=	**Packed**
		| *LocalOption*
		| *Action*
LocalOption	::=	**Color**
		| **Dashpattern**[a]
		| **PackedMode**
		| **Outline**
		| **Normal**
		| **CustomLocalOption**
Action	::=	**Point**
		| **Segment**
		| **Triangle**
		| **Quadrangle**
		| **CustomAction**

[a]Fonctionnalités non encore implémentées.
[a]

TAB. 6.1: La grammaire de GANJ

6.6 Le principe d'une session

Nous appellerons *session* l'ensemble des échanges de données entre le serveur et un client. Une session commence par une connection du client au serveur[2]. Lorsqu'un client notifie au serveur qu'il essaie de se connecter, le serveur détermine alors un canal de communication (sous UNIX, un *socket*) par lequel vont transiter l'ensemble des messages constituant la session. Après une phase d'initialisation, où client et serveur échangent leurs caractéristiques, le client peut émettre en direction du serveur des requêtes de types particuliers : les *GANJ_Rep* (pour représentation), les *GANJ_SubRep* (pour sous-représentation) et les *GANJ_UserInput*. Chacun de ces types de requêtes se retrouvent dans la grammaire décrite en 6.5.

Les GANJ_Rep : d'un point de vue logique, les instructions qui entrent dans cette catégorie sont celles qui ont une influence sur la *totalité* des instructions qui suivront. Si par exemple un client émet une requête contenant une instruction **Xmin**, cela signifie que la scène composée ne prendra pas en compte les (futures) instructions dont les sommets auront une valeur de x inférieure à la valeur spécifiée par l'argument de l'instruction **Xmin**.

Les GANJ_SubRep : la différence avec la catégorie précédente tient à l'introduction d'une notion de portée : certaines instructions ont une portée limitée par la validité de la GANJ_SubRep. Cette catégorie d'instruction comprend d'une part les "vraies" instructions graphiques (les *actions*), et d'autre part des instructions qui n'ont pas la même portée que celles contenues dans une GANJ_Rep (les *options locales*). Si par exemple un client émet l'instruction **Color**, l'ensemble des actions requises par la suite ne nécessitant pas la spécification d'une couleur utiliseront la couleur spécifiée comme argument de **Color**, ceci jusqu'à la prochaine instruction **Color** rencontrée par l'interpréteur graphique, ou jusqu'à la fin de la validité de la GANJ_SubRep.

Les GANJ_UserInput : cette catégorie d'instruction particulière permet à un client de déclencher l'activation au niveau du serveur d'un mode intéractif, afin de récupérer des valeurs spécifiées par l'utilisateur.

Ce partitionnement "rigide" en trois grands types d'instructions est motivé par de multiples raisons.

- Une des premières motivations est de rapprocher la grammaire de GANJ de celle d'OpenGL, qui est elle-même extrèmement rigide. Dans ce cadre, les GANJ_SubRep correspondent aux paires *Begin/End* d'OpenGL [NDW93]. Cela permet de ne pas avoir à ordonner les instructions issues des clients pour les rendre conformes à la grammaire d'OpenGL, et en conséquence de bénéficier des mêmes accélérations que si le client émettait directement des appels OpenGL. Ainsi, le regroupement des instructions générant des primitives géométriques dans la catégorie des GANJ_SubRep permet de ne pas générer d'erreurs en provenance de la couche OpenGL sous-jacente.

[2] cela signifie notamment que le serveur est constamment en attente d'une connection.

- D'autre part, les différentes actions sont toutes regroupées dans une même GANJ_SubRep afin de rendre leur traitement (c'est-à-dire leur décodage, vérification et interprétation) *asynchrone*[3]. Cela peut mener à des traitements en parallèle de requêtes issues du même client, et cela peut apporter en conséquence une amélioration des performances du serveur.
- Les GANJ_UserInput nécessitent un traitement particulier de la part du serveur. En effet, lorsqu'une instruction de cette catégorie est requise par un client, le serveur active son mode interactif. Cela nécessite ainsi que le serveur trace les scènes déjà composées, et pour cette raison une GANJ_UserInput *ne peut pas être exécutée pendant une phase de spécification de scène*, et donc à l'intérieur d'une GANJ_SubRep. Par contre, une GANJ_UserInput doit être attachée à une GANJ_Rep, afin de bénéficier des tracés précédemment effectués, et des options globales.
- Enfin, des instructions particulières ont été spécifiées dans la grammaire, permettant d'exécuter du code précompilé, présent dans des librairies et chargé dynamiquement. Afin de garantir une sécurité de fonctionnement au niveau du serveur, puisque certaines instructions comprises dans ces instructions peuvent générer des erreurs au niveau de la couche OpenGL, une des catégories ci-dessus doit être *obligatoirement* affectée à chacune de ces instructions. Cette affectation revient au programmeur de l'application cliente, puisque celui-ci est *a priori* le seul à savoir quels types d'appels de fonctionalités d'OpenGL sont présentes dans le code chargé dynamiquement, et donc dans quelle catégorie ranger ces instructions. La rigidité de la classification permet donc de garantir un certain niveau de sécurité dans l'exécution des ces instructions non connues statiquement par le serveur.

[3]cela demande cependant un certain effort de synchronisation de la part du programmeur d'une application cliente, en particulier pour l'utilisation des options locales

Chapitre 7

Le serveur

Nous allons étudier les détails (techniques) d'une implémentation qui a été réalisée effectivement, dans un environnement UNIX. Nous verrons ainsi quels ont été nos choix en ce qui concerne l'interpréteur graphique (en particulier, les choix concernant le modèle de programmation graphique, de système de fenêtrage ainsi que la boîte à outils utilisée pour l'interface graphique), avant d'étudier les aspects reliés aux requêtes proprement dites (c'est-à-dire le traitement des erreurs, le décodage des données et la bibliothèque de communication).

7.1 L'interpréteur graphique

7.1.1 La librairie graphique

La partie graphique est programmée avec Mesa, qui est une implémentation *logicielle* gratuite d'OpenGL. Ce choix peut sembler à première vue limitatif, car l'informatique graphique peut avoir des besoins en termes de puissance tellement importants que le choix d'une librairie uniquement logicielle peut sembler pénalisant, notamment en terme de performances. Cependant, ce jugement peut être nuancé, grâce aux points suivants :

- Mesa, comme les implémentations mixtes logicielles/matérielles d'OpenGL, est l'implémentation d'un standard. Cela signifie que non seulement les API (définies en 6.5) sont point par point identiques, mais que les effets d'un appel d'une primitive de l'API sont strictement définis dans les spécifications du standard[Kil97]. Ainsi, la même application faisant appel à des primitives de l'API d'OpenGL peut utiliser indifférement n'importe quelle implémentation d'OpenGL[1], et ses résultats sont sensés être les mêmes. Bien que le serveur GANJ n'a pas pu encore être testé sur des machines où réside une implémentation matérielle d'OpenGL (pour des raisons bassement matérielles de manque de moyens) il serait intéressant de voir si ce point peut être testé.

[1]de plus, grâce à des mécanismes d'édition dynamique de liens, une recompilation du serveur n'est théoriquement pas obligatoire.

– En général les implémentations d'OpenGL sont payantes, ou fournies avec certaines machines, ce qui peut être pénalisant quant à la diffusion d'un logiciel, au moins dans la communauté scientifique. Mesa est une implémentation *shareware* d'OpenGL, et de fait peut être récupérée par n'importe quelle personne intéressée.

Les diverses implémentations d'OpenGL (logicielles, matérielles) fonctionnent de la même façon. Les divers objets à représenter sur un écran sont spécifiés sous forme de sommets, et sont empilés sur un *pipeline* graphique. Suivant le mode spécifié (par exemple des triangles), et suivant les différentes options ou modes de représentation (par exemple, sans rendu de lumière, mais avec un ombrage porté sur les triangles) la scène sera représentée dans un *buffer* (le *frame buffer*) à partir du décodage des sommets (dans l'exemple des triangles, trois sommets par trois). Ce fonctionnement[2] apporte une souplesse considérable tout en laissant de grandes facilités pour accélérer le processus, notamment de manière matérielle.

L'interpréteur graphique, lorsqu'il traite une instruction, la transforme en un certain nombre de primitives élémentaires d'OpenGL et fait appel à ces primitives. Par le biais des mécanismes internes d'OpenGL, ces primitives seront interprétées par la suite, et le rendu de la scène tri-dimensionnelle finale sera effectué par OpenGL.

Les instructions pouvant être traitées par l'interpréteur graphique peuvent être de deux types :

les instructions simples : ce sont les instructions présentes par défaut dans la grammaire, comme par exemple **Triangle**, **Xmin** ou **Normal**. À ce type d'instructions correspond généralement une primitive graphique d'OpenGL, ce qui fait que lorsque le serveur ne traite que des instructions simples, il permet d'exécuter à distance du code OpenGL, sur une machine *a priori* non prévue pour cela. L'intérêt principal est de piloter une application réalisant des appels OpenGL sur un ensemble de machines, en particulier ne supportant pas l'extension GLX.

les instructions complexes : ces intructions sont *chargées dynamiquement* sur requête du client, c'est-à-dire qu'elles sont accessibles au serveur sous forme de librairies. Le serveur charge le code correspondant à l'instruction lorsque le client le lui notifie par l'intermédiaire des instructions **NeedLib** et **NeedFunc**, et exécute le code précédemment chargé lorsque le client requiert les intructions **CustomAction**, **CustomLocalOption** ou **CustomGlobalOption**. Ces instructions complexes permettent d'exécuter du code sur la machine hébergeant le serveur, et ainsi permet de faire l'économie des transferts sur le réseau des différentes données géométriques présentes dans l'instruction. Nous reviendrons plus en détail sur ces primitives dans le chapitre 8.2, lors de la description de l'API C++.

[2]pour plus de détails, voir [Kil96].

7.1.2 Le système de fenêtrage

Les spécifications d'OpenGL en font un standard graphique général, *a priori* indépendant du système d'exploitation, de la machine, et de plus il n'a pas été prévu dans ces spécifications si les applications basées sur OpenGL doivent s'exécuter dans un système de fenêtrage ou pas. Cette liberté dans l'implémentation permet une grande souplesse au niveau des choix matériels et logiciels, par le biais de la spécification d'un contexte de tracé. Ce contexte peut être ensuite attaché à une fenêtre, à un fichier, à une zone de mémoire, etc ...

Le système de fenêtrage X Windows[OQL88] est de loin le plus répandu sur les stations de travail Unix. Ce système permet de gérer le tracé de scènes tri-dimensionnelles dans des fenêtres, par le biais de l'extension GLX. Pour l'instant, Mesa n'offre pas encore une interface GLX correcte, et les liens entre X et OpenGL ont été effectués de manière spécifique. Cependant, dès qu'une version *shareware* de GLX verra le jour, il pourra être intéressant d'y adapter le principe de fonctionnement du serveur GANJ afin de profiter de l'extension GLX.

L'intégration d'OpenGL dans le système de fenêtrage X permet d'assurer une bonne portabilité du serveur, ainsi que de fournir les facilités d'utilisation classiquement liées à X, en particulier l'exécution et l'affichage sur deux hôtes différents. D'un point de vue programmation, cela permet de définir facilement des primitives de gestion des périphériques comme la souris, mais aussi d'adresser les *buffers* d'OpenGL comme des fenêtres X. Tous ces mécanismes assurent au programmeur une gestion cohérente des évènements X prédéfinis, et permettent ainsi d'adopter facilement un modèle de programmation par évènements, modèle des plus utilisés de nos jours.

7.1.3 L'interface graphique

Pour gérer de manière efficace les différentes informations issues du système de fenêtrage, il peut être intéressant d'adopter une *boîte à outils*, c'est-à-dire un ensemble de méthodes de haut niveau permettant d'effectuer des traitements classiques à partir des évènements X. Tk[Ous94, Wel95] est une *boîte à outils* permettant d'avoir une gestion hiérarchique d'entités telles que les fenêtres, mais aussi leurs composantes internes, *boîte à outils* elle-même basée sur un langage de commandes (*scripting language*), Tcl.

Cette surcouche permet facilement de gérer les interactions avec l'utilisateur, comme par exemple des évènements de type "actions à la souris", et offre des facilités de créations de boutons, de gestion des périphériques (souris, clavier, ...), de gestion des fenêtres (redimensionnement de fenêtres, création d'autres fenêtres de configuration, ...) , de gestion des couleurs (création et changement de table de couleurs), etc ...

Un autre argument en faveur de cette *boîte à outils* est qu'elle dispose, par le biais de Tcl, d'un langage de description interprété (par opposition à compilé), indépendant de la machine sur laquelle il s'exécute, ce qui améliore encore la facilité de portage de l'application. Ainsi, les diverses définitions de boutons, de fenêtres sont interprétées

et n'ont donc pas besoin d'être recompilées en fonction de la machine sur laquelle le serveur est exécuté.

Enfin, Tcl/Tk est un produit *shareware* contrairement à des *boîte à outils* plus répandues comme Motif, ce qui en fait un produit facilement accessible.

7.1.4 Résumé

Nous pouvons résumer les rôles de ces différentes composantes sur la figure 7.1.

Les flèches sur cette figure symbolisent les interactions entre chaque entité qui y est dessinée. Ainsi, nous voyons que l'interpréteur graphique *envoie* des commandes OpenGL, qui seront affichées dans la fenêtre du serveur. Cette interaction n'est que dans un seul sens, contrairement aux interactions entre la fenêtre du serveur et l'interface graphique, afin de prendre en compte les interactions avec l'utilisateur du serveur.

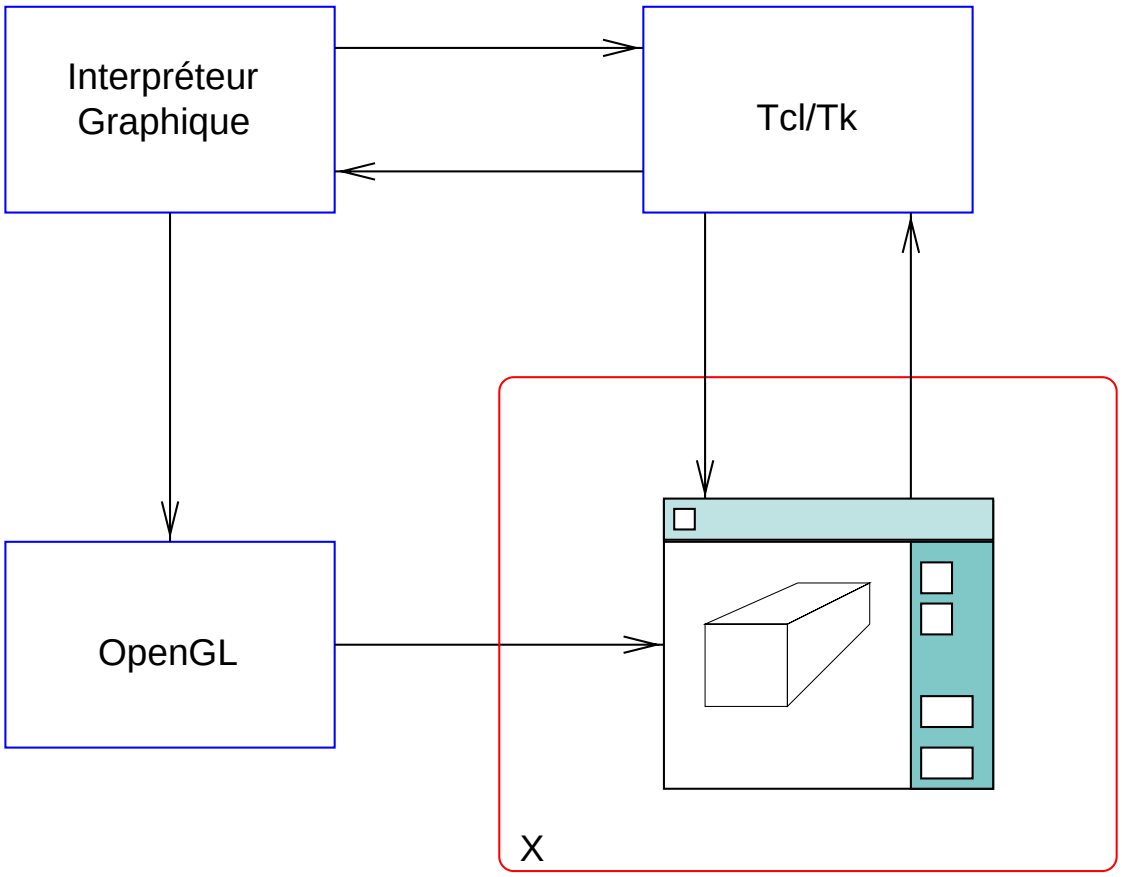


FIG. 7.1: Les différentes composantes graphiques

7.2 Le traitement des erreurs

Les erreurs qui peuvent se produire pendant une session peuvent avoir plusieurs provenances :

- Elles peuvent provenir d'un des modules du serveur, et être occasionnées par une mauvaise manipulation, conséquence des requêtes du client. Par exemple, si le client émet une requête de type *Action*, alors qu'aucune GANJ_SubRep n'a été ouverte. Un autre type d'erreur peut se produire dans l'exécution d'une action chargée dynamiquement, ne respectant pas la grammaire d'OpenGL. Dans ce cas, c'est la couche OpenGL sous-jacente qui émet une erreur.
- Elles peuvent aussi provenir d'un module externe au serveur : à titre d'exemple, une erreur de réseau peut occasionner une erreur au niveau du serveur lorsque celui-ci essaie de récupérer un message.

Il est particulièrement important de détecter ces erreurs, dans un premier temps pour prévenir l'exécution de mauvaises instructions dans les différents modules du serveur (car des mauvaises instructions peuvent donner des résultats imprévisibles, voire même arrêter l'exécution du serveur), puis pour notifier au client que sa requête n'est pas valide, afin que le programmeur de l'application cliente puisse modifier la requête incriminée. Pour ces deux raisons, les erreurs sont classées en deux catégories distinctes :

Les erreurs dépendant du client : ces erreurs sont dans la plupart des cas des erreurs de grammaire : le flot de requêtes émanant du client ne respecte pas la grammaire présentée dans la table 6.1. Elles peuvent être aussi des erreurs de syntaxe, c'est-à-dire que la syntaxe de certaines requêtes émanant du client peut être incorrecte (mauvais nombre d'arguments, mauvais types d'arguments, mauvais nom d'instruction). Ces erreurs sont prises en compte au niveau du serveur, et notifiées au client.

Les erreurs indépendantes du client : ce sont typiquement des erreurs dues à des problèmes de réseau, ou des erreurs d'exécution du client. Dans ce cas, le serveur retourne à son état initial (avant la connection du client), sans notification spéciale au client.

7.3 Décodage des messages

Le serveur GANJ s'exécute sur plusieurs types de machines, pas forcément accélérées matériellement pour le graphique. Cela implique en particulier que la quantité de calculs inhérente à toute représentation tri-dimensionnelle pourra être faite de manière logicielle, et cette quantité n'est en général pas négligeable. Un test (détaillé en 7.5) montre que pour une expérience traçant un ensemble de 1600 triangles et deux boules composées chacune de 1250 triangles, avec un calcul de lumière, le temps utilisateur de calcul monte à 6 secondes, dont plus de trois consacrées au remplissage

d'une `SubRep`, le reste étant occupé par le reste de la session. Cela signifie que trois secondes se sont écoulées entre l'ouverture de la GANJ_SubRep et sa fermeture.

Ce temps n'est probablement pas optimal. En effet, pendant qu'une GANJ_SubRep est ouverte, le serveur continue à recevoir des requêtes, à décoder des messages, etc. L'efficacité du serveur passe donc par une utilisation efficace de la resource graphique. Nous allons étudier dans la suite en quoi la multi-programmation légère peut être utile pour cela.

7.3.1 Les processus légers

Les *processus légers* [GGM93, dd94] (en anglais *threads*) sont des fils d'exécution qui s'exécutent de manière concurrente dans un même espace d'adressage, contrairement aux processus UNIX ou processus lourds qui s'exécutent dans des espaces d'adressage différents.

Un processus lourd peut contenir plusieurs milliers de processus légers se partageant l'espace mémoire du processus lourd[Riv97]. Chaque processus léger qui s'exécute dans le contexte d'un processus lourd partage l'espace d'adressage de celui-ci. Les différents processus légers s'exécutent de manière indépendante, et plusieurs processus légers peuvent communiquer facilement par l'utilisation de variables partagées.

Un autre intérêt des processus légers est leur rapidité de commutation entre deux processus légers, ainsi que leur rapidité de création. De plus, contrairement aux processus lourds dont la gestion de l'ordonnancement est effectuée par le système, la gestion de l'ordonnancement des processus légers est gérée par l'utilisateur.

Enfin, lorsqu'un programme met en jeu plusieurs processus légers, on parle de multi-programmation légère (*multi-threading*). L'utilisation des processus légers permet ici de paralléliser de manière simple et efficace certains traitements (intervenant dans le décodage des messages) qui n'ont pas besoin d'être exécutés séquentiellement.

7.3.2 Les messages empaquetés

Lorsque plusieurs instructions sont empaquetées dans un même message, et que ces messages sont destinés à être traités en parallèle, on ne peut plus compter sur des notions d'ordre entre les instructions. Ainsi, une des contraintes les plus fortes en ce qui concerne l'empaquetage des instructions est qu'*on ne peut empaqueter que des instructions indépendantes*. Ainsi, les seules instructions suceptibles d'être empaquetées sont les instructions apparaissant dans les GANJ_SubRep, et par conséquent les actions et les options locales.

Le traitement des messages peut être accéléré en partageant leur décodage entre plusieurs processus légers. De plus ces processus légers peuvent accéder à l'interpréteur graphique de manière indépendante, ce qui permet d'espérer une exploitation plus efficace de l'interpréteur graphique.

7.3.3 Principe du placement des messages

Pour que le décodeur puisse traiter en parallèle les requêtes que lui envoie un client, il a la structure décrite figure 7.2. La figure montre trois parties indépendantes. Le décodeur est composé des deux premières parties (étiquetées **I** et **II** sur la figure). La troisième partie (étiquetée **III**) est l'interpréteur graphique décrit en 7.1. Les parties **I** et **II** suivent un modèle de *Producteur/Consommateurs*.

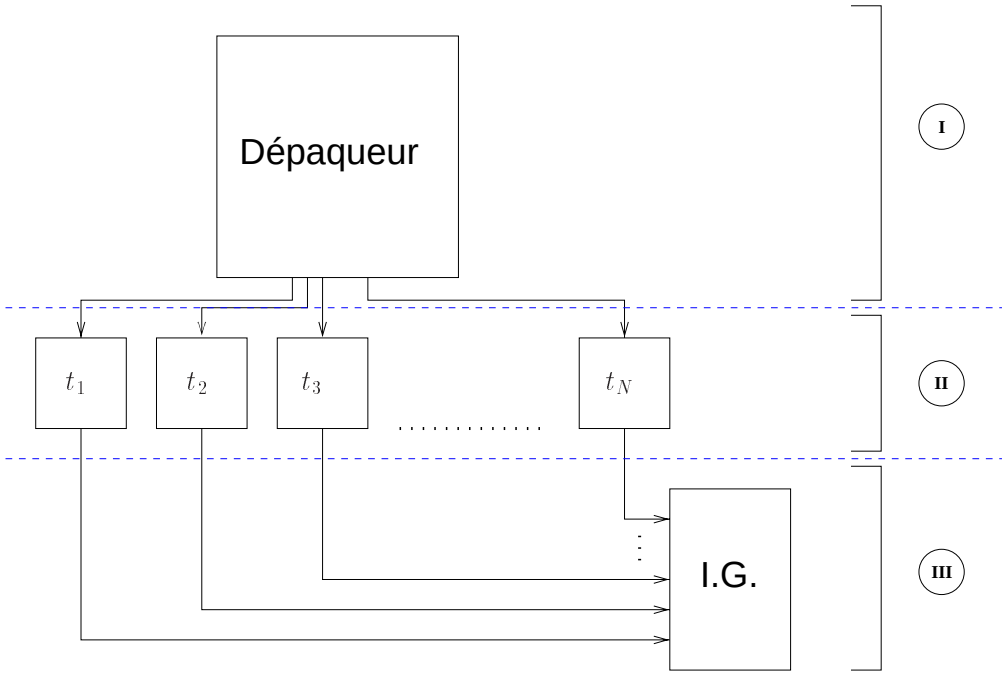


FIG. 7.2: Structure du décodeur et accès à l'interpréteur graphique

- La première partie (**I**) est composée de ce qui a été appelé sur la figure le *Dépaqueur* : ce dépaqueur joue un rôle de producteur. C'est lui qui, couplé à l'interface de communication, reçoit des requêtes sous forme de messages de la part du client et les met à disposition des processus légers qui vont les décoder.
- La deuxième partie (**II**) regroupe N processus légers[3] qui jouent le rôle de *consommateurs* : ils sont bloqués en attente d'un message de la part du dépaqueur, et décodent le message que le dépaqueur leur a affecté lorsqu'ils sont débloqués.
- Enfin, sur la troisième et dernière partie (**III**) est représenté l'accès à l'interpréteur graphique : lorsqu'un consommateur obtient une instruction à partir du message, cette instruction est passée à l'interpréteur graphique.

[3]ce nombre peut être géré dynamiquement et ainsi être adapté aux capacités matérielles (puissance, mémoire) de la machine hôte du serveur.

7.3.4 Synchronisation des processus légers

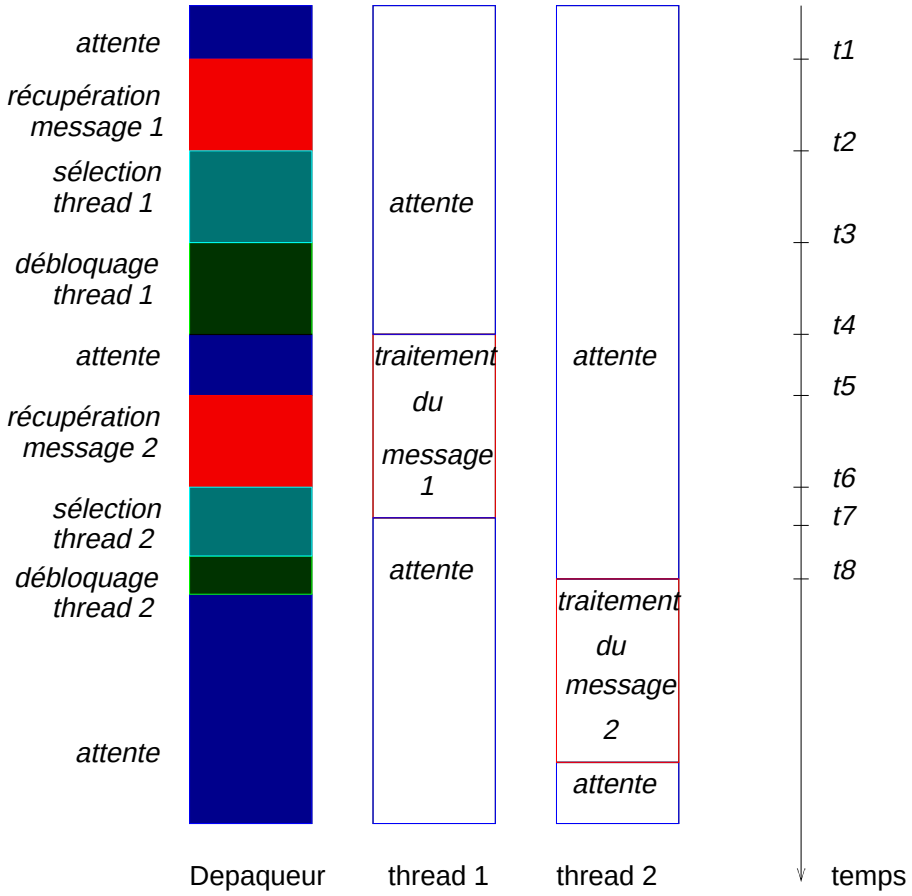


FIG. 7.3: Synchronisation des processus légers

La synchronisation des processus légers est très importante dans ce problème (comme dans toutes les applications utilisant la multi-programmation légère [Tan92]), surtout en ce qui concerne la génération d'instructions graphiques dans un langage aussi rigide qu'OpenGL : comme la grammaire de ce langage est très stricte, les instructions ne peuvent pas être interprétées dans n'importe quel ordre. Par exemple, des instructions de type *Action* ne peuvent être interprétées lorsque aucune GANJ_SubRep n'a été créée. Ainsi, si le traitement des instructions est mal synchronisé, une *Action* peut être interprétée avant la création de la GANJ_SubRep, alors que le client avait spécifié *dans le bon ordre* ces deux instructions, ce qui entraîne une erreur du serveur. La synchronisation entre toutes ces tâches est expliquée sur la figure 7.3.

Sur cette figure sont représentées les différentes actions qui se produisent pendant l'exécution des processus légers en fonction du temps. Le nombre de processus légers n'est ici que de deux dans un but de simplification, mais en général le serveur fonctionne en mettant en jeu plus de consommateurs.

Dans un premier temps, le dépaqueur est en attente d'un message de la part du client. Lorsque ce message arrive (temps t1), le dépaqueur le récupère, puis choisit le consommateur auquel le décodage du message sera confié (temps t2) : diverses stratégies pour ce choix ont été implémentées, la plus courante est de choisir le premier processus léger qui a été placé en attente. Ensuite, lorsqu'un processus léger est choisi, le dépaqueur le débloque (temps t3), puis se remet en attente d'un message en provenance du client.

Lorsque le deuxième message arrive (temps t5), et que le dépaqueur doit choisir un consommateur à qui confier le décodage du message (temps t6), nous voyons que le processus léger numéro 1 est encore occupé à traiter le premier message : le dépaqueur choisit donc tout naturellement le processus léger numéro 2 pour traiter le deuxième message (temps t7).

7.3.5 Protection des données

Mesa, l'implémentation logicielle d'OpenGL utilisée pour exploiter l'environnement graphique Ganj n'est pas intrinsèquement sécurisée pour l'utilisation de plusieurs processus légers. C'est pourquoi les données envoyées à l'interpréteur graphique ont besoin d'être protégées, notamment par des mécanismes d'exclusions mutuelles. Ainsi, à chaque fois qu'un consommateur envoye une instruction à l'interpréteur graphique, il ne doit pas le faire en même temps qu'un autre car des erreurs, notamment au niveau de la composition des polygones pourraient se produire. Par exemple, dans l'hypothèse où aucun mécanisme de protection de données n'est utilisé, si deux consommateurs spécifiaient chacun un polygone en même temps, il y aurait de fortes chances que les sommets des deux polygones se mélangent et que le résultat obtenu soit assez différent du résultat escompté.

Nous avons donc défini un ensemble d'indicateurs d'exclusion mutuelle commun à tous les processus légers. A chaque fois qu'un consommateur accède à l'interpréteur graphique, il consulte un indicateur. Cette consultation peut être bloquante, auquel cas le processus léger correspondant ne pourra être débloqué que lorsque l'indicateur aura été libéré. Nous pouvons voir ce principe sur l'exemple de la figure 7.4.

Sur cette figure, les deux consommateurs ont besoin de se synchroniser pour accéder à l'interpréteur graphique, et le deuxième reste bloqué tant que le premier n'a pas libéré l'indicateur. L'interpréteur graphique est donc accédé par un seul consommateur à la fois. En conséquence, les données exploitées par l'interpréteur graphique sont protégées, et ainsi les résultats graphiques devraient être cohérents avec les requêtes du client.

7.3.6 La perte des erreurs d'exécution

Le problème principal de ce traitement en parallèle de plusieurs messages réside dans la perte de la trace d'une instruction : en effet, une instruction n'est plus considérée en tant que tel mais en tant que partie d'une requête. Cela pose donc des problèmes au niveau du retour des erreurs dans le sens serveur→client. Les erreurs

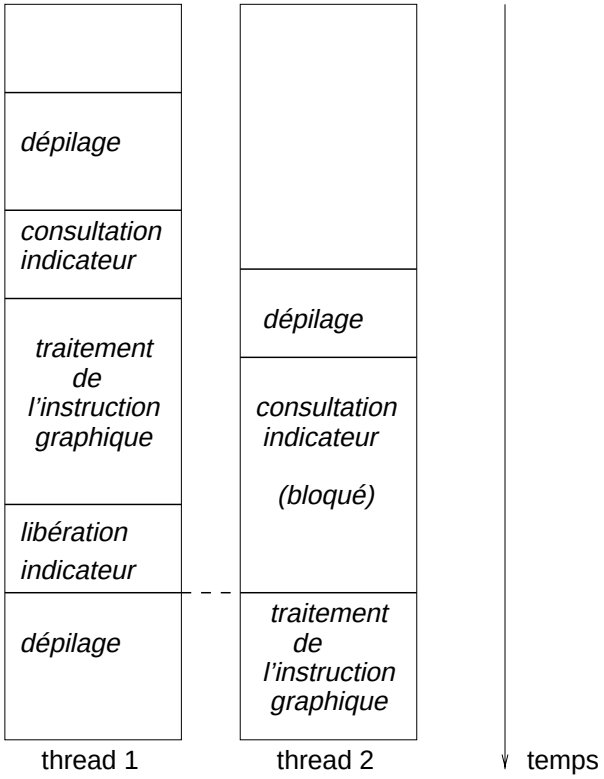


FIG. 7.4: Protection des données

relatives à chaque message sont bien prises en compte par le serveur, mais le fait d'envoyer au client un message d'erreur relatif à celle-ci perd son sens, vu que le client serait notifié d'une erreur à la fin du traitement du message contenant l'instruction illicite. Cela implique en particulier que le client serait dans l'incapacité de prévoir un traitement spécifique à l'instruction fautive pour la prise en compte de cette erreur.

Le traitement des erreurs d'exécution tel qu'il était décrit dans le chapitre 7.2 a donc été modifié : lorsque le serveur détecte une erreur dans une instruction, il n'exécute pas l'instruction en question, mais notifie au client que cette erreur s'est produite pendant l'interprétation de ses requêtes lorsque le serveur a fini d'interpréter le message contenant la requête menant à l'instruction illicite. Le client doit donc prévoir un traitement lorsque de tels cas se produisent.

7.4 Communications

Le serveur est en attente de messages en provenance du réseau. Les messages sont traités de manière *synchrone*, c'est-à-dire que le serveur ne traite les requêtes que d'un seul client à la fois. Cependant, afin que l'ensemble du serveur (en particulier la partie d'exploitation graphique des résultats) ne soit pas bloqué en attente de messages sur le réseau ce qui le rendrait inutilisable pendant ce temps, l'attente d'une connection est gérée par un processus léger différent de celui de l'exploitation graphique.

Du point de vue des communications, le principe de connection est le suivant :

- Attente d'une connection : le décodeur (c'est-à-dire les consommateurs ainsi que le dépaqueur) est dans un état bloqué. Cela signifie que le serveur est en attente d'une connection, et que pendant cette attente l'utilisateur peut manipuler les scènes composées auparavant.
- Un client demande une connection : le processus léger du dépaqueur est débloqué, un *socket* UNIX est créé pour la connection. Son indentificateur est envoyé au client, qui lui-même se connecte sur ce socket. Pendant ce temps, les manipulations des scènes précédentes sont bloquées.
- Le client et le serveur peuvent maintenant communiquer par l'intermédiaire de ce socket. Ils peuvent ainsi échanger des messages, tels que les requêtes de la part du client, mais aussi les résultats d'une requête d'interaction utilisateur que le client a pu émettre, ainsi que des codes d'erreur éventuellement générés.
- Fin de connection : elle peut être requise par le client, lorsque celui-ci a fini d'émettre ses requêtes, ou en cas de notification d'erreur. Le socket créé est détruit, le décodeur bloqué, et l'utilisateur peut maintenant manipuler la scène qui vient d'être composée. De plus, le serveur se remet en attente d'une connection.

L'interface de communication qui a été implémentée est d'assez bas niveau, ce qui peut en assurer la portabilité sur plusieurs types de machines et de systèmes d'exécution.

7.5 Quelques tests

La question qui se pose naturellement est de savoir dans quelle mesure un utilisateur peut espérer profiter de la multi-programmation légère pour améliorer les performances du serveur, en particulier sur une machine mono-processeur. En effet, puisque la partie d'interprétation graphique est effectuée en exclusion mutuelle par les consommateurs du décodeur, le gain en termes de performances n'est pas acquis *a priori*. Le résultat recherché par l'introduction de la multi-programmation légère étant de permettre aux différents consommateurs une occupation efficace de la ressource graphique, nous avons mis au points quelques tests qui permettent de vérifier ce point.

7.5.1 Importance du nombre de consommateurs

Nous avons dans un premier temps testé les performances en faisant varier le nombre de consommateurs, pour deux expériences similaires consistant à tracer 16000 puis 32000 triangles. Les courbes de la figure 7.5 ont été obtenues en répétant cinquante fois le test puis en faisant la moyenne des temps mesurés.

Avant de décrire les paramètres de l'expérience, nous pouvons constater tout d'abord que les deux courbes sont décroissantes, ce qui semble indiquer que l'augmentation du nombre de consommateurs a une influence positive (jusqu'à une valeur seuil) sur les performances générales du serveur.

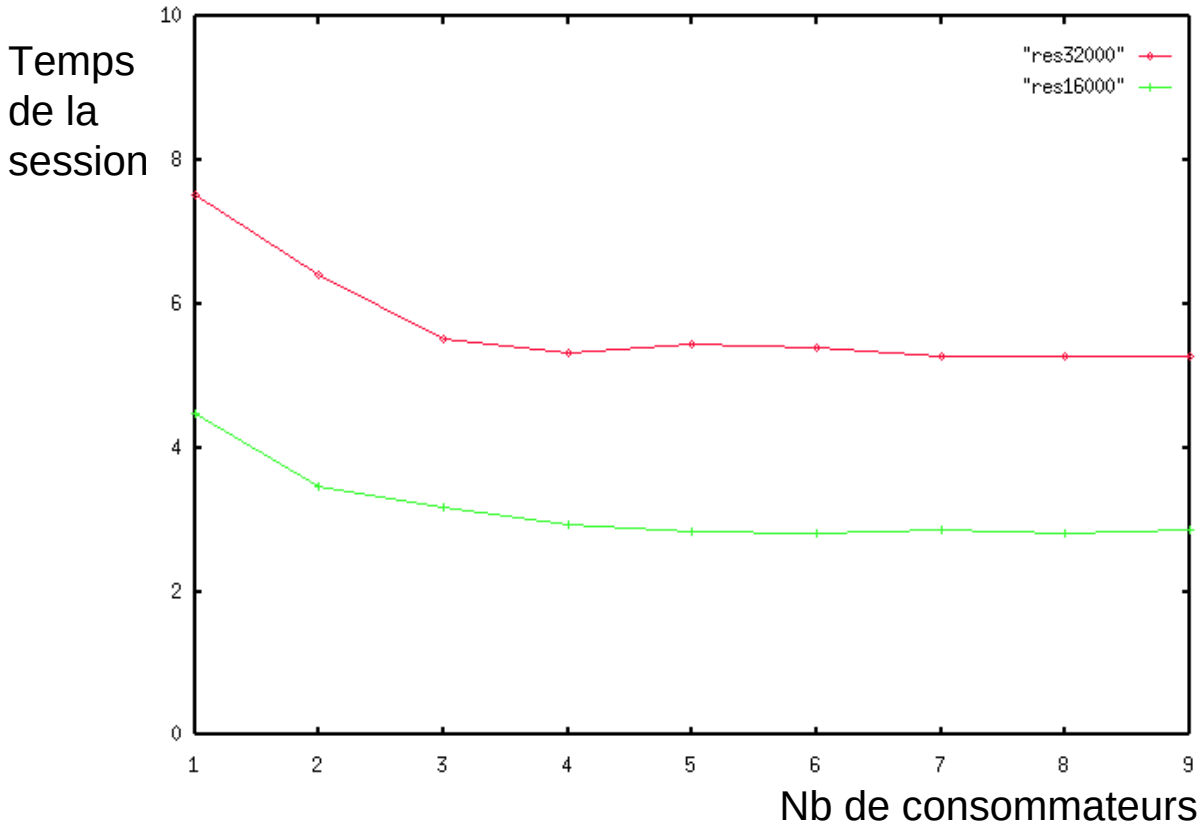


FIG. 7.5: Durée de la session, en fonction du nombre de consommateurs

La machine utilisée pour conduire ce test est un SUN *monoprocesseur*. Nous voyons ainsi que la multi-programmation légère peut apporter un gain significatif (d'environ 25%) sur les performances générales du serveur, même sur des machines ne possédant qu'un seul processeur. De plus, les instructions graphiques interprétées au niveau du serveur sont des tracés de triangles, c'est-à-dire des traitements courts, qui ne nécessitent que peu de manipulations avant d'être interprétés par l'interpréteur graphique. Cela semble indiquer (et nous pourrons le vérifier en 7.5.2) que l'accélération montrée sur la figure 7.5 peut être due à une meilleure utilisation de la ressource graphique. En effet, puisque le nombre de processus légers accédant à l'interpréteur augmente, le nombre d'instructions bloquées en attente d'un traitement augmente lui aussi. Ainsi, puisque le temps de traitement d'une instruction est constant (celui-ci n'a pas été multiprogrammé), *le temps d'attente* entre le traitement de deux instructions consécutives a tendance à diminuer, et donc le temps total de la session à diminuer lui aussi.

Pour ce test, nous avons mesuré le temps total de la session, temps qui comprend aussi bien le temps mis par le serveur pour décoder les messages, mais aussi le temps d'initialisation de la session. Il peut donc être intéressant de mesurer le gain de temps *pendant une GANJ_SubRep* apporté par l'utilisation des processus légers.

7.5.2 Occupation de l'interpréteur graphique

Afin de vérifier que l'utilisation des processus légers permet de mieux gérer les accès à la ressource graphique, les temps passés par l'interpréteur graphique à traiter les instructions qui composent une GANJ_SubRep ont été relevés. Nous obtenons les courbes sur la figure 7.6, obtenues elles aussi en répétant cinquante fois le même test et en faisant la moyenne des temps mesurés.

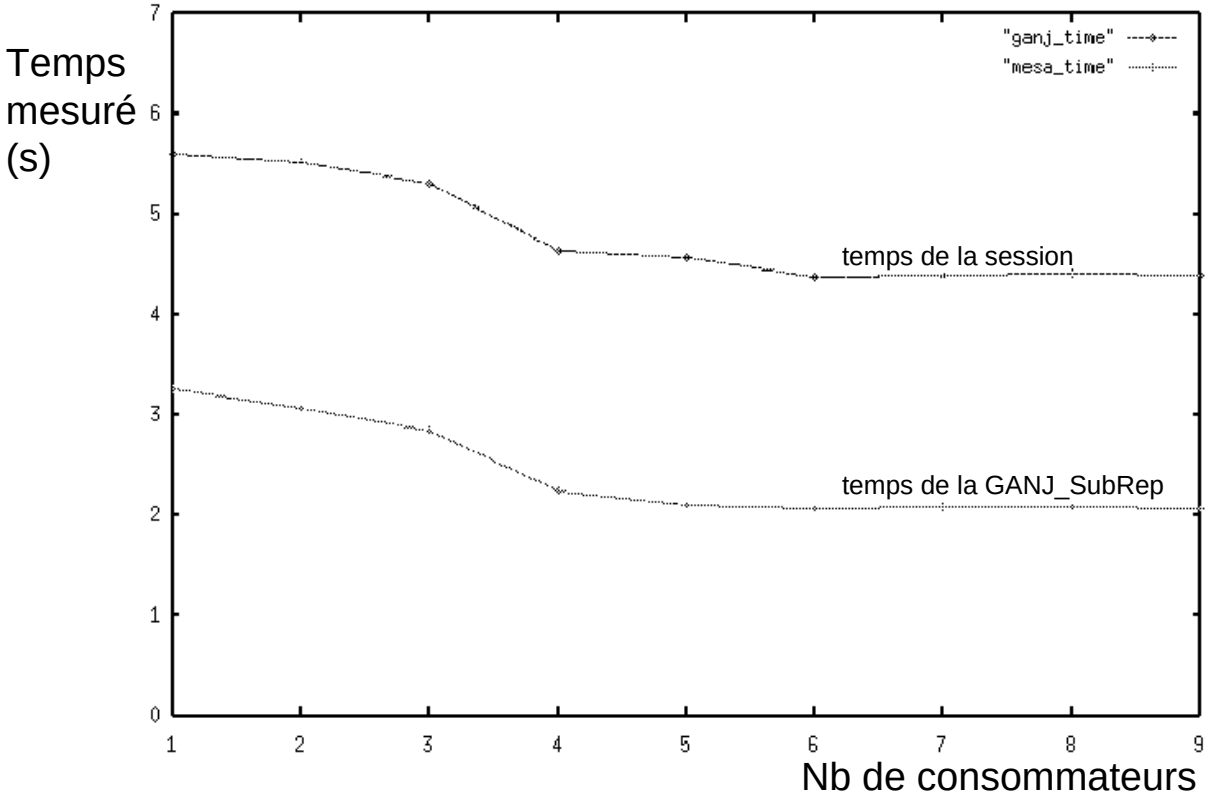


FIG. 7.6: Mesure du temps d'accès à l'interpréteur graphique

Les deux courbes de la figure 7.6 ont une différence à peu près constante, ceci quel que soit le nombre de consommateurs mis en jeu. Cela est rassurant, car la totalité des instructions dont le traitement est parallélisé est contenu dans les GANJ_SubRep, ce qui signifie que le nombre de consommateurs mis en jeu n'a absolument aucune importance sur les instructions *non comprises* dans les GANJ_SubRep, ce qui est vérifié sur la figure.

Le temps mesuré pendant le traitement des instructions comprises dans une GANJ_SubRep est composé du temps passé à les décoder, du temps passé à synchroniser les différentes instructions générées au niveau de l'interpréteur graphique, et du temps passé par celui-ci pour interpréter les instructions graphiques. Rappelons que la librairie graphique utilisée pour interpréter les instructions n'est pas sécurisée pour la multi-programmation légère, en conséquence la troisième catégorie mentionnée ne peut pas être accélérée, car les données passées à l'interpréteur ont été protégées par

un mécanisme d'exclusion mutuelle. L'accélération provient donc des deux premières catégories de temps consommé.

Afin de mettre en évidence les causes de cette accélération, l'expérience utilisée pour le traçage de ces deux courbes a mis en oeuvre deux traitements longs (des tracés de boules avec 1250 triangles) ainsi que 8000 traitements courts (des tracés de triangle). Le premier traitement long est requis en début de GANJ_SubRep, alors que le second l'est en milieu.

L'accélération se décompose en deux phases. La première se situe lorsque le nombre de consommateurs varie de 1 à 3, puis une légère amélioration des performances du serveur se produit lorsque le nombre de consommateurs passe à 4, pour stagner ensuite. La première accélération peut s'expliquer par le fait que lorsque le serveur effectue le traitement du premier message, il met en attente les instructions qui sont émises par le client pendant ce traitement. Il se bloque ensuite lorsque le deuxième message arrive, pendant que le premier message est encore en traitement. Par contre, lorsque le nombre de processus légers vaut 4, un tel blocage ne se produit plus, car les instructions courtes sont mises en attente pendant le traitement des deux instructions longues, et ainsi, lorsque les traitements longs ont fini d'être effectués, l'interpréteur graphique peut interpréter les instructions plus courtes. Cette mise en attente permet d'éviter une situation de saturation de l'interpréteur graphique. Cette ressource est ainsi mieux utilisée lorsque le nombre de consommateurs augmente.

Nous pouvons examiner la figure 7.7 afin de voir le placement dans le temps des différents messages. Cette figure représente une photo-écran d'une application développée par nos soins afin justement de pouvoir réaliser des études fines des placements des messages pendant le temps d'une session. Cette application permet d'exploiter graphiquement les traces générées par le serveur, chaque trace correspondant à un type d'action et à la date d'exécution de cette action.

Les boîtes hachurées sur cette figure représentent le temps de traitement d'un message par un consommateur, c'est-à-dire le temps écoulé entre le moment du débloquage du processus léger qui l'exécute par le dépaqueur et le moment où le processus léger a fini son traitement, et où il se met en état de blocage.

Nous voyons sur la figure 7.7 que le consommateur numéro 1 doit traiter le message long, alors que les autres doivent traiter des messages nettement plus courts. Dans ce cas, le message long est composé d'une instruction chargée dynamiquement, plus quelques dizaines d'instructions plus simples (des tracés de triangles). Ainsi, le traitement d'un tel message reviendra à interpréter l'instruction longue, ce qui prend du temps, puis à interpréter quelques dizaines d'instructions dont le traitement peut être rapide. Le consommateur qui doit effectuer ce traitement occupera donc pendant un temps important l'interpréteur graphique (avec un mécanisme d'exclusion mutuelle), puis le libérera, pour l'occuper ensuite pendant des temps plus courts à chaque fois qu'il devra interpréter une instruction courte.

Pendant que le consommateur numéro 1 n'utilise pas la ressource graphique, les autres consommateurs (qui traitent des messages courts) peuvent l'utiliser. Ainsi, comme montré sur la figure 7.7, ces consommateurs effectuent des traitements en même temps, et donc le dépaqueur peut continuer à accepter des connections, sans

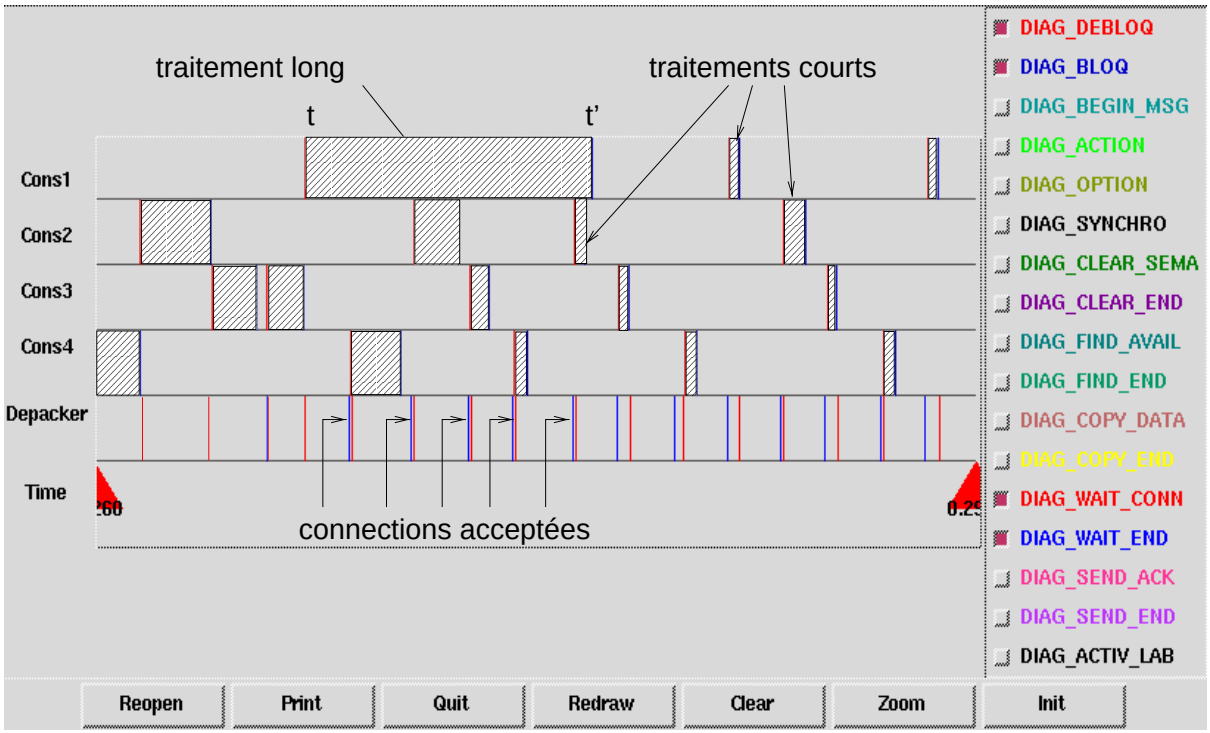


FIG. 7.7: Placement des messages en fonction du temps

que le consommateur ait fini de traiter son message long.

Enfin, en guise de conclusion, l'écart entre les deux courbes sur la figure 7.6 peut amener deux remarques :

- Sur cette expérience, l'utilisation de plusieurs processus légers concurrents ne dégrade pas les performances générales du serveur (cela est dû au faible nombre de processus légers utilisés pour les expériences).
- L'initialisation de la session représente une part importante dans le temps utilisé par la session. Cette portion pourrait être largement réduite par le biais de quelques optimisations, comme par exemple un stockage des instructions qui la composent, ou une optimisation du mécanisme de chargement associé aux instructions complexes, *i.e.* celles qui sont chargées dynamiquement au niveau du serveur.

Chapitre 8

Les clients

Dans le contexte de l'environnement graphique GANJ, le terme de *clients* est un terme générique : il s'agit en fait de n'importe quelle application pouvant se connecter au serveur, et échanger des données avec lui.

Les clients sont ici sous deux formes possibles :

une application : écrite en C ou en C++, ou tout autre langage de programmation pour lequel une interface de programmation a été réalisée. Le programmeur de cette application réalise dans ce cas des appels aux points d'entrée de l'interface de programmation, que ceux-ci soient les primitives géométriques présentes dans la grammaire de GANJ, ou des instructions afin d'effectuer des requêtes de chargement dynamique.

un package d'un système de Calcul Formel : Axiom, Maple. Dans ce cas, l'interface de programmation doit pouvoir tenir compte de la structure des objets internes au système de Calcul Formel. De même que précédemment, un programmeur utilise des fonctions prédéfinies afin de se connecter au serveur, et doit prévoir la conversion des objets de haut niveau qu'il manipule en primitives géométriques élémentaires acceptées en tant qu'instructions graphiques au niveau du serveur. En effet, nous n'avons pas prévu de compiler du code s'exécutant dans les systèmes de Calcul Formel de manière à ce que ces codes puissent être chargés dynamiquement au niveau du serveur.

La principale différence entre ces types de clients tient au fait que la première catégorie de clients est compilée, tandis que la seconde catégorie est généralement interprétée[1]. Ces deux catégories ont des points communs, qui seront détaillés en 8.1. L'interface de programmation en C++ qui a été réalisée pour l'exploitation graphique dans l'environnement GANJ sera ensuite étudiée, puis l'interface de programmation pour Maple sera détaillée. Notons de plus qu'une interface de programmation a été réalisée pour AXIOM, mais nous ne la détaillerons pas ici.

[1] même si certains systèmes de Calcul Formel peuvent générer du code compilé, nous nous en sommes tenus à une version interprétée des interfaces de programmation que nous avons réalisées

8.1 Organisation générale

Une application client est typiquement composée de la partie utilisateur de l'application, *i.e.* le code que l'utilisateur écrit, ainsi que d'appels à des routines permettant d'effectuer les requêtes au niveau du serveur. Ces routines sont contenues dans des librairies lorsque l'application est écrite dans un langage de programmation, ou bien sont sous forme de packages lorsque l'application est un programme Maple ou bien un shell script. Ces routines sont en général de haut niveau, comme par exemple *tracer un polygone* ou alors *récupérer une donnée à la souris*.

Trois types différents de requêtes ont été recensés en 6.6 :

les GANJ_Rep : les requêtes regroupées dans cette catégorie permettent au client d'envoyer au serveur des options qui affecteront le déroulement ultérieur de la session.

les GANJ_SubRep : les requêtes regroupées dans cette catégorie permettent de composer la scène.

les GANJ_UserInput : lorsque le client adresse une requête de cette catégorie au serveur, le client se bloque en attente d'une réponse du serveur. Cette réponse viendra lorsque le serveur aura pu saisir une valeur (spécifiée par l'utilisateur) du type spécifié dans la requête du client.

L'ordonancement de ces requêtes doit être le plus souple possible : en effet, il semble parfaitement justifié que ces différents types de requêtes puissent s'effectuer dans n'importe quel ordre. Par exemple, une requête d'entrée utilisateur peut s'effectuer après un premier tracé, pour que le client recalcule certaines données d'après les paramètres spécifiés par l'utilisateur au vu des résultats du premier calcul. Il s'ensuit que la position de ces types de requêtes dans un programme doit être parfaitement interchangeable. De plus, puisque l'environnement graphique GANJ est basé sur le modèle du Client/Serveur, ce n'est pas la structure *statique* du programme qui est importante, mais bien l'ordre des requêtes pendant l'exécution. Les appels doivent donc, *pendant l'exécution du programme*, se conformer à la grammaire qui a été définie : en effet, ils seront interprétés à ce moment là par le serveur. Si le serveur détecte un erreur (de grammaire, notamment), il le notifie au client qui doit le prendre en compte. Ceci induit deux remarques :

- La détection des différents types d'erreurs définis dans le chapitre 7.2 est importante au niveau du client, puisque le client doit prévoir un comportement à adopter lorsqu'une erreur se produit. Pour prendre un exemple, lorsque la connection entre le serveur et le client est rompue, le client ne peut plus envoyer de requêtes au serveur, et doit donc soit stopper son exécution, soit tenter d'ouvrir une nouvelle connection vers le serveur.
- Le procédé naturel pour la prise en compte des erreurs est bien celui des exceptions, en C++ notamment. Les classes qui permettent à un client de communiquer avec le serveur seront à terme capables de lancer des exceptions lorsque de telles erreurs se produisent.

8.2 Implémentation en C++

Le C++ est le langage naturel pour l'exploitation de l'environnement graphique GANJ: nous évoquions en 8.1 le traitement des erreurs par les exceptions, mais il se trouve aussi que la structure d'une application client est bien adaptée aux langages orientés objets.

8.2.1 Les classes de base

Quatre classes ont été écrites afin de permettre à une application client de communiquer avec le serveur GANJ. Un souci constant pendant la réalisation pratique de ces classes a été de cacher au maximum les détails techniques du modèle GANJ (en particulier, tout ce qui concerne le réseau et la programmation graphique, afin de rendre l'utilisation de ces classes plus faciles à un utilisateur). Les quatre classes réalisées correspondent pour trois d'entre elles aux trois types de requêtes déjà relevés :

La classe GANJ_Rep : cette classe permet de construire la partie basse de la session graphique : elle contient des méthodes pour spécifier des options globales, des entrées utilisateur, ainsi que pour gérer les paramètres de la session.

La classe GANJ_SubRep : cette classe permet de spécifier les instructions graphiques : elle contient les méthodes pour spécifier les options locales, ainsi que les instructions graphiques. Lors de la création d'un objet de cette classe, une paire Begin/End est ouverte au niveau de la couche OpenGL, et elle est fermée lors de sa destruction.

La classe GANJ_UserInput : cette classe permet d'émettre une requête d'entrée utilisateur, et de récupérer le résultat donné par l'utilisateur si l'utilisateur a donné un résultat, et un message particulier si l'utilisateur n'en a pas donné, c'est-à-dire qu'il a interrompu le processus de sélection.

La classe GANJ_App : cette classe permet d'initialiser et de fermer la session graphique, ainsi que d'effectuer les initialisations spécifiques à l'utilisateur par le biais d'une méthode virtuelle. Typiquement, toute application C++ voulant exploiter l'environnement graphique GANJ devra dériver de cette classe, et le programmeur devra écrire sa propre méthode pour l'initialisation de son application (en particulier pour la prise en compte des options en ligne).

Cette dernière classe permet de fournir un format standard d'application à un programmeur utilisant l'environnement graphique GANJ.

8.2.2 Structure d'une application

Une application client sera ainsi basée sur la spécification d'une classe dérivant de la classe `GANJ_App`, comme montré dans la table 8.1.

La classe `My_App` dérive ainsi de la classe `GANJ_App`, et elle surcharge la méthode `GUser_Init` par une méthode permettant d'effectuer les initialisations propres à la classe `My_App`, comme notamment la prise en compte des arguments de la ligne de

```
class My_App : public GANJ_App {

  public :
       void GUser_Init ( int argc, char **argv ) {
       // initialisations spécifiques à l'application
        // ...
       };

       // constructeur :
      My_App ( /* ... */ ) ;

       // autres méthodes
      void Traitement1 ( /* ... */ ) {    /* ... */ } ;
};
```

TAB. 8.1: La classe de base d'une application client

commande. Il ne reste plus qu'à créer un objet de cette classe dans le programme principal, comme montré sur la table 8.2.

La table 8.2, page 143, montre l'usage des différentes classes définies précédemment. Ainsi, l'application construit un objet d'une classe dérivée de `GANJ_App`, et cet objet hérite donc des méthodes d'initialisation, de la connection avec le serveur et de fermeture de cette connection. Par le biais de la méthode virtuelle `GUser_Init`, le programmeur peut donc faire ses propres initialisations (avec la prise en compte des options en ligne propres à son application), puis effectuer le traitement qu'il désire faire par la classe qu'il a réalisé.

Dès lors que le programmeur désire adresser des requêtes graphiques au serveur, il crée un objet de classe `GANJ_Rep`. Lors de la création de cet objet, une référence sur l'objet de classe `GANJ_App` est passée : cela permet d'*attacher* la représentation nouvellement créée à l'application qui est en train de s'exécuter, en particulier cela va permettre à la représentation de récupérer tous les paramètres issus de l'initialisation. Lorsque cette représentation est créée, toutes les opérations permettant de communiquer avec le serveur sont possibles : saisie de paramètres utilisateur, spécification d'options globales, et en particulier (comme sur l'exemple de la table 8.2) la création de sous-représentations qui contiendront les requêtes d'instructions graphiques.

L'exemple 8.2 illustre la simplicité de l'API, en particulier pour l'utilisation des instructions simples, lorsque le langage de programmation est aussi puissant que le C++.

```
int main ( int argc, char **argv ) {
  My_App my_app ;

  my_app.GUser_Init ( argc, argv ) ;

  // autres traitements spécifiques de l'application

  // initialisation de la session
  // permet de récupérer les options du client
  my_app.GInit ( argc , argv ) ;

  // Traitements spécifiques à la classe My_App
  my_app.Traitement1 ( /* ... */ ) ;

  // début des requêtes graphiques
  // on attache une représentation graphique
  // à l'application my_app
  GANJ_Rep Grep ( my_app )  ;

  // quelques requêtes globales
  OK = GRep.Autozoom_Mode ( NO_AUTOZOOM_ ) ;
  // ...

  // ouverture d'une sous représentation
   {
      GANJ_SubRep GSR  ( &GRep , ``dessin'' ) ;

      //trace le segment [(12,12,0),(-12,-12,0)] avec comme couleur
      // celle spécifiée dans l'entrée 14 de la palette du serveur
      GSR.Segment ( 12. , 12. , 0. , 14 , -12. , -12. , 0. , 14 ) ;

   } // la sous-représentation est détruite ici,
     // un ``END'' est envoyé au serveur.

   return my_app.GClose() ; // fermeture de la session
}
```

TAB. 8.2: Un exemple de code d'une application cliente

8.2.3 Instructions complexes

Considérons une application qui doit tracer un certain nombre d'objets de haut niveau, comme par exemple des sphères. L'utilisation de l'environnement graphique

GANJ pour effectuer ce genre de tracés peut se faire de deux manières :

- la première consiste en la décomposition de chaque sphère en triangles, puis en communiquant par le réseau douze valeurs (quatre valeurs par sommet), ceci pour tous les sommets composant la triangularisation des sphères. Cette solution est satisfaisante dans un premier temps, pour avoir une idée rapide de la scène à représenter, mais peut présenter un coût important, lorsque plusieurs sphères sont à tracer dans une même scène notamment.
- la seconde manière consiste en l'utilisation des *instructions complexes* évoqués rapidement en 7.1, page 123. Ces instructions fournissent des facilités pour effectuer du chargement dynamique de code au niveau du serveur.

Considérons que l'utilisateur qui a écrit l'application client sache programmer en C, et qu'il ait écrit une fonction qui triangularise une sphère, et réalise les appels OpenGL pour la tracer. Supposons de plus que cette fonction ait été compilée, et que le code résultant soit stocké dans une librairie partagée. Les systèmes d'exploitation les plus courants (Solaris, Linux, mais aussi les dernières versions d'AIX) permettent de charger dynamiquement du code, c'est-à-dire pendant l'exécution d'un programme.

Lorsque le serveur a été compilé pour prendre en compte ces chargements dynamiques, le client peut effectuer les requêtes permettant de charger et d'exécuter des actions précompilées, comme par exemple le tracé d'une sphère.

Nous allons examiner le code d'un exemple permettant de tracer une boule en utilisant ce mécanisme. Ce code (table 8.3, page 145) est basé sur la classe `My_App` décrite précédemment. Après les initialisations d'usage, le client fait une requête `DL()`. Cette requête permet au client de s'assurer que le serveur est bien capable de gérer[2] les futures requêtes de chargement qui seront exécutées par la suite.

Ensuite, le client crée un objet de classe `GANJ_DL`, avec comme paramètre le nom de la librairie[3] dans laquelle se trouve le code à charger. Cet objet est alors passé à la requête `NeedLib`, afin que le serveur charge cette librairie. Lorsque le serveur ne renvoie pas d'erreur, le client peut alors déclarer selon le même principe la fonction à charger dans cette librairie (ici, il s'agit de la fonction `BouleFonction`), fonction qui est déclarée comme étant une action, c'est-à-dire un élément d'une `GANJ_SubRep`, avec quatre arguments.

Lors de l'appel de cette instruction au niveau du serveur, le code de la fonction correspondant à l'instruction est exécuté, avec les arguments passés par le client lors de l'appel.

Nous pouvons faire quelques remarques sur ce mécanisme :

- Le passage *a priori* du type de l'instruction qui sera exécutée par la suite est intéressant car il permet au serveur de vérifier la correction grammaticale de

[2]précaution d'autant plus importante que la version du serveur s'exécutant sous AIX en version inférieure à 4.x ne le permet pas, contrairement aux versions s'exécutant sous Solaris ou Linux.

[3]cette librairie doit être accédée par le serveur, et donc en particulier doit se trouver sur un disque visible depuis le répertoire du serveur, ou dans un répertoire permettant à l'éditeur dynamique de liens de la retrouver.

```
int main ( int argc , char **argv ) {
 My_App my_app ;

 my_app.GUser_Init ( argc , argv ) ;
 my_app.GInit ( argc , argv ) ;

 GANJ_Rep GRep ( my_app ) ;

 if ( GRep.DL () < 0  ) {
   cerr << ''DLs not available'' << endl ;
   throw ;
 }


 GANJ_DL lib ( ''libboule.so'' ) ;
 GRep.NeedLib ( lib ) ;

 GANJ_DL_Func Boule ( lib , ''BouleFonction'' , 4 , GANJ_ACTION ) ;

 double args[4] ;
 args[0] = 1. ;
 args[1] = 0. ;
 args[2] = 0. ;
 args[3] = 0. ;

 {
  GANJ_SubRep GSR ( &GRep , ''boule'' ) ;
  GSR.Packed ( 1 ) ;

  GSR.Call_Action ( Boule , args ) ;

 }

 GRep.Close () ;
}
```

TAB. 8.3: Un exemple de chargement dynamique

l'appel ainsi réalisé, et donc en cas d'erreur à ce niveau de ne pas exécuter l'instruction qui pourrait provoquer des erreurs plus graves, au niveau d'OpenGL notamment.

– Le prototype de la fonction `BouleFonction` est standard :

```
int BouleFonction ( int nargs , double *args ) ;
```

Prototype de la fonction BouleFonction

Ce prototypage standard permet au serveur, lorsqu'il charge l'instruction, d'empiler les arguments de l'instruction et d'en récupérer le résultat (le code des erreurs éventuellement générées).

Ce mécanisme est très utile lorsque l'utilisateur sait caractériser l'objet à tracer par la donnée de quelques paramètres, car il peut faire économiser beaucoup de temps sur les transferts par réseau. De plus, il permet à un ensemble d'utilisateurs de partager les fonctionalités qu'ils ont écrites, sans pour cela les intégrer dans la grammaire.

Ce mécanisme a permis d'écrire une librairie permettant de gérer les lumières à l'intérieur des scènes, sur la base de la gestion des lumières en OpenGL.

8.3 Les packages : implémentations dans les systèmes de Calcul Formel

Un *package* dans un système de Calcul Formel est un regroupement logique de fonctionnalités, permettant de développer des applications qui utilisent les méthodes du package. Le problème des packages vient essentiellement de la perméabilité des systèmes de Calcul Formel comme Maple avec le monde extérieur, et cela implique en particulier des développements spécifiques pour chaque système de Calcul Formel.

8.3.1 Architecture générale

L'organisation générale d'un package est décrite sur la figure 8.1 ; ce package, destiné à une exploitation sous Maple, permet de générer des instructions simples.

Cette figure est composée de trois parties bien distinctes : la partie I est celle correspondant aux applications s'exécutant dans l'environnement Maple. La partie II correspond aux applications s'exécutant dans le shell à partir duquel a été lancée la session Maple, alors que la partie III concerne le serveur Ganj.

Cette architecture — compliquée au demeurant — est motivée par les contraintes liées à l'interfaçage Maple/Unix. En effet, Maple et Unix communiquent très mal, bien que la dernière version de Maple (V, release 4) facilite les transferts d'information entre des applications Unix et Maple.

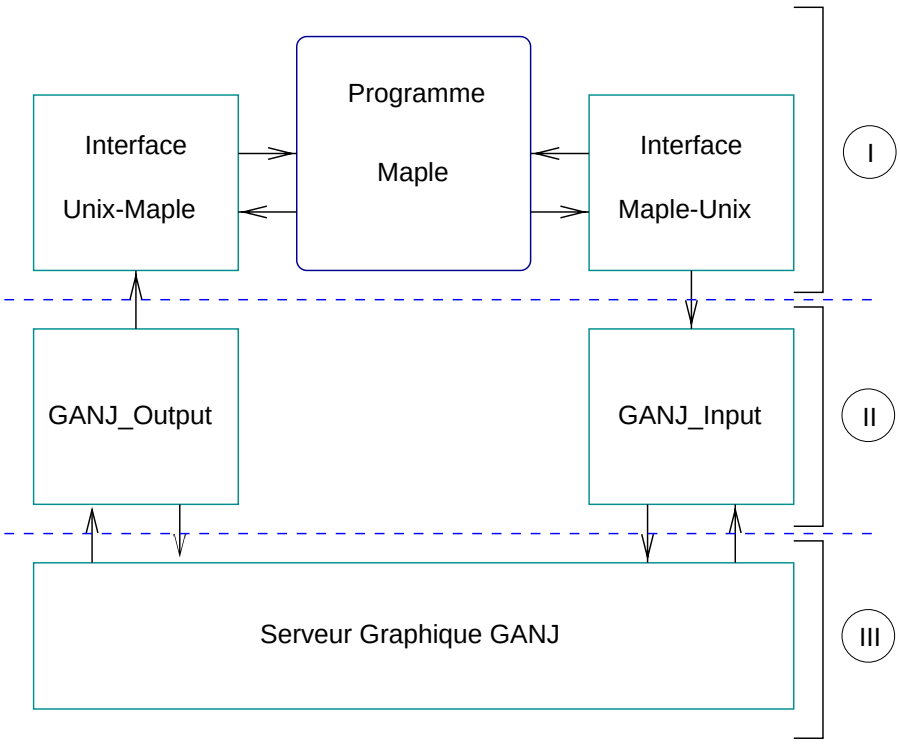


FIG. 8.1: Organisation du couplage Maple/GANJ

8.3.2 GANJ_Input et GANJ_Output

Ces deux applications s'exécutent au niveau du shell, et sont lancées par des programmes Maple grâce à des *pipes* : les pipes permettent de connecter deux applications par l'intermédiaire d'un flot d'entrée–sortie, c'est-à-dire qu'une des deux applications prend en entrée standard tout ce qui est produit en sortie standard de l'autre application.

Ainsi, le programme GANJ_Input attend des instructions en entrée standard, puis les encode en requêtes qu'il va adresser au serveur GANJ. C'est au niveau de cette application que sont gérées les erreurs, ainsi que les connections au réseau.

De même, le programme GANJ_Output se connecte au serveur GANJ, puis lui envoie une requête d'entrée utilisateur (requête dont les paramètres sont passés par Maple lors du lancement du programme GANJ_Output, c'est-à-dire à la création du pipe), et lorsqu'il reçoit la réponse du serveur (soit une liste de valeurs, soit une valeur signifiant l'annulation par l'utilisateur de la requête), la transmet par l'intermédiaire de sa sortie standard vers le programme Maple.

Ces deux types d'actions ont été dédoublées à cause de la pauvreté de l'interface Maple/Unix : il est impossible depuis Maple de créer un double-pipe sur la même application shell.

8.3.3 Interfaçage Unix/Maple

Le point précédent montre qu'un programme s'exécutant dans une session Maple peut communiquer avec le serveur GANJ par l'intermédiaire des deux outils GANJ_Input et GANJ_Output. L'interface entre Maple et ces deux programmes est réalisée par l'intermédiaire de *double pipes*, symbolisés sur la figure 8.1 par des doubles flèches. Cette facilité offerte par Maple permet à deux applications s'exécutant dans la session Maple en cours de communiquer par entrées/sorties. En particulier, lors de l'appel d'un fork() sous Maple, les descripteurs de fichier (*file descriptors*) associés peuvent permettre aux deux branches d'exécution de communiquer. Considérons l'exemple 8.4 écrit en Maple.

```
my_fork := proc ()

  # ouverture d'un pipe interne à Maple
  fd := process[pipe] () ;
  # fd est un tableau de file descriptors
  # fd[1] est associé à la lecture, fd[2] à l'écriture

  pid := process[fork] () ;

  # fork renvoie 0 dans le nouveau ``processus'' crée
  # et le pid du ``processus'' créé dans le ``processus'' père
  if pid =0 then

      #ecrit dans le fd[2] a l'attention du ``processus'' père
      writeline ( fd[2] , `Yes Master ?` ) ;
      quit
  else #processus père

  printf (  ` PARENT read %s  \n ` , readline ( fd[1] ) ) ;

  process[wait] () ;

  process[pclose] (fd[1]) ;
  process[pclose] (fd[2]) ;
  fi ;

end ;
```

TAB. 8.4: Un exemple simple de fork() en Maple

Sur cet exemple, l'exécution principale du programme est séparée en deux branches. L'environnement d'exécution du programme est copié à l'appel de `fork()`, et les deux branches s'exécutent chacune dans leur propre environnement. Les variables affectées après le `fork()` ne sont modifiées que dans leur propre environnement d'exécution. Nous voyons ainsi que le processus héritant de l'environnement initial d'exécution (le processus "père") et le processus héritant de la copie de l'environnement initial d'exécution (le processus "fils") s'exécutent de manière concurrente. Il n'est pas très intéressant pour une application client d'attendre que le serveur soit prêt à accepter sa requête, ni tout simplement de se mettre en attente sur le réseau. C'est pour cette raison que l'exécution est divisée en deux branches concurrentes, car cela permet à l'application client de continuer à faire ses calculs pendant que l'autre branche d'exécution effectue les connections au serveur, puis compose les requêtes à adresser au serveur. Le corps de la procédure Maple `GANJ_init_input`, sur la table 8.5, page 150, permet d'étudier ce principe.

Cette procédure sépare l'exécution du programme Maple en deux, et sur la branche principale d'exécution elle ne fait que renvoyer le pid (*process identificator*) de la branche secondaire d'exécution. Par contre, la branche secondaire ouvre un pipe sur le programme GANJ_input, avec comme paramètre le nom de la machine sur laquelle le serveur GANJ est lancé. Ensuite, ce processus va se mettre en attente de messages sur le *file descriptor* `input_fd`, messages à transmettre au serveur par l'intermédiaire du *file descriptor* `fd`, jusqu'à ce que la fin de la transmission soit détectée, auquel cas la connection au serveur sera fermée.

La table 8.6, page 151 montre un exemple typique de session Maple. Cet exemple permet de tracer un triangle spécifié depuis Maple. Nous voyons pour cela qu'il suffit d'appeler la procédure d'initialisation décrite dans la table 8.5, puis, une fois que la connection est effectuée, d'empiler les primitives graphiques à faire parvenir au serveur.

Cette solution présente l'avantage de ne pas obliger le client Maple à se mettre en attente que le serveur lui accorde une connection. En particulier, cela signifie que le client Maple pourra continuer à mener à bien son calcul, en empilant des requêtes graphiques comme dans la procédure présentée sur la table 8.6.

8.3.4 Exemple

L'exemple 8.7 permet de récupérer une *plot structure* interne à Maple, puis de la tracer sur le serveur GANJ à partir de la session Maple. Une *plot structure* est le résultat d'un appel à une primitive de tracé, comme par exemple `plot3d`.

Les avantages inhérents aux primitives graphiques de Maple sont multiples, le plus important étant probablement celui de la simplicité d'utilisation. Par exemple, Maple sait comment tracer la plupart des objets qu'il permet de manipuler, et par conséquent l'utilisateur n'a pas à programmer de mode particulier de visualisation des objets qu'il manipule.

```
GANJ_init_input := proc (  input_fd , hostname )
 local fd, pid ;

  pid := process[fork] () ;

  if pid = 0 then

     fd = popen ( `ganj_input  -stdin -s `.hostname  ,WRITE ) ;
     string := readline ( input_fd ) ;
     pas_fini := Is_Finished ( string ) ;


     while pas_fini do

          writeline ( fd , string ) ;

          string := readline ( input_fd ) ;
          pas_fini := Is_Finished ( string ) ;
     od ;

     process[pclose] ( fd ) ;
     quit

  else

     res := pid ;
 fi ;


 res ;
end;
```

TAB. 8.5: La procédure GANJ_init_input

```
exemple := proc ()
 tri := array [0..11]  ;

 fd := process[pipe] () ;
 pid := GANJ_init_input ( fd[1] , `ponant.imag.fr` ) ;

 #compose un triangle par ses sommets
 # (x,y,z,couleur)
 tri[0] := 1. ; tri[1] :=  0 ; tri[2] := .5 ; tri[3] := 40 ;
 tri[4] := 0. ; tri[5] := -1. ; tri[6] := 0. ; tri[7] := 64. ;
 tri[8] := -1. ; tri[9] :=  0. ; tri[10] := .5 ; tri[11] := 87 ;

 GANJ_SubRep ( fd[2] , `triangle`) ;

 GANJ_Triangle ( fd[2] , tri ) ;

 GANJ_EndSubRep ( fd[2] )  ;

 #ferme la session
 GANJ_Close ( fd[2] ) ;

end;
```

TAB. 8.6: Un exemple de session

Par contre, un des principaux désavantage de Maple au niveau graphique se situe au niveau des performances. Si nous voulons tracer la surface de la fonction

$$f(x,y) = x^2 + y^2, x = \Leftrightarrow 1..1, y = \Leftrightarrow 1..1,$$

Maple va mailler le pavé $[\Leftrightarrow 1, 1] \times [\Leftrightarrow 1, 1]$ par une grille régulière de 25×25 par défaut, et tracer les 625 polygônes dans une fenêtre sur l'écran. Pour réaliser cette opération, il prend subjectivement (sur un SUN Sparc 20, 128 Mo de mémoire) à peu près 5 secondes. Le problème se situe dans la manipulation de la scène tri-dimensionnelle ainsi créée, car Maple va prendre 5 secondes par la suite pour retracer la surface, par exemple après une rotation. Ainsi, pour avoir une vue rapide de la surface à tracer, les primitives graphiques de Maple peuvent suffire, mais dès qu'il est question de manipuler la scène créée, les performances sont de loin insuffisantes pour une utilisation courante (c'est-à-dire avec une grille de taille largement plus important que 625 polygônes).

Pour cela, la primitive `ganj_plot` a été définie : cette primitives reconnaît le type de la surface à tracer et, au lieu de tracer les polygônes dans une fenêtre Maple, les envoie au serveur GANJ. Nous pouvons espérer ainsi de meilleures performances dans une utilisation typique de manipulation de scène tri-dimensionnelle.

Le corps de cette procédure est décrit sur la table 8.7, page 153. Le paramètre formel `pl` est une *plot structure*, c'est-à-dire le résultat d'un appel à `plot3d`.

Les points à tracer se trouvent spécifiés par Maple dans la *plot structure*, et ainsi sont accessibles dans une liste de polygones. Les routines `meshpoly` et `gridpoly` parcourent ces listes, et envoient les composantes au serveur GANJ par l'intermédiaire du *file descriptor* récupéré par l'appel à `process[pipe] ()`.

D'un point de vue performances, cette solution a un temps d'initialisation plus long que le `plot3d` de Maple, car il faut parcourir les listes de polygônes, composer des polygônes à partir des coordonnées spécifiées dans ces listes, et ensuite les envoyer au serveur.

Le programme 8.8, page 154 trace deux surfaces avec des appels à la routine `ganj_plot`, et renvoie les temps (CPU) mis pour réaliser les appels. De plus, nous avons noté (approximativement, car Maple ne permet pas de mesurer le temps CPU utilisé pour effectuer un tracé) le temps mis pour faire un rafraîchissement, soit par Maple, soit par le serveur. Les résultats sont visibles sur le tableau 8.9, page 154.

Sur cette table, nous avons noté les temps des rafraîchissements, car ils sont de loin les plus importants pour un utilisateur qui attend le résultat de son tracé. Nous avons par contre mesuré les temps CPU pour le traitement des polygônes.

Le temps d'initialisation de la routine `ganj_plot` est effectivement non-négligeable, voire même pénalisant, surtout si l'utilisateur n'a que peu de manipulations à effectuer (rappelons qu'à chaque manipulation, un rafraîchissement se produit). Par contre, 'économie de temps de rafraîchissement obtenue en utilisant la routine `ganj_plot` peut être intéressante dès lors que l'utilisateur doit manipuler sa scène dans l'espace, car le serveur offre des performances largement supérieures à l'outil graphique de Maple, en termes de manipulations de scènes à trois dimensions.

```
ganj_plot := proc (  pl ,  hostname, sess_name  )
  local fds, i ;

  fds := process[pipe] () ;

 GANJ_init_input ( fds[1] , hostname) ;

  GANJ_SubRep ( fds[2] , sess_name  ) ;

   #reconnaît le type de tracé à effectuer
   if op(0,op(1,pl)) = MESH then

    meshpoly ( fds[2] , pl ) ;

  else

   if op(0,op(1,pl)) = GRID then
    gridpoly ( fds[2], pl ) ;
  fi;

 fi;

 GANJ_EndSubRep ( fds[2] ) ;

 GANJ_Close ( fds[2] ) ;

 process[wait] () ;
 process[pclose] (fds[1]);
 process[pclose] (fds[2]) ;

end ;
```

TAB. 8.7: La procédure ganj_plot()

```
starttime := proc  ( m , n )

plm := plot3d ( [x , s^3 , 3*s^2 ] , x=-1..1 , s=-1..1, grid =[m, n] ) ;
plg := plot3d ( y^2+x^2 , x=-1..1 , y=-1..1 ) ;

time_mesh := time () ;
ganj_plot ( plm , abel , pl1 ) :
time_mesh := time () - time_mesh ;

time_grid := time () ;
ganj_plot ( plg , abel , pl1 ) :
time_grid := time () - time_grid ;

[time_mesh , time_grid ] ;
end;

startime () ;
```

TAB. 8.8: Un programme de mesure de temps CPU

$n \times m$	time_mesh	time_grid	refresh Maple	refresh GANJ
25×25	8.830	8.610	$\sim 5s$	$< 1s$
30×30	11.570	14.060	$\sim 7s$	$< 1s$
40×40	22.680	22410	$\sim 10s$	$\sim 1s$

TAB. 8.9: Performances comparées Maple / GANJ

Quatrième partie

Application : visualisation de solutions d'équations différentielles complexes

Chapitre 9

Structure d'une expérience

9.1 Principe d'une expérience

Les expériences présentées dans cette partie applications permettent d'étudier des équations ou des systèmes différentiels dans le plan complexe. Cette étude peut se décomposer en quatre parties, représentées sur la figure 9.1 par quatre boîtes.

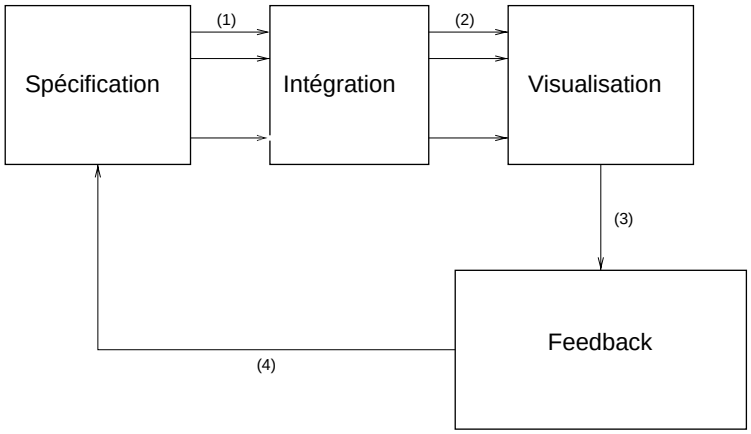


FIG. 9.1: Le déroulement d'une expérience

- Spécification : cette partie comprend aussi bien la spécification du chemin d'intégration, que des ensembles de conditions initiales, mais aussi des différents paramètres inhérents à l'intégration, telle que l'équation ou le système à étudier, les tolérances pour les calculs d'erreur, les pas d'intégration et leur nombre maximal.
- Calcul (intégration) : cette partie peut être effectuée par un ou plusieurs modules de calcul. Ceux-ci prennent en entrée un certain nombre de paramètres (les points dans le plan complexe entre lesquels réaliser l'intégration, une condition initiale, des paramètres de contrôle), et renvoient un ensemble de valeurs, le résultat de l'intégration.

- Visualisation : cette partie correspond à l'exploitation graphique des résultats renvoyés par l'intégrateur tout au long de l'expérience. Cette exploitation peut être locale, c'est-à-dire qu'elle ne concerne qu'un certain type restreint de résultats, comme par exemple les valeurs renvoyées par l'intégrateur pendant le calcul, mais aussi globale, c'est-à-dire qu'elle peut mettre en relation plusieurs grandeurs mathématiques calculées pendant le déroulement de l'expérience.
- Feedback : le feedback est le résultat d'une analyse de l'utilisateur des résultats produits par l'expérience, et conduit en général à de nouveaux calculs, après avoir respécifié certains paramètres initiaux de l'expérience, ou à une validation des résultats.

Les flèches représentées sur la figure 9.1 connectent les quatres boîtes du dessin, et représentent chacunes des mouvements de données :

1. Ces données sont les données spécifiées par l'utilisateur, et qui sont exploitées par l'intégrateur. Elles peuvent être éventuellement partagés entre plusieurs outils de calcul.
2. Ces données sont les résultats de l'intégration. Ces résultats sont stockés, avant d'être exploités.
3. Ces données représentent l'information que l'utilisateur tire de la visualisation. Cette information est bien évidemment liée au mode de visualisation utilisé pour exploiter graphiquement les résultats de l'intégration.
4. Ces données proviennent de l'utilisateur, afin de recalculer certaines valeurs en fonction des conclusions qu'il a tirées de la visualisation des résultats. Elles sont transmises par les moyens d'interaction auxquels l'utilisateur a accès, que ceux-ci soient directement intégrés à la visualisation (principe des `GANJ_UserInput`), ou que ceux-ci soient spécifiques à l'expérience (c'est-à-dire qu'ils n'utilisent pas les facilités de la visualisation, comme par exemple un procédé de sélection spécifique au module de spécification ; par exemple, un module de saisie de courbes).

La plupart des données peuvent être stockées dans une structure de données que nous allons présenter, et qui permet d'avoir une représentation logicielle des chemins.

9.2 Les chemins

9.2.1 Motivations

Un chemin sur un ensemble $\mathcal{E}$ est la donnée d'une application *continue* de l'intervalle réel $[0, 1]$ dans cet ensemble. D'un point de vue expérimental, ils peuvent avoir des fonctions multiples :

- Ils servent pour la spécification mathématique du problème : en effet, le chemin en tant qu'objet mathématique occupe une position centrale en analyse complexe. Par exemple, les résultats d'une intégration dans le plan complexe

dépendent fortement de la spécification des chemins d'intégration, c'est-à-dire des extrémités des chemins mais aussi de la paramétrisation des chemins, à extrémités fixées.

- Ils servent aussi à calculer les solutions des équations ou des systèmes différentiels : en effet, les intégrations numériques donnent les valeurs de la solution en certains points, ces points constituant des chemins. Ces chemins sont très proches des chemins d'intégration, car ils sont l'image des chemins d'intégration par l'approximation de la solution calculée par l'intégration, et donc possèdent le même échantillonnage.
- Ils peuvent aussi servir à la visualisation des résultats issus d'intégrations dans le plan complexe : les éléments de base des chemins résultats peuvent être de nature variable (par exemple, les éléments de base peuvent être pris sur des surfaces de Riemann), et donc peuvent nécessiter des traitements graphiques particuliers afin d'en exploiter les spécificités.

Pour répondre à ces différents types d'utilisation, le plus grand soin doit être apporté à l'implémentation logicielle des chemins, en particulier au niveau de la généricité : si nous proposons une représentation la plus générale possible des chemins, nous aurons moins de travail spécifique à accomplir par la suite.

D'une manière pratique, une représentation logicielle de chemins doit pouvoir permettre certaines opérations :

- la possibilité de pouvoir calculer avec les éléments de base des chemins, mais aussi avec les chemins eux-mêmes (manipulations formelles).
- la possibilité de donner une visualisation adaptée au chemin lui-même, pour les différents types de chemins que nous utiliserons.

9.2.2 Différents types de chemins : cadre expérimental

Les chemins considérés sont typiquement des chemins dont les éléments de base sont des éléments de surfaces de Riemann (que ce soit dans $\mathbb{C}$ pour la spécification, les calculs ou la visualisation ; ou dans des surfaces de Riemann plus générales, notamment lorsque nous cherchons à calculer des déterminations continues). Étant donné un ensemble $\mathcal{E}$ ses éléments peuvent servir d'éléments de base pour des chemins si nous pouvons construire deux applications, μ et $\mathcal{R}$:

- l'application μ de $\mathcal{E}$ dans $\mathbb{C}$, qui donne la projection dans le plan complexe d'un élément de $\mathcal{E}$.
- l'application $\mathcal{R}$ de $\mathcal{E}$ dans $\mathbb{R}^N$ ($N = 2, 3, 4$) qui permet de visualiser les éléments de $\mathcal{E}$.

Si ces deux applications sont connues pour un ensemble $\mathcal{E}$ donné, la définition d'une algorithmique répondant à nos besoins sur les chemins est possible. En effet, le chemin pouvant être projeté dans $\mathbb{C}$ à tout moment, des calculs avec les éléments projetés sont possibles (et ainsi les algorithmes classiques fonctionnant avec des nombres complexes peuvent être utilisés), ainsi que la visualisation du chemin par le biais de

l'application $\mathcal{R}$. De plus, les opérations usuelles sur les chemins ne dépendant pas de la nature de l'ensemble $\mathcal{E}$ (par exemple, des concaténations) seront assumées par l'implémentation générique décrite ici.

9.2.3 Lien avec les surfaces de Riemann

Dans le cas particulier où $\mathcal{E}$ est une surface de Riemann, la définition 1.2.3.2 permet de construire facilement l'application μ. En effet, cette définition assure, pour un élément de $\mathcal{E}$ quelconque, de l'existence d'un ouvert U_i contenant cet élément et d'un homéomorphisme Φ_i permettant de projeter cet élément sur $\mathbb{C}$ (identifié ici à U_0) :

$$\forall x \in \mathcal{E}, \exists i \in I, x \in U_i$$

Nous pouvons alors poser

$$\mu(x) = \Phi_i(x)$$

Ainsi, l'application μ est définie pour toute surface de Riemann, de manière à ce que μ bénéficie des propriétés d'analycité de la fonction de raccordement entre deux ouverts ayant une intersection non vide (la troisième propriété dans la définition 1.2.3.2). Cette propriété est très importante, car elle assure que les projections des chemins obtenues par l'application μ sont bien cohérentes avec la définition de la surface de Riemann, et permet en particulier, si la position sur la surface d'une extrémité du chemin est connue, de retrouver le chemin original à partir de sa projection sur le plan complexe.

Une surface de Riemann, en tant que variété mono-dimensionnelle sur $\mathbb{C}$ peut être plongée dans $\mathbb{R}^3$[Spr57]. En effet, une variété complexe mono-dimensionnelle correspond implicitement à une variété réelle bi-dimensionnelle, et donc une surface dans $\mathbb{R}^3$. Par contre, rien n'assure que cette surface soit simple, c'est-à-dire sans points multiples, ni facile à visualiser. Cette application de plongement peut être néanmoins choisie comme application $\mathcal{R}$, afin notamment de visualiser les éléments des chemins. Par contre, si la surface de Riemann doit être visualisée dans sa globalité, un travail spécifique devra être apporté sur la visualisation de la surface afin d'en obtenir une vision synthétique.

9.2.4 Exemples

Dans les deux exemples suivants, les applications μ et $\mathcal{R}$ sont construites sur des surfaces de Riemann compactes : la sphère de Riemann et le tore complexe. Le cas de la surface de Riemann du logarithme (qui n'est pas compacte) sera ensuite étudié.

La sphère de Riemann

Considérons le plan complexe, à qui nous rajoutons le point à l'infini. D'un point de vue topologique, cet ensemble peut être vu comme une sphère. De plus, cet en-

semble a une structure de surface de Riemann, lorsque les projections stéréographiques par rapport aux deux sommets de la sphère sont définies (figure 9.2) :

$$\begin{array}{rcccc} \pi_N & : & S_1 \Leftrightarrow \{N\} & \rightarrow & \mathbb{C} \\ \pi_S & : & S_1 \Leftrightarrow \{S\} & \rightarrow & \mathbb{C} \end{array}$$

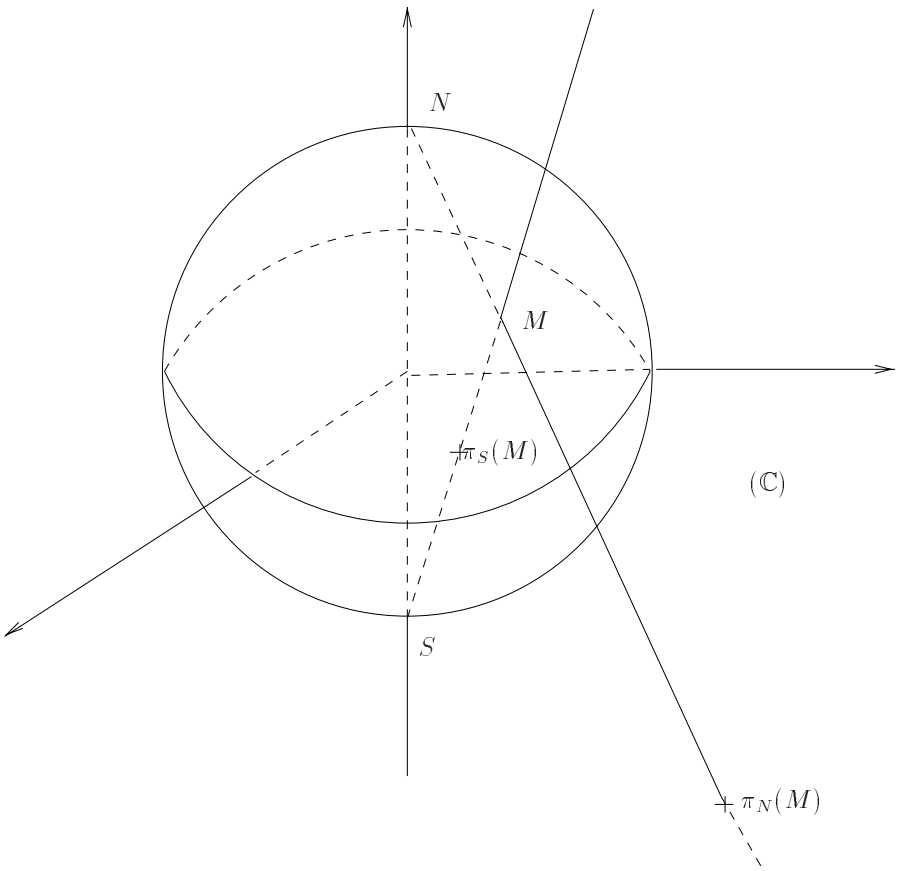


FIG. 9.2: La projection stéréographique

Si le sommet N de la sphère correspond au point à l'infini du plan complexe, l'application μ peut être choisie comme

$$\forall x \in S_1,\ \mu(x) = \begin{cases} \pi_N(x) & \text{si} \quad x \in S_1 \Leftrightarrow \{N\} \\ \pi_S(x) & \text{si} \quad x \in S_1 \Leftrightarrow \{S\} \end{cases}$$

En toute généralité, la première projection π_N sera choisie comme application μ, sauf lorsqu'interviendra le point à l'infini, auquel cas la seconde projection, π_S, sera utilisée pour l'évaluation de μ.

L'application $\mathcal{R}$ peut être choisie comme l'application définissant une sphère dans $\mathbb{R}^3$, à partir d'un point du plan. Cette application peut être prise comme l'inverse de la projection π_N, où les points de $\mathbb{C}$ de module arbitrairement grand sont identifiés au point à l'infini.

Le tore complexe

Le tore complexe est topologiquement équivalent à la donnée de deux sphères : il suffit pour cela de le "couper en deux", comme sur la figure 9.3.

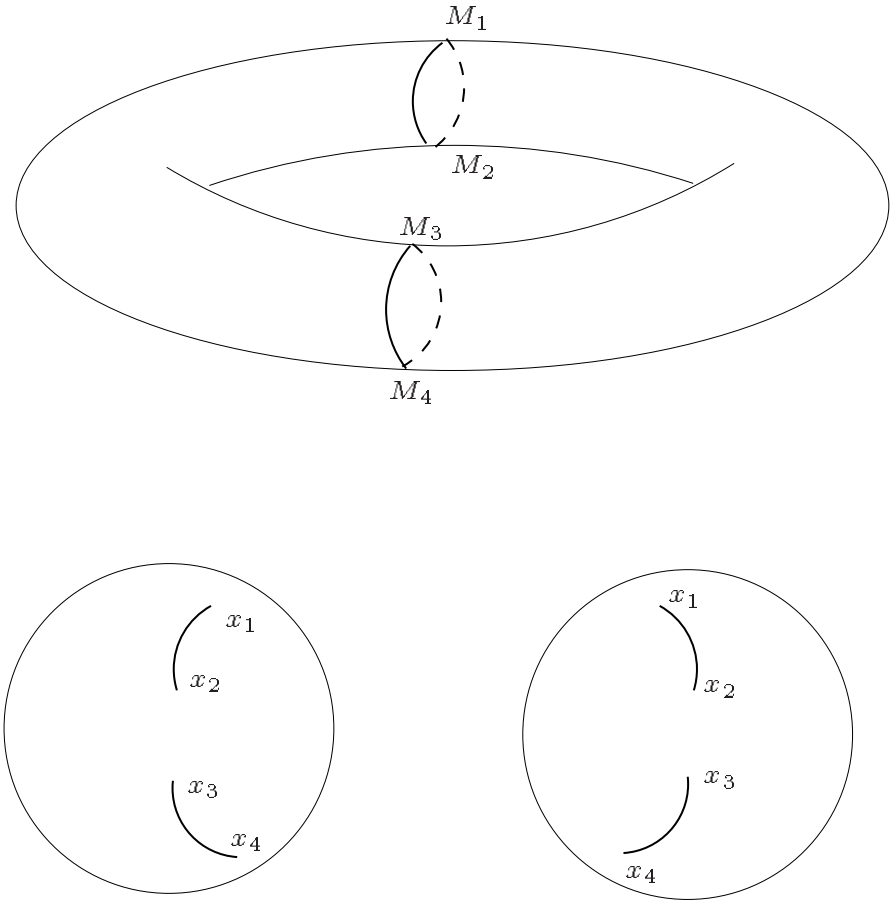


FIG. 9.3: Le tore vu comme deux sphères

Ainsi, un tore a une structure de surface de Riemann si on lui adjoint la donnée des coordonnées locales définies sur les deux sphères complexes associées (il y en a quatre en tout : deux par sphère). Les applications μ et $\mathcal{R}$ seront donc définies de la même manière que précédemment, c'est-à-dire localement sur quatre ouverts du tore. Notons que ces constructions peuvent se généraliser aux tores à plus d'un trou (les tores de genre supérieur à 1).

La surface de Riemann du logarithme

L'application μ peut être facilement construite en posant

$$\forall z \in \mathcal{M}_{\log} \ , \ z = (p,k) \ , \ \mu(z) = p$$

La surface de Riemann du logarithme est souvent qualifiée de "spirale", car si nous posons

$$\forall z \in \mathcal{M}_{\log}\ ,\ z = (p,k)\ ,\ \mathcal{R}(z) = \begin{pmatrix} \Re(p) \\ \Im(p) \\ \mathrm{Arg}(z) \end{pmatrix}$$

alors lorsque $\mathrm{Arg}(z)$ parcourt $\mathbb{R}$, à $|z|$ fixé, le point $\mathcal{R}(z)$ parcourt une trajectoire en forme de spirale.

9.2.5 Première conclusion sur les chemins

L'étude précédente montre que les surfaces de Riemann sont les structures idéales pour y construire des chemins complexes, car elles possèdent intrinsèquement les définitions des deux applications μ et $\mathcal{R}$. Dans certains cas, les chemins peuvent être considérés comme des chemins à éléments dans le plan complexe (dans ce cas, les deux applications peuvent être choisies comme des applications identités). Cependant, pour représenter des résultats sous forme plus compacte, ils pourront être représentés sur la sphère de Riemann. Par exemple, pour visualiser des solutions d'équations différentielles au voisinage du point à l'infini, il peut être intéressant de spécifier des chemins sur la sphère de Riemann, chemins faisant un tour autour du point N de la sphère [Tes97].

Notons cependant que lorsque la complexité des surfaces de Riemann augmente, la détermination d'une application $\mathcal{R}$ permettant de visualiser facilement les éléments des surfaces de Riemann devient un problème difficile, ce qui explique qu'en général nous nous sommes limités concrètement à la prise en compte des surfaces de Riemann hypergéométriques[Spr57], et à la surface de Riemann du logarithme.

9.3 Réalisation logicielle des chemins

9.3.1 Représentations discrètes et continues

Considérons un chemin C basé sur un ensemble $\mathcal{E}$, paramétré par une fonction Φ définie sur l'intervalle réel $I = [t_1, t_2]$:

$$\begin{array}{rccc} \Phi & : & I & \rightarrow & \mathcal{E} \\ & & t & \mapsto & \Phi(t) \end{array} \qquad (3.1)$$

Φ étant continue sur I.

Cette représentation des chemins est intéressante, en particulier pour effectuer des opérations de type *formelles* sur les chemins, par exemple une concaténation de deux chemins, un changement de paramétrisation, etc ... Ces opérations ne demandant pas d'évaluations numériques des points du chemin, ce mode de représentation présente le double avantage d'être compact et rapide à manipuler [DJ95]. D'un point de vue programmation, il est plus délicat à implémenter et à manipuler pour un utilisateur. Lorsqu'un chemin est défini de cette manière, nous dirons qu'il a une *représentation continue* (par opposition à discrète).

Généralement, il arrive un moment où la fonction Φ doit être évaluée, par exemple pour visualiser le chemin dans l'espace. Une *représentation discrète* doit donc aussi être prévue pour ce chemin : pour cela, considérons $n \in \mathbb{N}$, et une subdivision de l'intervalle I :

$$(\theta_0, \theta_1, \ldots, \theta_n)$$

telle que $\theta_0 = t_1$ et $\theta_n = t_2$. En posant $e_i = \Phi(\theta_i)$ pour i compris entre 0 et n, une représentation discrète du chemin C est obtenue en considérant l'ensemble des segments qui composent ce chemin :

$$\{(e_0, e_1), (e_1, e_2), \ldots, (e_{n-1}, e_n)\}.$$

Considérant que, d'un point de vue logiciel, *la représentation d'un chemin est avant tout discrète avant d'être continue*, nous considérerons les chemins comme un ensemble de segments, et éventuellement nous leur associerons une représentation continue. Cette double définition permet d'apporter plus de souplesse aux traitements des chemins, que ces traitements soient de nature numérique ou formelle. De plus, si le besoin s'en fait sentir, une représentation continue peut être retrouvée à partir d'une représentation discrète par l'utilisation d'une fonction d'interpolation.

9.3.2 Les segments

Considérons une classe `E`, qui implémente les éléments de l'ensemble $\mathcal{E}$ introduit précédemment, et en particulier les deux fonctions μ et $\mathcal{R}$.

Un segment de $\mathcal{E}$ est une paire ordonnée (avec un sens de parcours) de points à éléments dans $\mathcal{E}$. Ayant défini la classe `E` avec les conditions précédentes, la classe `Path_Seg<E>` (des segments dont les extrémités sont des éléments de $\mathcal{E}$) peut être définie par :

- ses extrémités P_1 et P_2, comme objets de classe `E`,
- des méthodes de construction,
- une méthode de récupération des nombres complexes associés (la *projection*),
- une méthode de tracé.

La construction de ces segments ne pose généralement pas de problèmes, sachant que les méthodes particulières de projection et de tracé sont obtenues grâce aux méthodes présentes dans la classe `E`.

Récupération des complexes associés

Cette méthode permet d'effectuer des calculs sur les extrémités d'un objet de classe `Path_Seg<E>`. L'ensemble $\mathcal{E}$ permet, par le biais de la fonction μ, de récupérer le nombre complexe associé à l'élément de $\mathcal{E}$ considéré. Il s'ensuit qu'un segment de classe `Path_Seg<E>` peut être transformé en un segment basé sur la classe `Complex` en considérant $(\mu(x_1), \mu(x_2))$ où x_1 et x_2 sont les extrémités du segment dans $\mathcal{E}$.

Sachant que la méthode implémentant la fonction μ doit être présente dans la classe `E` pour pouvoir y construire des chemins, la récupération des nombres complexes ne pose donc aucun problème.

Tracé d'un segment

La fonction $\mathcal{R}$, permet d'associer à un élément de $\mathcal{E}$ sa représentation dans $\mathbb{R}^N$, $(N = 2, 3, 4)$: ainsi une représentation graphique d'un objet de classe `Path_Seg<E>` sera obtenue en traçant dans $\mathbb{R}^N$ le segment correspondant à $(\mathcal{R}(x_1), \mathcal{R}(x_2))$.

9.3.3 Structures de données des chemins

Nous pouvons alors définir la classe `Chemin<E>`, comme une classe de chemins à éléments sur `E`. La représentation discrète des chemins a été privilégiée comme classe de base des chemins, sachant qu'une représentation continue peut en dériver. La classe `Chemin<E>` contient les données suivantes :

- une liste de segments de classe `Path_Seg<E>`,
- une référence au premier et au dernier élément de cette liste.

De plus, cette classe fournit les méthodes suivantes :

- construction de la structure,
- spécification d'un nouveau segment, qui peut être rajouté après le dernier segment composant le chemin,
- tracé des segments composant le chemin.

Enfin, l'opérateur = a été redéfini pour les besoins des chemins.

9.3.4 Les chemins paramétrés

Nous pouvons maintenant considérer ces chemins comme une classe dérivée de la classe `Chemin<E>` : la classe `PChemin<E>`. Cette classe particulière fournit, en plus des données de la classe `Chemin<E>`, les données suivantes :

- les bornes de l'intervalle de paramétrisation,
- le nombre de points à considérer dans cet intervalle,
- une fonction de paramétrisation.

La méthode de paramétrisation est utilisée pour construire les segments composant le chemin. L'intérêt de faire dériver cette classe de chemins paramétrés de la classe générale des chemins permet de considérer les chemins comme des données numériques classiques, par exemple pour en donner une représentation graphique ou pour réaliser des intégrations numériques dessus, mais aussi de manipuler de manière plus formelle ces chemins paramétrés : en effet, de nombreuses manipulations sur les chemins ne nécessitent pas d'évaluation numérique pour être réalisées, comme notamment la concaténation de deux chemins, et peuvent donc se faire à moindre

coût de manière formelle. La construction de la représentation discrète sous-jacente se fait alors par une évaluation de la méthode de paramétrisation pour certaines valeurs du paramètre de parcours, valeurs calculées en prenant en compte les bornes de l'intervalle de paramétrisation ainsi que le nombre de points d'échantillonage.

La classe `PChemin<E>` est une classe *purement virtuelle*, c'est-à-dire qu'il n'est pas possible de créer d'objets de cette classe directement. Cela est dû au fait que la méthode implémentant la fonction de paramétrisation est une méthode abstraite. Il faut donc, pour construire un objet héritant des méthodes de la classe `PChemin<E>`, passer par des classes dérivées, implémentant cette méthode de paramétrisation. À titre d'exemple, plusieurs classes directement dérivées de cette classe ont été définies avec comme ensemble d'éléments de base les complexes :

- la classe `Circle` définie par la fonction

$$\begin{array}{rccc} \Phi & : & [0,1] & \rightarrow & \mathbb{C} \\ & & t & \mapsto & \rho e^{2i\pi t} + c \end{array}$$

  où ρ est le rayon du cercle, et c son centre.

- la classe `Segment` définie par la fonction

$$\begin{array}{rccc} \Phi & : & [0,1] & \rightarrow & \mathbb{C} \\ & & t & \mapsto & bt + (1 \Leftrightarrow t)a \end{array}$$

  où a et b sont les extrémités du segment.

Nous avons donc la hiérarchie de classes suivante :

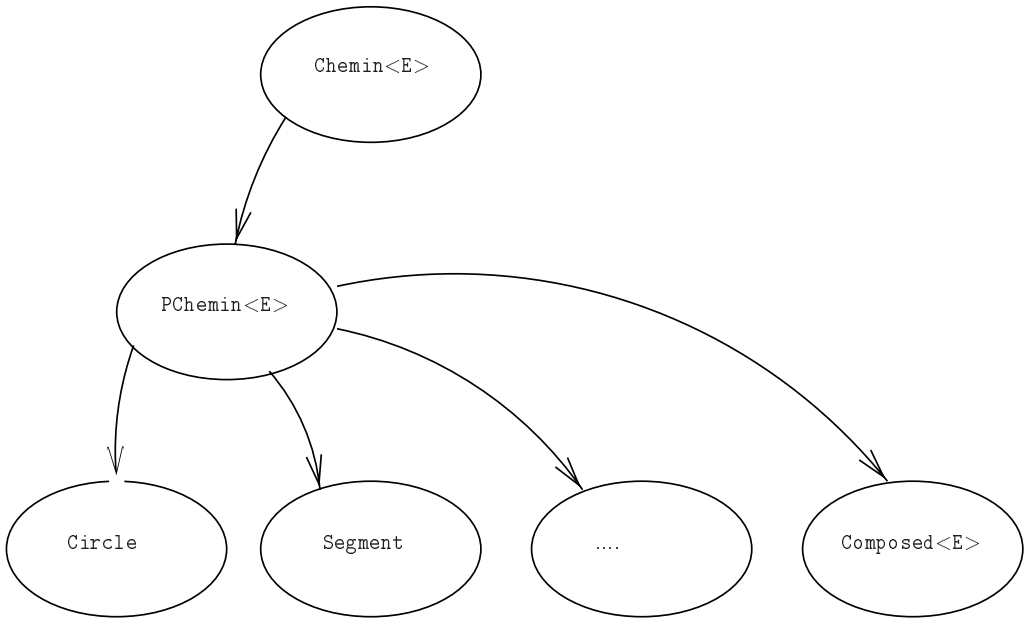


FIG. 9.4: La classe `Chemin<E>` et ses classes dérivées

Cette organisation logique permet de définir facilement d'autres classes de chemins, telles que les chemins en spirale, ou d'autres chemins plus complexes et intéressants au niveau de l'analyse complexe, comme les chemins de Hankel par exemple : un chemin de Hankel est une chemin classique de l'analyse complexe, utilisé notamment dans la méthode de la transformée de Laplace d'une équation différentielle. Ces chemins sont composés d'un cercle et de deux segments, dont une extrémité est repoussée à l'infini. La concaténation des chemins permet de construire facilement de tels chemins.

9.3.5 Concaténation des chemins

La concaténation de deux chemins est l'opération la plus élémentaire qui puisse être faite sur deux chemins. Elle consiste à les coller "bout à bout" afin d'obtenir un nouveau chemin. Si les chemins sont donnés sous forme discrète, cette concaténation ne pose pas vraiment de problème, sachant que la représentation discrète d'un chemin correspond à la donnée de la liste des segments qui le composent, et donc la concaténation de deux chemins revient à rajouter la liste des segments composant le deuxième chemin à celle du premier. En revanche, dans le cas de la concaténation de deux chemins donnés par une représentation continue, il peut être intéressant de donner une représentation elle aussi continue pour le résultat de la concaténation.

Si nous considérons deux chemins C_1 et C_2 , paramétrés par les fonctions Φ_1 et Φ_2, sur deux intervalles $[t_1, t_2]$ et $[t_3, t_4]$, nous pouvons noter C le résultat de la concaténation de C_1 et C_2. Alors C est défini par :

- l'intervalle $[t_1, t_2 + (t_4 \Leftrightarrow t_3)]$,
- la fonction de paramétrisation Φ :

$$\begin{array}{rcl} \Phi \ : \ [t_1, t_2 + (t_4 \Leftrightarrow t_3)] & \rightarrow & \mathbb{C} \\ t & \mapsto & \left\{ \begin{array}{ll} \Phi_1(t) & si \quad t < t_2 \\ \Phi_2(t \Leftrightarrow t_2) & si \quad t \geq t_2 \end{array} \right. \end{array}$$

D'après la définition ci-dessus, les deux chemins C_1 et C_2 n'ont pas besoin d'être adjacents (c'est-à-dire que $\Phi_1(t_2) = \Phi_2(t_3)$) pour que le chemin résultant de la concaténation soit continu : en effet, il existera de manière discrète un segment qui reliera le dernier point de la discrétisation de C_1 au premier point de celle de $C_2 \Leftrightarrow$ lorsque les deux chemins seront concaténés. Cependant, il est préférable que C_1 et C_2 soient adjacents, car si ce n'est pas la cas, le résultat de la concaténation n'est pas un chemin au sens mathématique du terme.

D'un point de vue logiciel, nous avons introduit une classe `Composed<E>`, qui permet de réaliser des concaténations de *chemins paramétrés*. Cette classe dérive de la classe `PChemin<E>`, c'est-à-dire que les objets de la classe `Composed<E>` ont obligatoirement une représentation continue. L'objet ainsi construit possède une méthode de paramétrisation conforme à la définition de la fonction Φ ci -dessus. Lorsque les chemins considérés sont fermés, cette classe permet d'avoir une représentation logicielle complète du groupe d'homotopie[Die68] (muni de la concaténation des chemins comme opération).

9.3.6 Algorithmes

Spécification d'un chemin

La spécification d'un chemin de classe `Chemin<E>`, lorsque `E` est la classe des éléments de base du chemin est l'action de remplir la liste des segments qui composent ce chemin :

```
public void Chemin<E>::Specify ( E point ) ;
```

La spécification de ces segments peut se faire de nombreuses manières : par des valeurs contenues dans un fichier, par la spécification d'un utilisateur grâce à une application qui permet de saisir des points à la souris ou au clavier. Dans le cas particulier où l'objet à spécifier est de classe `PChemin<E>`, cette spécification se fait par évaluation de la méthode de paramétrisation.

Les valeurs spécifiées sont stockées dans la liste contenant les segments, de manière *consécutive* (chaque point est spécifié une fois, mais stocké deux fois, comme fin d'un segment et comme début du segment suivant), ou *disjointe* (chaque point doit être spécifié deux fois si l'on veut que ce point soit le début d'un segment et la fin d'un autre). Ce mode de stockage doit être spécifié dans la définition du chemin.

Création et spécification d'un chemin paramétré

L'intérêt de la vision discrète d'un chemin paramétré est de pouvoir fournir des valeurs lors d'un calcul relatif à ce chemin, ou lors de sa visualisation graphique. Cela signifie en particulier qu'un chemin paramétré n'a pas forcément besoin d'être construit en tant que liste de segments.

Il faut donc pouvoir séparer la *création* d'un chemin paramétré et sa *spécification* :

- la création d'un chemin paramétré est la mise en place de tous ses éléments, y compris l'allocation de la tête de la liste qui contiendra éventuellement les segments qui le composent.
- la spécification d'un chemin paramétré est l'action qui permet de spécifier les sommets des arêtes composant la discrétisation du chemin, notamment au moment où une représentation discrète en est nécessaire.

Si la paramétrisation du chemin est donnée par la méthode

```
public E PChemin<E>::parametrisation ( double t ) ;
```

dans ce cas la spécification du chemin se fera de la manière suivante :

```
public void PChemin<E>::Specify () {
E point ;

  for ( int i=0 ; i < sample_rate ; i++ ) {
     point = parametrisation ( t0 + i/sample_rate * (t1-t0) ) ;
```

```
    Specify ( point ) ;
    //spécification numérique du point de classe E
  }
}
```

Certaines opérations avec les chemins paramétrés sont possibles sans que ces chemins soient spécifiés au préalable : par exemple, la concaténation de deux chemins paramétrés ne nécessite pas leur spécification préalable, car cette concaténation ne fait que créer un autre chemin paramétré dont la fonction de paramétrisation prend en compte la définition de ce chemin comme concaténation de deux autres chemins. Cela autorise certaines opérations sur des chemins en évitant des évaluations coûteuses (en temps, en mémoire).

Intégration dans GANJ

Considérons que la méthode implémentant la fonction $\mathcal{R}$ de l'ensemble $\mathcal{E}$ dans la classe `E` soit la méthode

```
public void E::Format ( double& x, double& y, double& z, int& color ) ;
```

c'est-à-dire que pour tout objet de classe `E`, on peut en donner une représentation graphique dans $\mathbb{R}^N$, ($N = 2, 3, 4$[1]) en faisant agir cette méthode sur l'objet concerné, ses coordonnées spatiales étant contenues dans les variables `x,y,z` et la couleur avec laquelle il sera tracé à l'écran étant contenue dans la variable `color`.

La méthode de tracé d'un segment dans l'environnement graphique GANJ sera alors très simple à mettre en oeuvre. Considérons pour cela que l'objet de classe `Path_Seg<E>` ait accès à une référence sur un objet de classe `GANJ_SubRep`. Nous pouvons alors définir la méthode suivante :

```
public void Path_Seg<E>::Draw ( GANJ_SubRep& GSR) {

 P1.Format ( x1, y1, z1, c1)  ;
 P2.Format ( x2, y2, z2, c2)  ;

 GSR->Segment ( x1, y1, z1,
                x2, y2, z2,
                c1, c2 ) ;
}
```

La méthode de tracé d'un chemin de classe `Chemin<E>` dans l'environnement graphique GANJ sera ainsi basée sur la méthode précédente :

[1] Le cas $N = 4$ est pris en compte par un codage de couleur

```
public void Chemin<E>::Draw ( GANJ_SubRep& GSR) {

  Path_Seg<E> *PS = Path_List ;
  // Path_List est le premier segment composant le chemin

  while ( PS != NULL ) {

  PS.Draw ( GSR ) ;
  PS = PS.next ;
  // next est un pointeur sur le prochain segment
  }


}
```

Le tracé d'un chemin est ainsi conditionné par la méthode de visualisation implémentée dans la classe `E`. Ainsi, si la classe `E` est la classe des nombres complexes, le chemin sera tracé dans le plan complexe, et si `E` est la classe des éléments de S_1, le chemin sera tracé avec ses extrémités sur la sphère de $\mathbb{R}^3$ représentant ${S_1}^2$.

9.4 Exploitation des résultats

La classe `MultiRep` implémente plusieurs méthodes de visualisation (présentées dans le chapitre 2.3.2). Cette classe est une *classe abstraite*, c'est-à-dire qu'on ne peut pas construire directement d'objets de cette classe, car certaines méthodes de cette classe sont virtuelles. Il faut pour cela passer par une classe dérivée, qui doit obligatoirement définir ces méthodes.

La classe générique `MultiRep` gère principalement une table contenant des pointeurs vers les résultats obtenus par intégration. Selon le mode de visualisation souhaité, ces résultats sont exploités différemment :

- la classe `Equation_Rep` : cette classe permet de visualiser l'ensemble des données issues de l'intégration comme des solutions d'une équation différentielle d'ordre N, plutôt que comme un système. Elle permet l'exploitation des solutions en termes de *fonction* et de *dérivées* (c'est-à-dire que la première composante de la solution d'un système différentiel est considérée comme une fonction solution d'une équation différentielle d'ordre N, et la composante i est considérée comme la i^{eme} dérivée de cette fonction), et ainsi elle permet de mesurer visuellement le comportement d'une solution de manière conjointe avec celui de sa dérivée.
- la classe `System_Rep` : cette classe permet de visualiser l'ensemble des données issues de l'intégration comme des solutions de systèmes, c'est-à-dire sans interprétation particulière des différentes composantes.

[2]Si le segment est ré-échantilloné, les points intermédiaires résultant de cet échantillonage peuvent même être projetés sur la sphère, faisant ainsi coller le chemin à la surface.

– il existe de nombreuses autres classes permettant d'exploiter graphiquement les résultats bruts de l'intégration. Ces classes fournissent les méthodes permettant d'effectuer un traitement spécifique sur les données. Par exemple, la classe `Time_Rep` permet de repérer les valeurs calculées pendant une intégration grâce à la valeur du paramètre d'intégration, ou la classe `Function_Rep` permet d'étudier les valeurs de la solution tout en visualisant la distance de la variable d'intégration à un point de plan complexe, typiquement une singularité dont la position peut être calculée en fonction des conditions initiales.

Le principe commun à ces classes est de récupérer des données d'un ensemble de résultats provenant d'une intégration, et d'en proposer une visualisation adaptée. Le principe est détaillé sur la figure 9.5. Ainsi, sur cette figure, un composant stocke les résultats produits par l'outil d'intégration sous forme brute, et ces résultats sont exploités ensuite spécifiquement.

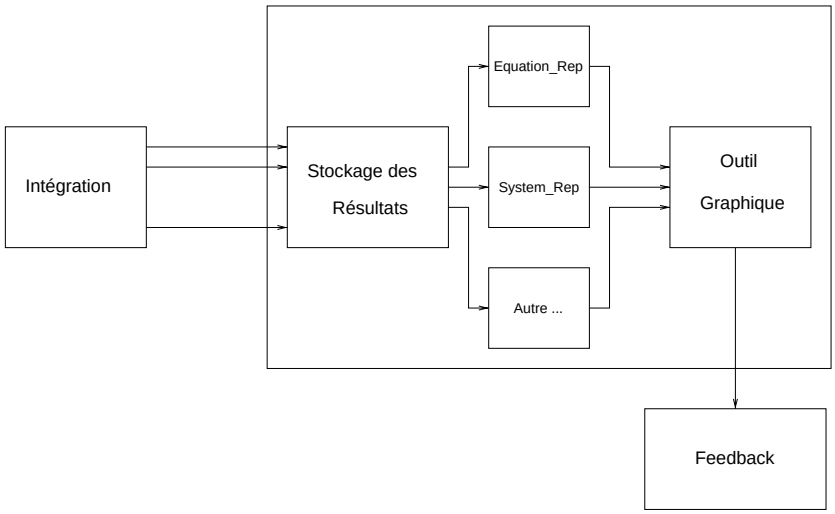


FIG. 9.5: Principe de l'exploitation graphique

Notons que des objets des classes précédemment citées n'ont pas forcément besoin d'être créés pendant l'intégration. En effet, les seules données calculées pendant une intégration sont les valeurs des solutions en certains points du plan complexe, ainsi que les valeurs de l'estimation de l'erreur qui peuvent être calculées en même temps. Cela implique en particulier que des objets permettant d'exploiter de manière spécifique les résultats de l'intégration peuvent être rajoutés *dynamiquement* pendant le déroulement de l'expérience. Plusieurs types de méthodes de visualisation peuvent être utilisées :

– les méthodes génériques : ces méthodes sont définies au niveau de la classe abstraite `MultiRep`, et permettent de tracer une solution quelconque en tant que courbe complexe, ou alors d'effectuer un post-traitement des solutions (comme par exemple un changement de classe pour les éléments de base des chemins). De plus, étant donnés deux objets d'une classe dérivée de `MultiRep`, il existe

au niveau de la classe générique des méthodes permettant d'exploiter ces objets en tant que données primaires ou secondaires (2.3.2).

- les méthodes particulières : ces méthodes permettent de donner une méthode de visualisation adaptée à des cas particuliers, comme par exemple les méthodes de tracé par défaut de la classe `Equation_Rep`, ou des méthodes de tracé particulières à la classe `System_Rep`.

9.4.1 Structure globale de l'expérience

Tous les éléments présentés dans ce chapitre ont été intégrés dans une classe : la classe `MPortrait`. La table 9.1 présente les différents composants de cette classe.

Type d'élément	Contenu	Classe
Spécification	– Les paramètres de départ. – Les chemins d'intégration. – Les chemins de conditions initiales.	– double, int – `Chemin` – `MChemin`
Intégration	– Un intégrateur, spécifié par l'utilisateur.	– `Integrator`
Exploitation Graphique	– Des objets de classe dérivée de `MultiRep`. – Des moyens d'accès à un outil graphique (GANJ).	– `System_Rep`, ... – `GANJ_Rep`, ...
Feedback	– Un module de prise en compte du feedback utilisateur.	

TAB. 9.1: Les différentes composantes de la classe `MPortrait`

Les différents modules de l'application sont tous définis par la spécification de classes génériques, ce qui permet de spécialiser ces modules pour une expérience particulière : par exemple, l'intégrateur peut être changé dynamiquement au profit d'un intégrateur plus puissant ou au contraire plus rapide, les objets servant à la visualisation peuvent être spécialisés à la visualisation d'un phénomène particulier. Cet outil peut être ensuite étendu par son utilisateur, qui n'a plus qu'à coder des modules adaptés à ses besoins : ces modules doivent juste se conformer aux interfaces qui ont été définies pour ceux présentés dans ce chapitre.

Chapitre 10

Visualisation de solutions d'équations différentielles complexes

Dans ce chapitre, nous allons montrer le déroulement d'une expérience visant à intégrer une équation différentielle, et à en visualiser les solutions. Nous montrerons d'abord quelques illustrations du formalisme des *portraits de phase étendus*, ainsi que de la technique de *représentation surfacique* des solutions.

10.1 Visualisation des solutions

10.1.1 La classe des chemins résultats

Dans le cas particulier de la visualisation des solutions d'une intégration dans le plan complexe d'une équation différentielle, les résultats des intégrations sont représentées sous la forme de *portraits de phase étendus*. Puisqu'un portrait de phase étendu est un ensemble de courbes tri-dimensionnelles, il est naturel de stocker les résultats des différentes intégrations dans des objets de classe `MChemin`. Ces classes sont des *templates*, c'est-à-dire qu'elles sont définies avec la donnée d'une classe abstraite. Dans le cas particulier des chemins résultats, la classe de base est la classe `CplusRP`. Cette classe définit essentiellement deux valeurs :

- une valeur complexe, qui est le résultat calculé pendant l'intégration,
- une valeur réelle, qui représente un paramètre additionnel, associé à la valeur complexe.

Supposons que nous intégrons une équation différentielle, avec x la variable d'intégration parcourant un chemin paramétré par une fonction Φ. La valeur complexe d'un élément du chemin est un des résultats de l'intégration, approximant la valeur $y(\Phi(t))$, alors que la valeur réelle est la valeur du paramètre de parcours du chemin d'intégration, c'est-à-dire t. Si un échantillonage en temps du chemin d'intégration est donné par

$$\{t_0, t_1, \ldots, t_N\}.$$

les point x_i fournissent alors une discrétisation du chemin d'intégration :

$$\{x_0 = \Phi(t_0), x_1 = \Phi(t_1), \ldots, x_N\Phi(t_N)\}.$$

Étant donnée une condition initiale, (x_0, y_0), où $y_0 \in \mathbb{C}$, alors les valeurs y^i sont calculées, à l'aide d'un intégrateur noté ci-dessous Integ, à chaque étape de l'intégration, les y_i étant définis par

$$\begin{array}{rcl} y^0 & = & y_0 \\ y^i & = & \text{Integ } (x_i, x_{i-1}, \ldots, y^{i-1}, y^{i-2}, \ldots) \end{array}$$

Ainsi, pendant l'intégration, le chemin résultat est rempli de points de la forme (v_i, p_i), où v_i est une valeur complexe et p_i une valeur réelle associée, tels que

$$\left[\begin{array}{rcl|rcl} v_0 & = & y^0 & p_0 & = & t_0 \\ v_1 & = & y^1 & p_1 & = & t_1 \\ \vdots & & & \vdots & & \\ v_N & = & y^N & p_N & = & t_N \end{array}\right.$$

Ce procédé décrit le remplissage des chemins résultats dans le cas de l'intégration d'une équation différentielle, il s'applique encore dans le cas des systèmes d'ordre N : dans ce cas, le chemin résultat est un ensemble de N chemins ayant la structure précédente.

10.1.2 Changement de la classe de base

L'intérêt d'une telle opération est de pouvoir exploiter de manière différente les valeurs précédemment calculées, dans le cas de la visualisation des résultats, ou dans l'optique d'une méthode de post-traitement. Supposons par exemple que l'on veuille définir une nouvelle méthode de tracé de ce chemin, comme le tracé de la partie complexe des points composant ce chemin sur la sphère de Riemann, et que le paramètre associé à ces valeurs soit porté de manière radiale sur la sphère de Riemann. Pour cela il suffit de définir une nouvelle classe héritant de la classe `CplusRP`, par exemple `CplusRP_Riemann`, cette classe fournissant une telle méthode de tracé, comme montré sur la table 10.1.

Si un objet de classe `MChemin<CplusRP_Riemann>` est construit, la méthode de tracé de ce chemin s'appuiera sur la méthode de tracé de la classe `CplusRP_Riemann`, en traçant le segment reliant dans l'espace les deux points obtenus par la méthode de tracé exposée sur la table 10.1.

La méthode présentée ci-dessus montre un appel à une autre méthode, membre de la classe des éléments de base du chemin : la méthode `Color`. Cette méthode permet d'affecter une couleur par défaut aux éléments de la classe de base des chemins. Si par

```
void CplusRP_Riemann::Format ( CplusRP_Riemann pt ,
                               double& x , double& y , double&z, int& c ) {
 double radius ;

 // réalise la projection stéréographique
 Stereographic ( pt , x , y , z ) ;

 // calcule le rayon
 radius = RADMAX * ( 1. + ( pt.t_ - pt.tmin ) / ( pt.tmax - pt.tmin ) ) ;

 x = x * radius ;
 y = y * radius ;
 z = z * radius ;

 c = pt.Color () ;
}
```

TAB. 10.1: Méthode de tracé de la classe `CplusRP_Riemann`

la suite la couleur affectée à un point n'est pas remplacée par une couleur issue d'une méthode plus spécifique, le point apparaîtra à l'écran avec sa couleur par défaut. Dans le cas où les deux extrémités d'une segment sont de couleurs différentes, les points intermédiaires sont tracés avec une couleur calculée par interpolation linéaire entre celles des extrémités du segment.

10.1.3 Exemple sur une équation

Considérons que nous voulons visualiser les solutions de l'équation différentielle

$$y' = x.y^3 \tag{1.1}$$

lorsque x parcourt un cercle centré en 2, de rayon 1; ceci pour différentes valeurs de y_0, situées sur un cercle de rayon petit (dans cet exemple, nous prendrons 0.3) centré en 0.

Les valeurs intermédiaires de y sont calculées en utilisant un intégrateur de type Runge-Kutta, effectuant un contrôle local de l'erreur d'intégration, réalisé par René Aïd ([Aid99, Aid96]). Les résultats sont présentés sur la planche 10.2.

Figure 10.2 Cette figure montre un ensemble de solutions, chacune étant obtenue à partir d'une condition initiale (x_0, y_0), pour y_0 pris régulièrement sur le cercle de centre 0, de rayon 0.3. Cet ensemble peut être retrouvé directement sur la figure en ne considérant que les points d'altitude 0. La couleur de chaque point des courbes a été obtenue en calculant la couleur par défaut du y_0 qui a été utilisé pour calculer les

valeurs de la solution. L'intérêt de cette représentation est tout d'abord de donner une représentation *dynamique* des solutions de l'équation différentielle. En effet, lorsque le *temps* d'intégration (i.e. le paramètre de parcours du chemin d'intégration) varie, le comportement de la solution de l'équation en fonction du temps peut être étudié de manière conjointe des variations de la variable d'intégration. De plus, les variations de la solution peuvent être visualisées de manière *globale*, ce qui permet d'apprécier visuellement la sensibilité de la solution de l'équation en fonction des conditions initiales. En effet, sur cette figure, les courbes issues des conditions initiales pour des valeurs de y_0 proches de l'axe réel (les courbes bleues pour les valeurs proches de l'axe réel positif, les courbes jaunes pour les valeurs proches de l'axe réel négatif) ont une concentration de couleur importante, ce qui d'un point de vue mathématique peut être interprété en termes de dépendance aux conditions initiales : pour deux conditions initiales proches, les courbes résultats de l'intégration restent proches. De même, les courbes issues des conditions initiales telles que y_0 est proche de l'axe imaginaire (les courbes vertes et violettes sur la figure) ont tendance à s'éloigner les unes des autres pendant le parcours du temps d'intégration : pour ces conditions initiales, les solutions sont plus sensibles aux conditions initiales.

10.2 Représentation surfacique des solutions et applications

10.2.1 Un exemple de représentation surfacique

Considérons l'équation (1.1). Dans l'expérience précédente, pour certaines conditions initiales, les valeurs de la solution étaient calculées, puis tracées en tant que courbes tri-dimensionnelles. Cependant, ces courbes ne permettent pas d'appréhender le portrait de phase de manière continue. Une représentation surfacique, obtenue par interpolation linéaire entre ces différentes courbes (comme expliqué en 2.2) permet d'obtenir une visualisation de la solution en tant que surface bi-paramétrée, par t, le temps d'intégration, et u, le paramètre de parcours de l'ensemble γ où les valeurs de y_0 sont prises.

Figure 10.3 Cette figure montre un exemple de visualisation surfacique, basée sur les courbes tracées sur la figure 10.2. Cette visualisation continue de la solution en tant que surface permet de l'envisager comme une fonction de *flot complexe*[1]. De plus, l'utilisation d'algorithmes de suppression des parties cachées de la scène tri-dimensionnelle permet de repérer facilement à quelles parties de la surface correspondent une valeur de temps d'intégration ou une condition initiale. Sur cette figure, la couleur affectée à chaque point de la surface est déterminée de la manière suivante :

[1]bien que cette notion de flot ne soit pas définie pour une équation de la variable complexe, car elle est dépendante du chemin d'intégration ; la fonction représentée ici est en fait la fonction de flot du système dynamique associé.

- si le point appartient à une courbe résultat calculée pendant l'intégration, sa couleur est celle de la condition initiale avec laquelle la solution a été calculée.
- si le point est situé entre deux courbes résultat, il appartient à un polygone, et sa couleur est une interpolation (bilinéaire) des couleurs des sommets du polygone.

Il est intéressant de noter que l'interpolation réalisée pour la réalisation de ce mode de visualisation facilite l'interprétation en termes de concentration de la couleur sur la surface. De plus, cette surface étant une variété de dimension (réelle) deux, sa visualisation ne nécessite que trois dimensions. Cela implique qu'il reste une dimension pour visualiser en même temps d'autres données issues de l'intégration ou qui peuvent être calculées extérieurement à l'intégration, comme le permettent les données primaires et secondaires. Enfin, d'un point de vue pratique, cette interpolation est réalisée sans surcoût pour l'application client, car elle est prise en charge au niveau serveur par l'intermédiaire du *smooth shading*.

10.2.2 Utilisation des données primaires et secondaires

Le nombre de dimensions réelles utilisées pour une visualisation surfacique est égal à trois : la quatrième dimension disponible peut être utilisée pour visualiser des données échantillonées sur un intervalle, dont les valeurs sont codées par une couleur. Ce deuxième jeu de données (les *données secondaires*) doit être lui-même contenu dans un objet de classe dérivée de la classe `MultiRep`, de même que l'objet *primaire*, contenant les données primaires (dans ce cas, les valeurs de la solution de l'équation).

Nous considérons ici l'exemple du tracé des valeurs du temps d'intégration, porté directement sur la surface représentant les solutions de l'équation. Pour cela, il suffit de déclarer la classe `Time_Rep`, qui implémente les méthodes abstraites de la classe `Multi_Rep`, c'est-à-dire les méthodes de tracé par défaut. Dans le cas de la visualisation en temps que donnée secondaire uniquement (comme c'est ici le cas), ces méthodes de tracé par défaut n'ont que peu d'intérêt. Par contre, le tracé du temps d'intégration porté directement sur la surface nécessite une méthode de quantification, méthode utilisée au niveau de la classe `Multi_Rep` afin d'affecter une couleur aux données secondaires tracées.

La méthode par défaut de quantification consiste à repérer le minimum (v_{min}) et le maximum (v_{max}) des valeurs de la donnée secondaire pour chaque jeu de données de la donnée primaire, et ainsi chaque valeur secondaire est affectée d'une couleur calculée par une fonction linéaire de la forme :

$$\mathcal{Q} : [v_{min}, v_{max}] \Leftrightarrow [0, couleurmax]$$

Une autre méthode de quantification permet de calculer les minimum et les maximum sur *l'ensemble* des données secondaires, v^*_{min} et v^*_{max}.

$$\mathcal{Q}^* : [v^*_{min}, v^*_{max}] \Leftrightarrow [0, couleurmax]$$

Cette deuxième méthode de quantification permet de repérer des pics de données sur l'ensemble des données secondaires, et peut se révéler particulièrement utile pour la visualisation de fonctions définies pour chaque valeur de la donnée primaire : par exemple, la visualisation de l'erreur globale d'intégration peut tirer parti de cette méthode de quantification. Par contre, pour la visualisation du temps d'intégration, les deux applications $\mathcal{Q}$ et $\mathcal{Q}^*$ sont égales.

Figure 10.4 Cette figure montre un exemple de visualisation de la solution de l'équation (1.1), avec comme données primaires les valeurs calculées par intégration de la solution de l'équation, et comme données secondaires le temps d'intégration. L'intérêt de ce mode de représentation est de pouvoir situer dans le temps les variations de la solution. En effet, si un certain type de comportement de la solution doit être détecté, comme par exemple de brutales variations en module, cette méthode de visualisation permet de repérer facilement les valeurs du temps auxquelles correspondent ces comportements. Une première application est de détecter les valeurs du temps pour lesquelles l'intégration produit de grosses variations en modules, et donc de raffiner localement autour des valeurs repérées la subdivision temporelle de l'intervalle d'intégration. Une deuxième application consiste à repérer à quelles valeurs de la variable d'intégration correspondent les valeurs du temps repérées visuellement sur la figure, et à adapter la discrétisation du chemin d'intégration en fonction d'un raffinement du chemin pour les valeurs du paramètre autour des valeurs relevées.

10.2.3 Lisibilité de ces surfaces

L'algorithme de composition des surfaces est très simple, puisqu'il consiste en la composition de polygones avec des valeurs consécutives de la solution calculée, et donc il n'introduit guère de surcoût d'exploitation. Les intérêts de ce modèle de visualisation surfacique sont de plus multiples :

- L'étude des surface est plus agréable à l'oeil que celle d'un ensemble de courbes, car elles donnent un sentiment de continuité, continuité sous-jacente aux grandeurs mathématiques qu'elles représentent.
- l'interpolation qui est réalisée est faite avec un coût zéro, car elle est réalisée au niveau de l'outil graphique et non pas de l'outil de calcul. En plus d'améliorer la visualisation des surfaces, elle permet d'économiser en temps de calcul aussi bien sur les intervalles d'intégration (les couleurs sont affectées à chaque valeur, et donc les couleurs intermédiaires sur une même courbe sont interpolées entre deux valeurs déjà calculées), que sur l'ensemble de conditions initiales (le sentiment de continuité est donné sans calculs intermédiaires, pour une résolution suffisement fine de l'ensemble de conditions initiales).

Ces remarques ne sont valides que si la surface composée est suffisemment régulière. En effet, si l'intégration de l'équation a été interrompue, c'est-à-dire que l'intégrateur n'a pas réussi à calculer les valeurs de la solution pour toutes les valeurs de la variable d'intégration, la composition de la surface peut ne pas être significative,

car les solutions peuvent très bien changer radicalement de comportement (nous en verrons des exemples dans le chapitre suivant), et donc l'interpolation des valeurs peut très bien ne pas représenter une situation réelle (voir figure 10.1).

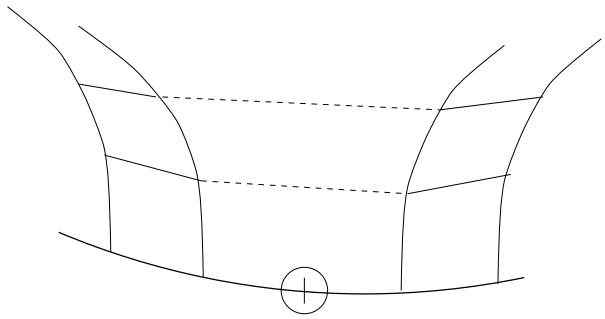


FIG. 10.1: Un exemple de non-interpolation

Pour la valeur de la condition initiale entourée sur la figure 10.1, l'intégration a été interrompue, parce que la méthode divergeait. Ainsi, il est possible que la solution prenne un module particulièrement important pendant l'intervalle de temps d'intégration, et donc les segments tracés en pointillés sur la figure, qui représenteraient l'interpolation entre les deux courbes consécutives, donneraient de fausses informations sur le comportement de la solution avec une condition initiale proche de celle entourée sur la figure. La solution dans ce cas consiste à ne pas composer les polygones décrits en pointillés sur la figure, et donc de laisser cette partie de la surface vide.

De plus, la composition des surfaces doit pouvoir prendre en compte des phénomènes difficiles à visualiser, comme des inversions de polygones, notamment dans le cas où les courbes qui composent la surface ont des trajectoires tourbillonantes. Dans ce cas, la visualisation surfacique est beaucoup plus difficile à apprécier que la visualisation sous forme d'ensemble de courbes.

10.2.4 Application à la visualisation de systèmes

Pour finir avec la représentation surfacique, nous allons voir des illustrations de la visualisation de systèmes, avec une visualisation des solutions du système

$$\left\{ \begin{array}{rcl} y_1' & = & y_2 \\ y_2' & = & y_1 + x^2 \end{array} \right. \tag{2.2}$$

Ce système provient en fait de l'équation linéaire d'ordre deux (2.3)

$$y'' = y + x^2 \tag{2.3}$$

Les solutions du système peuvent donc soit être considérées comme étant les composantes d'un système différentiel de dimension deux et de degré un, soit comme la solution de l'équation différentielle d'ordre deux, et la dérivée de cette fonction.

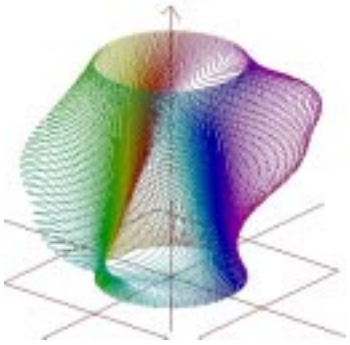


FIG. 10.2: Visualisation des courbes solutions

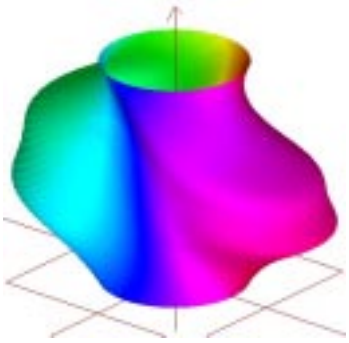


FIG. 10.3: Visualisation surfacique

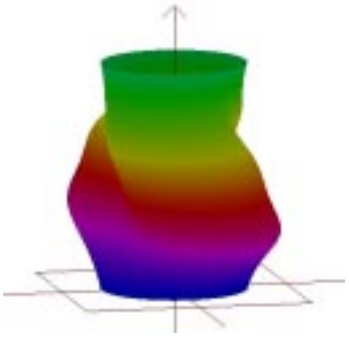


FIG. 10.4: Temps d'intégration en donnée secondaire

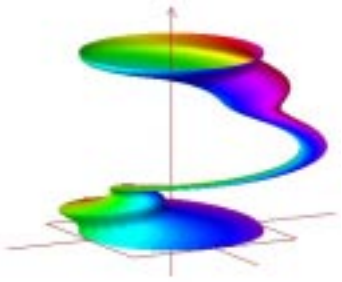


FIG. 10.5: Première composante solution

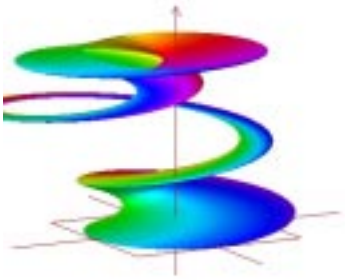


FIG. 10.6: Deuxième composante solution

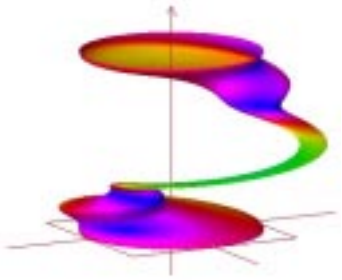


FIG. 10.7: La solution et sa dérivée

TAB. 10.2: Visualisation de solutions d'équations complexes

Dans le cas où ces solutions sont considérées comme les composantes d'une solution de dimension deux, un objet de classe `System_Rep` peut être utilisé pour les visualiser : cet objet possède toutes les méthodes nécessaires au tracé de chaque composante. Dans le cas où la première composante de la solution du système est considérée comme la solution de l'équation différentielle (2.3), il faudra utiliser un objet de classe `System_Rep`, qui possède les méthodes utiles pour visualiser les solutions de dimension un.

Figures 10.5 et 10.6 Ces deux figures montrent les deux composantes de la solution du système (2.2). L'affectation des couleurs aux points de la surface a été faite en fonction de la provenance en termes de conditions initiales, et cela permet donc de suivre sur la surface une solution (ou un ensemble de solutions) particulières. Néanmoins, il est difficile de considérer les deux composantes de manière globale.

Figure 10.7 Cette figure illustre le principe de la visualisation de la solution d'un système en tant que solution d'une équation d'ordre deux. La donnée primaire utilisée pour cette figure est ici la solution de l'équation (la première composante de la solution du système), alors que la donnée secondaire est le module de la dérivée de cette solution (la seconde composante de la solution du système). Il est ainsi plus facile d'étudier sur cette figure le comportement de la solution, en même temps que celui de sa dérivée.

10.3 Caractérisation visuelle de phénomènes multiformes

10.3.1 Exemple

Pour situer le problème, nous pouvons reprendre l'exemple de l'équation (1.1), et observer les résultats de l'intégration lorsque les conditions initiales varient. Considérons pour cela que l'ensemble sur lequel les conditions initiales sont prises est un cercle centré en 0, mais cette fois de rayon légèrement supérieur, que nous prendrons égal à 0.5. Le résultat est exposé sur la planche 10.4.

Figure 10.10 Cette figure montre le résultat de la nouvelle intégration, avec des conditions initiales appartenant au cercle centré en 0, de rayon 0.5, alors que le chemin d'intégration est toujours le même cercle. Certaines courbes (en rouge, en vert) correspondant à des conditions initiales dont les arguments sont proches de $\frac{\pi}{2}$ (pour les courbes en rouge) et de $\Leftrightarrow\frac{\pi}{2}$ (pour les courbes en vert) ne reviennent pas sur elles-mêmes après l'intégration, mais prennent une valeur opposée à leur valeur de départ. Ce comportement peut être dû à plusieurs facteurs :

- une mauvaise intégration, entraînant une dégradation des résultats pour des conditions initiales situées dans certaines zones (ceci peut être dû à un mauvais

comportement numérique de l'intégrateur, ou alors à des phénomènes comme ceux exposés dans la partie II).

– à un caractère multiforme de la solution de l'équation. Rien n'interdit à une solution d'une équation différentielle holomorphe d'être multiforme, car la solution est holomorphe (théorème 2.1.1.1), mais *localement*.

La figure 10.11 semble privilégier la seconde solution.

Figure 10.11 Cette figure montre le résultat de l'intégration, avec le même ensemble de conditions initiales que sur la figure 10.10, mais cette fois avec un chemin d'intégration parcouru *deux fois* : le cercle centré en 2, de rayon 1. Les courbes qui ne reprenaient pas leur condition initiale après intégration cette fois-ci reviennent bien sur elles-mêmes après l'intégration sur le nouveau chemin. Le comportement des solutions associées à ces courbes ressemble à celui des fonctions racines carrées.

10.3.2 Vérification par le calcul

Bien que l'équation différentielle (1.1) soit assez facile à résoudre de manière exacte, il est plus difficile de prévoir le comportement général d'une de ses solutions, associée à une condition initiale. En effet, il faut pour cela trouver l'emplacement des singularités de la fonction, puis ensuite étudier les positions relatives de ces singularités avec le cercle d'intégration, sur lequel la solution prend ses valeurs. En effet, si le chemin d'intégration entoure une ou plusieurs singularités (pour certaines valeurs de y_0), dans ce cas la solution n'a que peu de chance de retrouver sa valeur initiale après avoir parcouru le chemin d'intégration. Par contre, si le chemin d'intégration n'entoure aucune singularité, les valeurs de la solution aux extrémités du chemin fermé doivent être égales.

Calcul de la solution de l'équation

L'équation (1.1) est simple à intégrer. En effet, elle peut être réécrite

$$\frac{dy}{y^3} = x.dx$$

et donc, pour une condition initiale donnée (x_0, y_0),

$$\Leftrightarrow \frac{1}{2}(y^{-2} \Leftrightarrow y_0^{-2}) = \frac{1}{2}(x^2 \Leftrightarrow x_0^2)$$

d'où

$$y^{-2} = x_0^2 + \frac{1}{y_0^2} \Leftrightarrow x^2$$

ce qui nous donne

$$y(x) = \frac{1}{\sqrt{x_0^2 + \frac{1}{y_0^2} \Leftrightarrow x^2}} \tag{3.4}$$

Cette fonction admet deux points de branchement d'ordre deux, et la position de ces points de branchement dans le plan complexe varie en fonction de x_0 et y_0 : ce sont les singularités mobiles de l'équation (1.1). L'expression ci-dessus explique le fait que la solution calculée de l'équation ne retrouve pas, pour certaines conditions initiales, ses valeurs de départ après avoir parcouru *une fois* le chemin d'intégration [Wey55], et le retour sur les valeurs initiales après avoir parcouru *deux fois* le chemin d'intégration peut être aussi justifié par l'expression (3.4). Nous allons détailler ce raisonnement dans le chapitre suivant.

Position des singularités

Les singularités mobiles de la solution sont données par l'expression annulant le terme placé sous la racine de l'expression (3.4) :

$$\sigma_k(x_0, y_0) = (\Leftrightarrow 1)^{k-1} \sqrt{x_0^2 + \frac{1}{y_0^2}}, k = 1, 2 \tag{3.5}$$

Ainsi, dans les conditions de l'expérience décrite ci-dessus, pour certaines valeurs de y_0 dans le cercle centré en 0 de rayon 0.4, exactement une singularité se trouve *à l'intérieur* du cercle d'intégration. En effet, puisque $\sigma_1 = \Leftrightarrow\sigma_2$, si σ_1 se trouve à l'intérieur du cercle C, *non centré en* 0, alors σ_2 est à l'extérieur du cercle C. Ainsi, pour certaines valeurs de $y_0 \in \gamma$, une seule singularité sera présente à l'intérieur du cercle d'intégration : puisque la solution correspondant à y_0 possède cette singularité comme point de branchement, cette solution n'a pas les mêmes valeurs aux extrémités du cercle C.

De manière précise, nous pouvons calculer, étant donné un chemin d'intégration C (et donc une valeur x_0), la valeur limite de y_0 pour laquelle la solution de l'équation a un comportement multiforme. Pour cela, il faut calculer à quelle condition (portant sur $|y_0|$) le cercle d'intégration et l'ensemble (lorsque y_0 varie) des $\sigma_i(x_0, y_0)$ ont une intersection non vide.

Considérons que l'équation est intégrée pour x prenant ses valeurs dans le cercle C de centre c, de rayon r. À titre de simplification, supposons que $c \in \mathbb{R}_+$. La valeur de x_0 est choisie comme le plus grand point d'intersection de C avec l'axe réel, c'est-à-dire le point $x_0 = c + r$.

Donnons nous y_0 fixé dans $\mathbb{C}$, et soit $\sigma = \sigma(x_0, y_0)$ une des deux singularités de l'équation, et étudions à quelles condition σ est située à l'intérieur du cercle C. Nous commencerons par étudier à quelle condition σ peut être sur le cercle :

$$\exists\theta \in [0, 2\pi], \sigma = r.e^{i\theta} + c$$

En reportant cette valeur de σ dans sa définition, nous obtenons

$$\frac{1}{y_0^2} = (r.e^{i\theta} + c)^2 \Leftrightarrow x_0^2$$

Or, $x_0 = c + r$, d'où

$$\frac{1}{y_0^2} = (r.e^{i\theta} + c)^2 \Leftrightarrow (c + r)^2$$

En développant cette expression, nous obtenons la condition nécessaire et suffisante d'appartenance de σ au cercle C :

$$\frac{1}{y_0^2} = r^2 e^{2i\theta} + 2rce^{i\theta} \Leftrightarrow (r^2 + 2rc) = G_{c,r}(\theta) \tag{3.6}$$

La fonction $G_{c,r}(\theta)$ définie ci-dessus est une fonction de la variable réelle θ, à valeurs complexes. Nous noterons par la suite $F_{c,r}(\theta)$ le module de cette fonction, $|G_{c,r}(\theta)|$.

Une condition nécessaire pour avoir des solutions (en θ, pour r, c et y_0 donnés) est d'avoir l'égalité des modules des deux membres de cette égalité. Cela donne

$$\frac{1}{|y_0^2|} = F_{c,r}(\theta)$$

où nous pouvons expliciter $F_{c,r}(\theta)$:

$$F_{c,r}(\theta) = (r^2\cos(2\theta) + 2rc\cos(\theta) \Leftrightarrow (r^2 + 2rc))^2 + (r^2\sin(2\theta) + 2rc\sin(\theta))^2 \tag{3.7}$$

Pour avoir une condition nécessaire, il nous suffira donc d'étudier la fonction (3.7).

En calculant la dérivée de cette fonction, et en réduisant cette expression, nous obtenons

$$F'_{c,r}(\theta) = 8r^2\sin(\theta)(c^2 + 2rc\cos(\theta) + r^2\cos(\theta))$$

D'après l'expression ci-dessus, nous voyons que la fonction $F_{c,r}(\theta)$ admet un *extremum* pour $\theta = k\pi$, et dans le cas où $\theta \neq k\pi$, nous pouvons rechercher les autres extremums de cette fonction : ils sont donnés par les solutions en θ de l'équation

$$c^2 + 2rc\cos(\theta) + r^2\cos(\theta) = 0$$

ce qui nous donne

$$\cos(\theta) = \frac{\Leftrightarrow c^2}{r(r + 2c)} \tag{3.8}$$

Ainsi, dans le cas où $\frac{-c^2}{r(r+2c)} > 1$ l'équation (3.8) n'admet pas de solutions. Le *maximum* de la fonction $F_{c,r}(\theta)$ sera atteint pour $\theta = \pi$.

	cond.1	cond. 2	nombre d'intersections
cas 1	$\frac{c^2}{r(r+2c)} \geq 1$	$\frac{1}{y_0^2} > F_{c,r}(\pi)$	0
cas 2	$\frac{c^2}{r(r+2c)} \geq 1$	$\frac{1}{y_0^2} \leq F_{c,r}(\pi)$	2 (1)
cas 3	$\frac{c^2}{r(r+2c)} < 1$	$\frac{1}{y_0^2} > F_{c,r}(\theta_m^1)$	0
cas 4	$\frac{c^2}{r(r+2c)} < 1$	$F_{c,r}(\pi) < \frac{1}{y_0^2} \leq F_{c,r}(\theta_m^1)$	4 (2)
cas 5	$\frac{c^2}{r(r+2c)} < 1$	$\frac{1}{y_0^2} \leq F_{c,r}(\pi)$	2 (3)

TAB. 10.3: Le nombre d'intersections en fonction des paramètres d'intégration

Dans le cas où $\frac{-c^2}{r(r+2c)} < 1$, l'équation (3.8) admet deux solutions θ_m^1 et θ_m^2, symétriques par rapport à π, et la fonction $F_{c,r}(\theta)$ admet un *minimum* local en $\theta = \pi$. On a de plus l'égalité

$$F_{c,r}(\theta_m^1) = F_{c,r}(\theta_m^2)$$

puisque les valeurs de cette fonction sont symétriques par rapport à π.

La condition nécessaire pour avoir des solutions à l'équation (3.6) est donc résumée sur le tableau 10.3 (les nombres entre paranthèses indiquant les cas limites).

Nous pouvons aussi reprendre ces résultats sur la figure 10.8 où les différents cas de figure sont tracés (l'absisse représente les valeurs de θ, et l'ordonnée les valeurs de la fonction $F_{c,r}(\theta)$). Les lignes en pointillés sur cette figure représentent les positions de la valeurs de $\frac{1}{y_0^2}$ par rapport aux extrema de la fonction $F_{c,r}(\theta)$

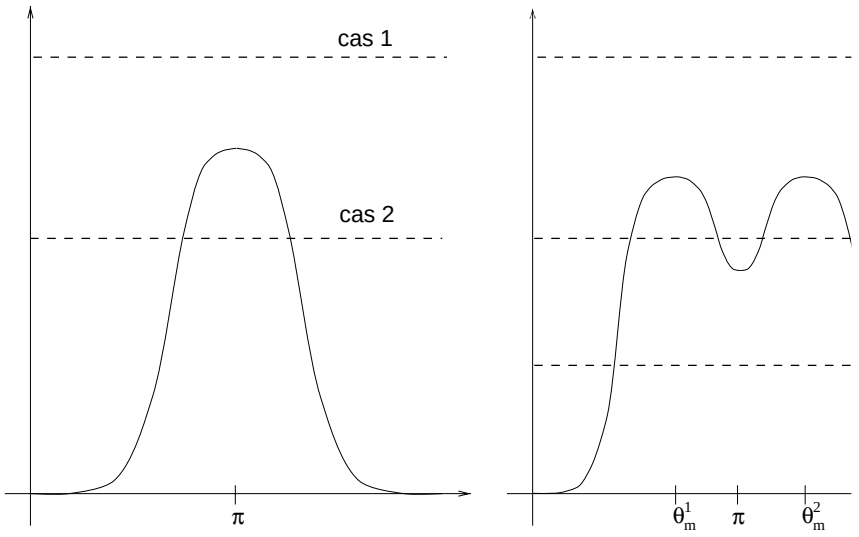


FIG. 10.8: Les différents cas de figure

Maintenant que nous avons obtenu des conditions nécessaires, nous allons étudier à quelles conditions elles sont aussi suffisantes. En effet, rien ne garantit qu'il y aura une intersection entre l'ensemble des singularités obtenu lorsque y_0 parcourt le

chemin des conditions initiales et le chemin d'intégration, car la partie arguments de l'équation (3.6) n'est pas forcément vérifiée. Nous allons donc vérifier que dans le cas où les divers paramètres d'intégration vérifient les conditions nécessaires 10.3, nous arrivons (sous certaines conditions sur γ) à construire les intersections entre le chemin d'intégration et l'ensemble des singularités.

Supposons donc que l'ensemble des conditions initiales γ est un cercle, centré en 0, et que le module des éléments de ce cercle vérifie la condition 1. Considérons de plus C fixé, tel que c et r vérifient la condition 2 de manière à ce que l'équation

$$\frac{1}{y_0^2} = F_{c,r}(\theta)$$

admet des solutions : nous nous plaçons donc dans les cas 2, 4 ou 5. Nous pouvons donc calculer une valeur θ_0 qui satisfait l'équation précédente. Nous pouvons donc en déduire une valeur σ_0, par la formule

$$\sigma_0 = r.e^{i\theta_0} + c$$

La question est alors de savoir s'il existe un $y_0 \in \gamma$ tel que

$$\sigma_0 = \sigma(x_0, y_0)$$

Réécrivons alors l'équation (3.6) sous la forme

$$|\frac{1}{y_0^2}| = F_{c,r}(\theta_0) \tag{3.9}$$

$$\arg(\frac{1}{y_0^2}) = \arg(G_{c,r}(\theta_0)) \tag{3.10}$$

Dans le cas où γ est un cercle centré en 0, $\mathrm{Arg}(y_0^2)$ prend toutes les valeurs de l'intervalle $[0, 4\pi]$, et par suite $\mathrm{Arg}(\frac{1}{y_0^2})$ prend toutes les valeurs de l'intervalle $[\Leftrightarrow 4\pi, 0]$. La condition (3.10) sera donc satisfaite deux fois pour chaque valeur de θ_0. La condition nécessaire dans le cas où γ est un cercle centré en 0 est donc bien suffisante.

Si γ n'est pas un cercle centré en 0, il faut alors étudier l'intervalle des valeurs prises par $\mathrm{Arg}(\frac{1}{y_0^2})$ lorsque y_0 parcourt γ, et vérifier si la valeur de $\mathrm{Arg}(G_{c,r}(\theta_m^i))$ est bien comprise dans cet intervalle. Un tel cas de figure est montré sur la figure 10.9.

Dans le cas de notre expérience, cela suffira à caractériser le comportement des solutions, en fonction des valeurs de y_0, c et r. Remarquons à ce propos que pour $y_0 \in \mathbb{R}$, on a $\sigma(x_0, y_0) \in \mathbb{R}$, et que de plus $\sigma(x_0, y_0) > x_0$.

Ainsi, lorsque y_0 parcourt γ, la première valeur prise par y_0 est réelle, et donc la solution calculée (lorsqu'elle a pu être calculée) ne présentera pas de caractère multiforme. Par la suite, si les conditions 10.3 sont satisfaites, la solution calculée de l'équation changera brutalement de comportement, pour certaines valeurs de y_0, pour revenir sur la fin de l'intégration à des valeurs différentes de ses valeurs initiales.

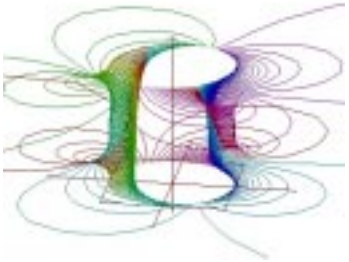


FIG. 10.10: $|y_0| = 0.5$: caractère multiforme

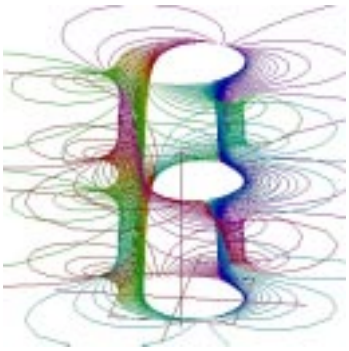


FIG. 10.11: $|y_0| = 0.5$, deux intégrations successives

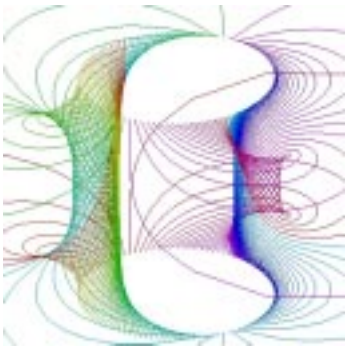


FIG. 10.12: Deuxième cas

FIG. 10.13: Quatrième cas

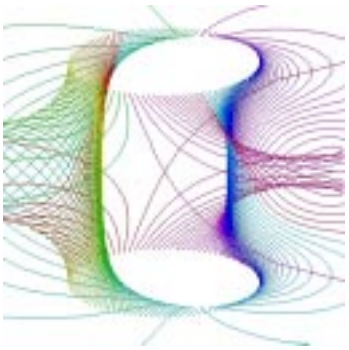


FIG. 10.14: Cinquième cas

TAB. 10.4: Caractère multiforme des solutions

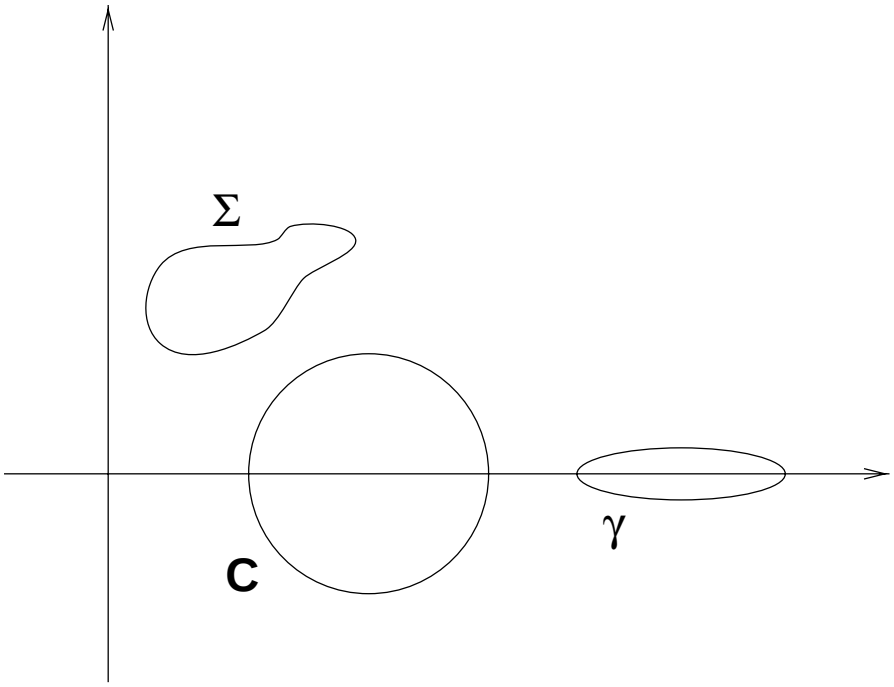


FIG. 10.9: Un cas de non-suffisance des conditions nécessaires

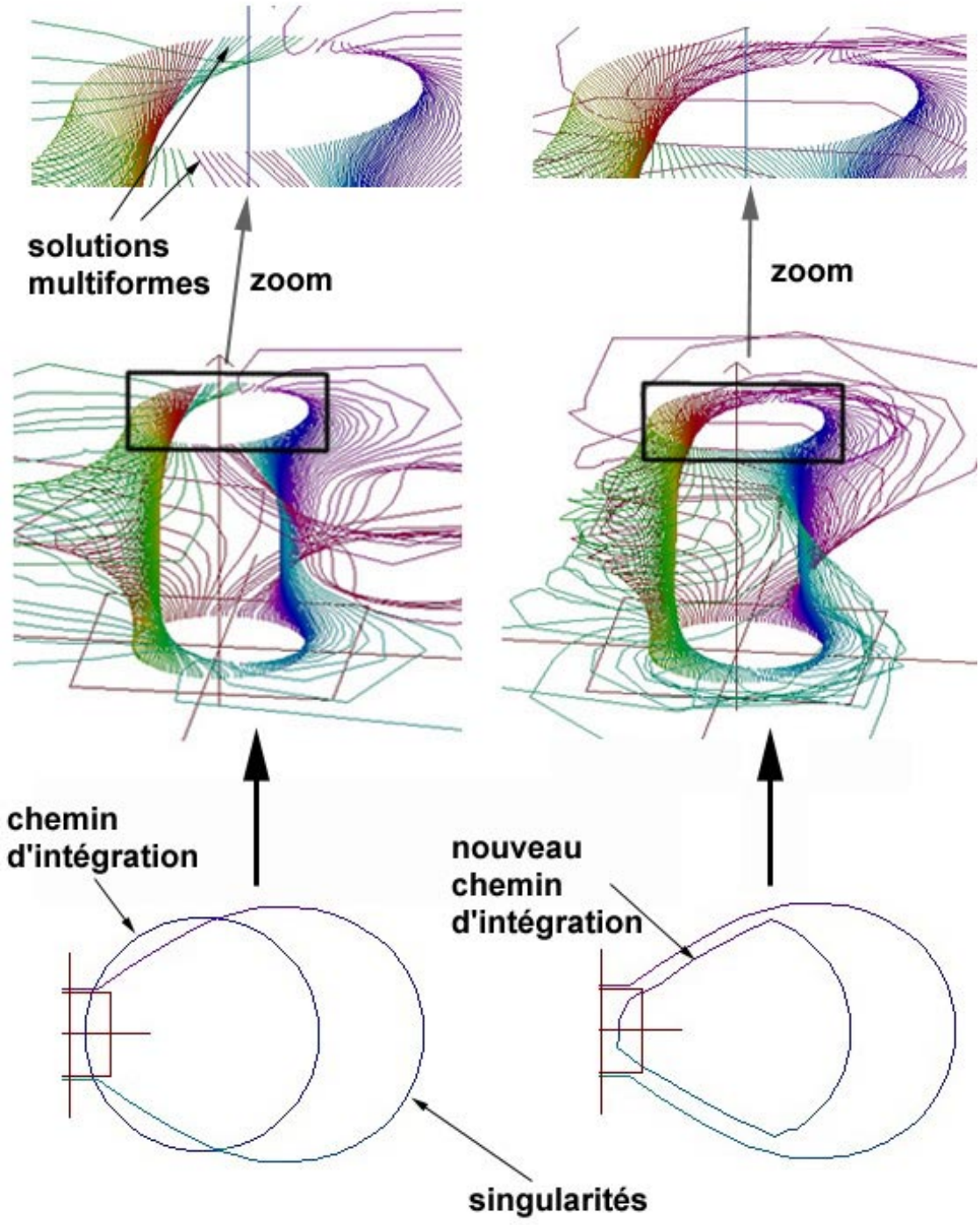
Les figures 10.12, 10.13 et 10.14 illustrent certains cas définis dans le tableau 10.3 : les cas 2, 4 et 5. La présence des courbes correspondant aux solutions admettant un comportement multiforme sur le chemin d'intégration confirme bien le comportement attendu des solutions, en fonction des conditions initiales. Une fois que ces comportements sont identifiés, l'utilisateur est capable de modifier les paramètres de l'intégration afin d'améliorer les résultats de l'intégration.

10.3.3 Un exemple de feedback utilisateur

L'intervention d'un utilisateur lorsqu'il est confronté à des solutions de nature multiforme peut s'effectuer essentiellement de deux manières :

Re-spécification des chemins d'intégration cela permet d'éviter les zones où les singularités de l'équation peuvent éventuellement se trouver (et donc d'apporter une amélioration de la qualité des résultats produits par l'intégration), ou au contraire de déterminer le comportement des solutions calculées autour de singularités (et donc d'apporter des renseignements sur la nature des singularités).

Re-spécification des ensembles de conditions initiales pour un chemin d'intégration donné, cela permet d'étudier les zones où les conditions initiales sont admissibles (dans une optique d'améliorations de résultat), ou au moins de fournir une carte, fonction des conditions initiales, où la solution admet un comportement multiforme (cela peut se rapprocher des *clearcuts* du chapitre 4).



TAB. 10.5: Un exemple de *feedback* utilisateur

La planche 10.5 montre un exemple de re-spécification de chemin d'intégration. Cette planche est organisée en deux colonnes : la colonne de gauche montre le résultat d'une intégration ayant entraîné la détection de solutions à caractères multiformes. Cela est visible sur le zoom (en haut de la planche) : les courbes de couleurs rouges et de couleurs vertes sont échangées après intégration, c'est-à-dire que les fonctions solutions y correspondant n'ont pas les mêmes valeurs aux extrémités de la courbe d'intégration, qui est pourtant une courbe fermée (il s'agit encore une fois d'un cercle). Cela peut être aussi vérifié sur la figure du bas, qui représente le chemin d'intégration, avec l'ensemble des singularités lorsque y_0 varie. L'intersection entre ces deux courbes implique la présence d'une singularité *à l'intérieur* du cercle d'intégration, et donc explique le phénomène des solutions multiformes.

Pour les raisons expliquées ci-dessus, il peut être intéressant, au moins d'un point de vue expérimental, de modifier les paramètres d'intégration, et en particulier le chemin d'intégration, afin notamment d'améliorer la qualité des résultats. Pour cela, l'utilisateur doit spécifier un nouveau chemin d'intégration. Nous avons choisi dans l'expérience décrite sur la planche 10.5 d'effectuer cette nouvelle spécification point par point, c'est-à-dire que l'utilisateur a cliqué à la souris sur l'écran les points par où doit passer le nouveau chemin d'intégration.

Les nouveaux points ne sont pas intégrés directement dans le nouveau chemin d'intégration : en effet, les segments reliant ces nouveaux points sont ré-échantillonés, afin de préserver la longueur moyenne des segments du chemin initial. Ce ré-échantillonage est nécessaire dès que des comparaisons doivent être faites entre les valeurs obtenues avant et après spécification : en particulier, si une estimation d'erreur est incluse dans le processus, elle dépend – entre autres – de la longueur des pas d'intégrations, et si les nouveaux segments n'ont pas à peu près la même longueur que les segments initiaux, la nouvelle estimation d'erreur ne pourra en aucun cas être comparée à l'ancienne.

Les résultats de la nouvelle intégration sont présentés sur la colonne de droite de la planche 10.5. Le nouveau chemin d'intégration possédant le même point de départ que l'ancien (x_0 ne changeant pas), et les y_0 étant les même que dans l'intégration précédente, l'ensemble des singularités lorsque y_0 varie ne change pas non plus. Le nouveau chemin d'intégration ne coupe donc pas cet ensemble des singularités. Comme on peut s'y attendre, la totalité des courbes solutions ont leurs extrémités qui ont la même projection sur le plan $t = 0$, ce qui signifie que les valeurs d'une fonction solution aux extrémités du chemin d'intégration sont bien les mêmes. Les solutions ne sont donc pas multiformes sur le chemin d'intégration.

Ces modifications pourraient être facilement automatisées. Pour cela, il suffit de repérer de manière automatique les courbes critiques, c'est-à-dire celles correspondant aux solutions multiformes sur le chemin d'intégration : il suffit pour cela de tester les valeurs de la solution aux extrémités du chemin d'intégration. Ensuite, il suffit de repérer les zones critiques du plan complexe, c'est-à-dire celles pour lesquelles les singularités ont une influence non-négligeable sur le comportement de la solution. Cela pourrait se faire par une estimation d'erreur. Ensuite, une spécification automatique du nouveau chemin d'intégration pourrait permettre au chemin d'intégration d'éviter

ces zones critiques, et donc d'améliorer la qualité des résultats produits.

10.3.4 Remarques

Échecs de l'intégrateur

L'intégrateur échoue pour certaines valeurs des conditions initiales, notamment celles pour lesquelles les singularités se trouvent *sur* le chemin d'intégration. En effet, en de tels points, l'évaluation de la fonction correspondant à la solution n'est pas possible (la fonction n'est pas définie en $\sigma_i(x_0, y_0)$). En général, de tels échecs permettent de repérer des changement éventuels de comportement de la solution. De plus, lorsque la variable d'intégration x s'approche d'une singularité, la qualité du résultat numérique devient incertaine [Aid99]. Il est donc particulièrement intéressant de détecter ce genre de comportements lorsque ceux-ci se produisent.

Visualisation

Les différentes figures proposées sur les planches 10.4 et 10.5 ont toutes été produites sous formes de courbes, et non pas de surface. Bien que la visualisation d'un ensemble de solutions d'une équation différentielle correspondant à un ensemble de conditions initiales (le flot du système dynamique réel de dimension deux associé, via le chemin d'intégration) soit plus agréable sous forme de surface, ce modèle n'est plus d'une très grande utilité lorsque les courbes correspondent à des solutions présentant un caractère multiforme.En effet, dans de tels cas, les orbites sont particulièrement tourbillonantes, et les surfaces qui sont composées à partir de ces courbes sont particulièrement difficiles à visualiser : les polygones qui les composent se trouvent inversés, et les courbes correspondant à des singularités proches du chemin d'intégration ont des comportements très perturbés (avec de grandes variations en modules et en arguments), ce qui rend peu significative leur interpolation par des polygones.

Chapitre 11

Visualisation de l'erreur globale

L'intégration d'une équation ou d'un système différentiel par une méthode numérique n'est intéressante qu'à condition d'avoir des moyens pour en controler la validité. Le calcul de différentes sortes d'erreurs en est un. Le contrôle de l'erreur locale[HNW87] (c'est-à-dire l'erreur commise sur un pas) est généralement utilisé au cours d'un processus d'intégration, mais ses valeurs ne sont pas forcément significatives et n'apportent généralement que peu de renseignements sur la validité globale d'une intégration. L'erreur *globale*, c'est-à-dire la différence entre la solution de l'équation différentielle et sa valeur approchée, est dans ce cas beaucoup plus adaptée.

Une estimation de l'erreur globale commise au cours d'une intégration est calculable, par un ensemble de techniques présenté dans [Ske86]. Certaines de ces techniques sont utilisables au cours de l'intégration, et le surcoût lié à leur calcul peut être important (l'estimation de Richardson nécessite une autre intégration de l'équation différentielle). Cependant, les renseignements que l'on peut en tirer peuvent être précieux, comme par exemple en calculant une *localisation* de l'erreur, c'est-à-dire le calcul des zones du chemin d'intégration où l'erreur a un module élevé, et donc où la solution calculée n'est pas forcément proche de la solution exacte d'une équation différentielle. La détermination de ces zones permet, grâce à des opérations simples, d'améliorer la qualité des résultats numériques d'une intégration, notamment grâce à des modifications des paramètres d'intégration.

11.1 Principe de l'estimation de l'erreur globale

De nombreuses méthodes existent pour estimer l'erreur globale commise pendant une intégration : la méthode de Richardson [CM84] et la méthode de Zadunaisky [Zad76] sont certainement les plus connues. Nous exposerons ici le principe de l'estimation de Richardson.

11.1.1 Cadre d'application

Considérons l'équation différentielle

$$y' = f(x, y) \tag{1.1}$$

où la variable d'intégration x appartient à une courbe C. Considérons de plus que y possède une valeur initiale y_0. L'intégration numérique de cette équation avec une méthode à pas variable (ici, le choix des différents pas que nous noterons h_i est contraint par le fait que la variable d'intégration doit rester sur le chemin d'intégration) fournit un certain nombre M de valeurs y_i, qui sont des approximations de $y(x_i)$, où les x_i sont des points de C.

La quantité

$$E_i = y(x_i) \Leftrightarrow y_i \tag{1.2}$$

est appelée *erreur globale* commise pendant l'intégration au point x_i. Si on considère de plus une autre suite $\hat{E}_i$, on dit que $\hat{E}_i$ est une estimation *valide* d'ordre relatif $r > 0$ de E_i si la relation suivante est vérifiée :

$$E_i = \hat{E}_i(1 + \mathcal{O}(H^r))$$

où $H = \max_i |h_i|$.

11.1.2 L'estimateur de Richardson

Considérons que l'équation (1.1) est intégrée avec une méthode à pas constant, h. Les valeurs de l'erreur peuvent être estimées à chaque pas d'intégration, en effectuant un calcul en trois étapes. Pour cela, supposons connues les valeurs x_i et y_i :

1. calcul de la valeur de y_{i+1} à partir de x_i et y_i, avec le pas h.
2. calcul de la valeur de $y^*_{2(i+1)}$ à partir de x_i et y^*_{2i}, avec le pas $\frac{h}{2}$ (cette étape correspond en fait à deux intégrations succesives avec le pas $\frac{h}{2}$). La valeur y^*_0 est choisie égale à y_0.
3. calcul de la valeur de
$$R_{i+1} = \frac{y_{i+1} \Leftrightarrow y^*_{2(i+1)}}{1 \Leftrightarrow 2^{-p}}$$

Les valeurs R_i ainsi calculées constituent une *estimation valide d'ordre relatif* 1 de l'erreur d'intégration [Hen62]. Cet estimateur de l'erreur globale est appelé *estimateur de Richardson.*

L'intérêt principal de cet estimateur, notamment par rapport à d'autres types d'estimateurs, est de donner une estimation rapide et néanmoins précise de l'erreur globale d'intégration [SW76, AL97]. Le surcoût associé à cette estimation n'est pas trop pénalisant (cela revient à mener deux intégrations en parallèle, et donc multiplie par trois le coût d'une intégration). De plus, dans le cas où le pas d'intégration n'est pas constant, l'estimateur de Richardson a encore un bon comportement. En effet, dans ce cas, la relation

$$R_i = E_i(1 + \mathcal{O}(H))$$

est valide, avec $H = \max_i |h_i|$, où les h_i représentent les pas d'intégration. Ainsi, nous disposons d'une solution simple à mettre en oeuvre pour estimer l'erreur globale, qu'il ne reste plus qu'à visualiser.

En général, l'estimateur de Richardson n'est jamais décrit dans la littérature que pour des intégrations réelles. Le résultat précédent concernant les intégrateurs à pas variable permettent de l'adapter facilement aux intégrations complexes.

11.2 Visualisation de l'erreur globale

11.2.1 Intérêt de la visualisation

La visualisation d'une estimation de l'erreur globale commise au cours d'une intégration est intéressante dans l'optique notamment d'une caractérisation visuelle de la qualité d'une intégration. En effet, l'erreur estimée pendant un processus d'intégration peut être vue comme l'approximation aux points x_i d'une fonction représentant l'erreur d'intégration. Cette fonction est *a priori* définie sur le chemin d'intégration, et l'intérêt de la visualisation est ici de comprendre comment cette fonction varie lorsque la variable d'intégration parcourt le chemin d'intégration.

Les principales applications de cette méthode de visualisation sont de repérer les extrema de la fonction, afin de pouvoir déterminer au moins visuellement quelles valeurs calculées par un intégrateur sont numériquement fiables. En effet, si le module de l'erreur prend des valeurs importantes pendant un processus d'intégration, les valeurs calculées pendant l'intégration correspondant à ces valeurs importantes du module de l'erreur ne sont pas forcément fiables, et donc les valeurs qui seront calculées postérieurement ne le seront pas non plus. Il est donc important de visualiser, en même temps que les solutions numériques issues d'un processus d'intégration, une estimation de l'erreur afin d'avoir une idée de la qualité numérique des valeurs numériques produites par un intégrateur.

De plus, la fonction représentant l'erreur étant définie sur le chemin d'intégration, il peut être intéressant de *localiser* cette erreur. Tout d'abord, la localisation peut se situer dans le temps, c'est-à-dire que les valeurs estimées de l'erreur peuvent être corrélées au paramètre du chemin d'intégration. Cela conduit dans un premier temps à un raffinement de la discrétisation temporelle du chemin d'intégration dans les zones où le module de l'erreur prend des valeurs importantes, afin de rendre les résultats numériques plus précis. Dans un deuxième temps, ces valeurs peuvent être corrélées à la position de la variable d'intégration sur le chemin d'intégration : cela correspond à une localisation spatiale de l'erreur d'intégration. Ainsi, l'influence de la proximité d'une singularité au chemin d'intégration peut être étudiée grâce à un mécanisme d'estimation d'erreur. Dans une optique d'amélioration des résultats, une modification (légère, pour que les résultats puissent être comparés) du chemin d'intégration peut être effectuée, de manière à ce que celui-ci reste éloigné d'une singularité et à ce que l'erreur estimée pendant l'intégration reste *en dessous* d'une certaine tolérance.

Ces différentes applications de la visualisation d'une estimation de l'erreur globale commise pendant une intégration peuvent donc se révéler importantes pour améliorer

la qualité numérique des résultats de l'intégration, mais aussi pour mieux comprendre le comportement d'une solution. Il est donc naturel de coupler cette notion d'estimation d'erreur avec les méthodes de visualisation illustrées dans le chapitre 10. Cependant, cette estimation n'est pas une *borne* de l'erreur commise pendant l'intégration, et donc les résultats sont à prendre au conditionnel.

11.2.2 Difficultés

La visualisation d'une estimation de l'erreur globale commise pendant une intégration présente plusieurs difficultés. La première tient au fait que cette visualisation doit représenter une fonction complexe, et que ses valeurs doivent être visualisées *en même temps* que celles de la solution. De plus, les variations du module de l'erreur doivent être corrélées au comportement global de la solution associée, et donc une méthode de visualisation conjointe de ces deux quantités doit être trouvée pour en exploiter tous les renseignements.

La seconde difficulté tient essentiellement à la nature des résultats numériques donnés par l'estimation de l'erreur. En effet, durant une intégration, les ordres de grandeur présentés par l'estimation de l'erreur globale peuvent être très variables : au début de l'intégration, l'erreur globale est de l'ordre du minimum machine, alors que pendant une intégration le rapport entre l'erreur estimée et la solution calculée peut être de l'ordre de 1, auquel cas la solution calculée n'est en général plus significative par rapport à la solution exacte. Ces variations d'ordre empêchent en général de représenter naïvement l'erreur globale estimée pendant une intégration par les valeurs de son module, et ainsi une méthode de visualisation de l'erreur globale estimée pendant une intégration doit prendre en compte ces caractéristiques des valeurs numériques.

11.2.3 Réalisation pratique

La solution des problèmes de visualisation de l'erreur globale passe par l'utilisation du principe des données primaires et secondaires illustré en 10.2.2. En effet, ce principe permet de visualiser une fonction de la variable complexe de manière conjointe avec le module ou l'argument d'une autre fonction, donnée de manière discrète. Ce principe résout donc la première difficulté, et de plus la seconde difficulté impose de choisir les valeurs estimées de l'erreur globale comme données secondaires. En effet, la visualisation du module de l'erreur en tant que donnée primaire entraînerait des écarts importants d'échelle à prendre en compte, et de plus ses valeurs en tant que telles ne sont intéressantes qu'avec la donnée des valeurs de la solution.

Pour que les valeurs de l'erreur globale puissent être visualisées en tant que données secondaires d'une expérience, il faut qu'elles soient stockées dans un objet de classe dérivée de la classe `MultiRep`. Pour cela, la classe `Error_Rep` a été implémentée, et elle dérive directement de la classe `MultiRep`. Le champ des données de cette classe est rempli *pendant* l'intégration, c'est-à-dire que si l'intégrateur utilisé pour une expérience possède une méthode d'estimation de l'erreur, le résultat de cette es-

timation est stocké au niveau de la classe `Error_Rep`. Ainsi, ces données vont pouvoir être exploitées en tant que données secondaires d'un tracé.

Pour effectuer l'échantillonage des valeurs de la donnée secondaire d'un tracé, afin que ces données puissent être exploitées sous forme de couleurs, il est nécessaire de définir une application de quantification de ces données. Deux telles applications sont introduites dans le chapitre 10.2.2, et chacune d'elles peut être utilisée pour la visualisation de l'erreur globale :

1. L'application $\mathcal{Q}$ définie par

$$\mathcal{Q} : [e_{min}, e_{max}] \Leftrightarrow [0, couleurmax]$$

   où e_{min} (resp. e_{max}) représente le minimum (resp. le maximum) des valeurs estimées de l'erreur globale sur *une* courbe résultat d'une intégration. Ainsi, chaque courbe possède sa propre application de quantification. L'utilisation typique de ce genre d'application est pour repérer si les maximum du module de l'erreur ont une position fixée sur l'intervalle de paramétrisation du chemin d'intégration : cela permet entre autres de détecter si l'erreur globale est cumulée pendant l'intégration, par exemple si le comportement de l'intégrateur se dégrade lorsque le paramètre du chemin d'intégration croît.

2. L'application $\mathcal{Q}^*$ définie par

$$\mathcal{Q}^* : [e^*_{min}, e^*_{max}] \Leftrightarrow [0, couleurmax]$$

   où e^*_{min} (resp. e^*_{max}) représente le minimum (resp. le maximum) des valeurs estimées de l'erreur globale sur *la totalité* des résultats d'une intégration. Cette méthode de quantification *globale* permet de repérer les zones du chemin d'intégration où le module de l'erreur est maximal, ceci pour toutes les conditions initiales. Cela est particulièrement utile lorsque plusieurs solutions (correspondant à plusieurs conditions initales) sont à étudier.

11.3 Exemples

11.3.1 Visualisation de l'erreur globale

Les deux méthodes de visualisation correspondant aux deux applications ci-dessus ont été implémentées. Les résultats graphique sont présentés sur la planche 11.1, page 199

Figure 11.1 Cette figure illustre la première méthode de visualisation, correspondant à un échantillonage du module des valeurs de l'erreur globale associée à chaque courbe solution. Elle illustre aussi les principales difficultés rencontrées avec cette méthode de visualisation dès lors que cette méthode de visualisation est appliquée à un ensemble de courbes. Sur la partie droite de la figure, les zones de la surface où

le module de l'erreur est important sont bien marquées, mais cependant il est impossible de comparer l'erreur estimée pendant une intégration particulière et celle estimée pendant une autre intégration. Par contre, cette méthode de visualisation donne ici une indication sur la propagation de l'erreur pendant les intégrations : l'erreur atteint son maximum en des altitudes différentes pour chaque courbe correspondant aux différentes conditions initiales, ce qui semble indiquer qu'il existe une singularité mobile pour ce système.

La partie gauche de la figure par contre montre les limitations de cette méthode : les taches sur cette figure sont complètement artificielles. En effet, pour ces intégrations, le module de l'erreur estimée est quasimment constant, ce qui fait qu'une infime variation introduit des changements de couleurs des points de la surface.

Figure 11.2 Cette figure illustre l'emploi de la deuxième méthode de quantification, afin notamment de détecter les extrema globaux du module de l'erreur estimée. Les taches vertes sur cette figure correspondent aux points de la surface correspondant aux valeurs maximales du module de l'erreur. Cette méthode de visualisation est aussi appelée *localisation temporelle* [ATV97], car elle permet de repérer dans quels sous-intervalles de l'intervalle paramétrant le chemin d'intégration le mécanisme d'estimation de l'erreur globale donne des résultats de module particulièrement important. De plus, cette méthode permet de repérer quelles conditions initiales ont produit des solutions avec une erreur relativement importante, et donc de déterminer des zones sensibles du plan complexe dans lesquelles une condition initiale donne des solutions numériques moins fiables que dans d'autres régions.

11.3.2 Localisation spatiale

La méthode dite de *localisation spatiale* permet de détecter, pour un ensemble de solutions correspondant à différentes conditions initiales, les points du chemin d'intégration pour lesquels le module de la valeur correspondante de l'erreur globale est supérieur à une valeur seuil fixée. En considérant l'erreur globale comme une fonction définie sur le chemin d'intégration, cela permet de repérer les zones du plan complexe où cette fonction possède un module élevé, et donc où la solution calculée par l'intégrateur est de mauvaise qualité numérique.

Pour cela, il faut déterminer l'ensemble

$$\{x \in \mathcal{P}, \exists y_0 \in \gamma, |\hat{E}_i(y_0)| > \delta\}$$

où δ est un nombre réel fixé dans l'intervalle $[e^*_{min}, e^*_{max}]$. La valeur $\hat{E}_i(y_0)$ est une valeur estimée de l'erreur globale calculée lors de l'intégration avec comme valeur initiale $y(x_0) = y_0$, à l'itération numéro i. Une des applications de cette méthode de localisation spatiale est de comprendre l'influence des singularités sur la qualité numérique des résultats de l'intégrateur.

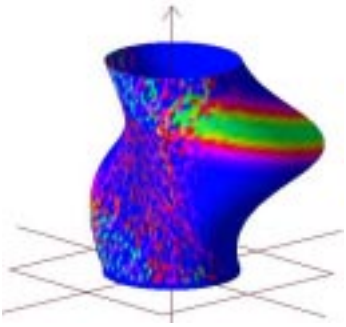


FIG. 11.1: Première méthode de quantification

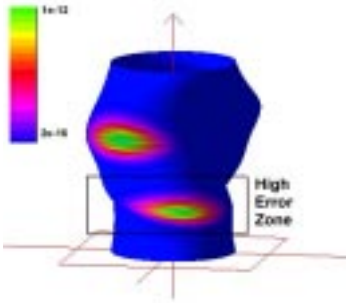


FIG. 11.2: Deuxième méthode de quantification

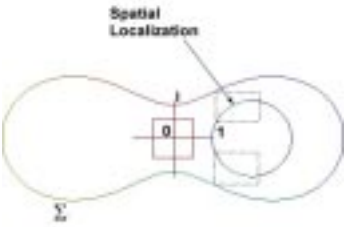


FIG. 11.3: Localisation spatiale de l'erreur

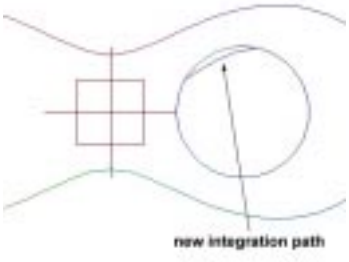


FIG. 11.4: Nouveau chemin d'intégration

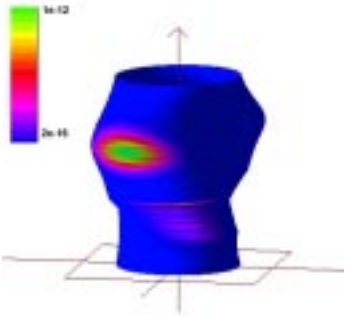


FIG. 11.5: Erreur calculée sur le nouveau chemin

TAB. 11.1: Visualisation de l'erreur globale

Figure 11.3 Cette figure montre la localisation spatiale de l'équation (1.1) définie dans le chapitre précédent :

$$y' = xy^3$$

Cette localisation spatiale a été calculée avec une valeur seuil de δ proche de e^*_{max}, afin de bien séparer les valeurs extrêmes de l'erreur de ses valeurs plus négligeables. Les points détectés par cette méthode sont portés directement sur le chemin d'intégration (et sont encadrés sur la figure). Sur cette figure apparaissent deux zones critiques pour l'intégration. Cette figure est à rapprocher de la figure 11.2, qui présentait deux taches correspondant à des pics du module de l'erreur : ces deux taches correspondent à des valeurs du module de l'erreur supérieures à la valeur seuil δ.

La seconde courbe notée Σ sur la figure 11.3 représente la position de la première singularité de l'équation (1.1) lorsque y_0 varie :

$$\sigma_1 = \sqrt{x_0^2 + \frac{1}{y_0^2}}$$

Cette singularité est une singularité *mobile* de l'équation (1.1), et sa position dans le plan complexe lorsque y_0 varie dans γ a été étudiée en 10.3.

Il est intéressant de remarquer sur cette figure que la proximité de Σ par rapport au chemin d'intégration semble être une cause de l'augmentation du module de l'erreur globale estimée pendant l'intégration. En effet, pour certaines conditions initiales, les points $\sigma_1(x_0, y_0)$ de Σ correspondant à ces valeurs de y_0 minimisent la distance de Σ au chemin d'intégration : ces points correspondent de plus aux points de la localisation spatiale. Il peut donc être intéressant d'étudier l'influence d'une re-spécification du chemin d'intégration en vue d'éloigner ce chemin de l'ensemble Σ, notamment pour voir l'influence de la proximité des singularités sur l'estimation de l'erreur globale.

11.3.3 Re-spécification des paramètres d'intégration

La re-spécification des paramètres d'intégration peut d'effectuer de plusieurs manières :

- de la même manière qu'avec les équations différentielles de la variable réelle, notamment par une modification des tolérances de l'intégrateur, ou par la perturbation des conditions initiales.
- par la modification du chemin d'intégration, méthode spécifique aux intégrations dans le champ complexe.

La seconde méthode a été appliquée à l'exemple de l'équation (1.1), comme sur l'exemple de la figure 11.4.

Figure 11.4 Cette figure montre un exemple de re-spécification du chemin d'intégration, de manière à ce que celui-ci reste éloigné de l'ensemble Σ. Ce test doit permettre de vérifier si la proximité des singularités mobiles de l'équation a effectivement une influence sur l'estimation de l'erreur. Pour cela, un nouveau chemin est

spécifié (par l'utilisateur) de telle manière à ce que ce nouveau chemin *s'éloigne* de l'ensemble Σ. De plus, le nouveau chemin n'a pas été modifié dans la deuxième région de la localisation spatiale, celle correspondant aux arguments de la variable d'intégration autour de $\frac{2\pi}{3}$: cela devrait permettre de vérifier que l'erreur correspondant à ces valeurs de la variable d'intégration reste bien au même niveau qu'avec l'ancien chemin d'intégration.

Figure 11.5 Cette figure montre la nouvelle erreur estimée pendant le processus d'intégration sur le nouveau chemin. La première tache a disparu, ce qui signifie que cette fois l'erreur commise sur la partie spécifiée du nouveau chemin d'intégration reste faible par rapport à son ancienne valeur. La seconde tache est encore présente, ce qui signifie que la présence d'une singularité de l'équation a encore une influence non négligeable sur l'estimation de l'erreur globale, notamment dans la partie non modifiée du chemin d'intégration, contenant la deuxième partie de la localisation spatiale.

Notons que cette diminution du module de l'estimation de l'erreur globale n'est *a priori* pas artificiellement provoquée par une diminution de la longueur moyenne des segments composant le chemin d'intégration : en effet, lorsque l'utilisateur spécifie un nouveau segment pour le chemin d'intégration, ce segment est rééchantilloné de manière à ce que la longueur moyenne des segments composant le nouveau chemin d'intégration soit du même ordre que la longueur moyenne des segments composant le chemin d'intégration initial.

11.4 Conclusion

Les exemples présentés ici montrent l'influence que peut avoir la proximité d'une singularité sur l'estimation de l'erreur pendant une intégration. Les techniques d'estimation d'erreur, bien que ne donnant que peu de garanties sur la véritable erreur d'intégration (c'est-à-dire que l'estimation elle-même peut ne pas se révéler tout à fait fiable [ATV97]), donnent cependant une idée de la qualité numérique des résultats d'une intégration. Cette série d'exemple montre aussi que, dans le cas des équations différentielles de la variable complexe où le chemin d'intégration peut être déformé, l'erreur globale calculée pendant le processus d'intégration peut être rabaissée en dessous d'un certain seuil. Ce point permet par exemple d'accélérer le processus d'intégration, car de manière classique avec les méthodes à pas variable, la longueur du pas d'intégration dépend de l'estimation de l'erreur globale calculée à chaque itération.

Il faut cependant différencier cette série d'exemple des cas généraux : ici, la position des singularités mobiles est connue, de même que la solution de l'équation, ce qui n'est généralement pas le cas. En particulier, le chemin a été déformé de manière à ce qu'il s'éloigne des singularités, ce qui est fait facilement si la position des singularités est connue à l'avance.

La donnée de l'erreur globale est ici corrélée à la position des singularités. Gé-

néralement, le but n'est pas d'éloigner le chemin d'intégration des singularités, mais de faire globalement diminuer l'erreur estimée pendant une expérience. Dans ce cas, quelques essais de déformation du chemin d'intégration peuvent permettre cette diminution, et donc globalement faire diminuer l'erreur d'intégration durant le processus. Le but ultime serait même, étant données des conditions initiales, de montrer l'influence de la déformation du chemin d'intégration sur la qualité numérique de l'intégration. En particulier, il serait intéressant de montrer à l'utilisateur les zones que le chemin d'intégration doit éviter pour garder une estimation de l'erreur aussi basse que possible durant l'intégration.

Ici, la procédure de modification du chemin d'intégration est faite manuellement, mais ce procédé pourrait être fait de manière automatique. Pour cela, il suffirait d'établir une carte dans un voisinage du chemin d'intégration initial, carte obtenue en intégrant l'équation différentielle sur des chemins obtenus à partir du chemin d'intégration initial par des déformations, et en estimant l'erreur commise par ces intégrations. Ensuite, un chemin d'intégration pourrait être établi, étant donnés quelques points d'ancrage définis à l'avance, de manière à ce qu'il évite les zones sensibles repérées par la carte. Ceci dit, la manipulation manuelle a l'avantage d'aider à la compréhension des phénomènes entraînant de mauvais résultats numériques.

Un autre problème rencontré pendant cette série d'expériences est la difficulté de trouver un compromis entre la lisibilité des surfaces résultats de l'intégration et la pertinence des valeurs estimées de l'erreur globale. En effet, nous avons discrétisé ici le chemin d'intégration avec une centaine de segments, et la faible longueur de ces segments donne une erreur globale assez faible, relativement aux valeurs calculées de la solution de l'équation. Pour avoir des valeurs plus significatives, il faut considérer une discrétisation plus grossière du chemin d'intégration, mais cela nuit à la compréhension globale des résultats graphiques de l'expérience.

Pour finir, nous pouvons mettre en avant le rôle joué par la visualisation étroitement couplée aux calculs : tout d'abord, l'utilisation de bonnes méthodes de visualisation aide à comprendre le caractère global de l'expérience, et de plus cela permet de comparer facilement et rapidement diverses expériences, grâce à l'interactivité du processus. Bien entendu, ce procédé mériterait d'être appliqué à d'autres équations, en particulier des équations dont peu de renseignements sont connus (tels que la position dans le plan complexe des singularités).

Conclusions

Conclusions

Bilan

Objectifs Initiaux

Le but de ce travail était initialement de mieux comprendre les phénomènes qui peuvent apparaître en analyse complexe. Pour cela, nous avons approfondi deux points distincts et complémentaires : les problèmes liés aux *calculs* avec des nombres complexes et ceux liés à la *visualisation* des fonctions complexes de la variable complexe.

- **Calculs avec des nombres complexes**

Les environnements de calcul permettant d'utiliser des nombres complexes ne sont généralement plus valides dès que les calculs dépassent le stade des évaluations de fonctions élémentaires. Ils ne permettent pas notamment d'évaluer les fonctions multiformes sur des chemins de manière à en obtenir des déterminations continues. Ces difficultés empêchent d'intégrer facilement des équations différentielles de la variable complexe définies par des fonctions multiformes, alors que cela peut être possible théoriquement.

Nous nous sommes particulièrement intéressés à deux aspects du problème : premièrement, l'utilisation des surfaces de Riemann pour améliorer *qualitativement* les calculs avec des nombres complexes. Cela a mené à la définition d'un modèle de programmation permettant de gérer les indéterminations liées aux éléments d'une surface de Riemann. Par ailleurs, nous nous sommes intéressés à la qualité *numérique* des calculs menés avec des nombres complexes, ce qui a conduit à la prise en compte de notions telles que l'erreur globale d'intégration.

- **Visualisation des fonctions complexes**

La visualisation des fonctions issues de l'analyse complexe n'apporte généralement que peu de renseignements, car les fonctions complexes sont d'abord difficiles à visualiser complètement. De plus, les méthodes de visualisation présentes dans les principaux systèmes de calcul (formel ou numérique) sont trop générales, et empêchent la visualisation efficace des fonctions de la variable complexe issues d'un problème particulier.

Nous avons choisi de nous limiter à la visualisation des solutions d'équations différentielles de la variable complexe. Les méthodes qui existaient précédemment pour les solutions d'équations différentielles linéaires sont efficaces, car elles permettent de saisir le comportement d'une solution au voisinage d'une singularité. Elles ne sont cependant pas spécialement adaptées à la visualisation de solutions d'équations différentielles non-linéaires, dont le comportement doit être visualisé non seulement au voisinage de certains points, mais aussi dans sa globalité, et lorsque les paramètres d'intégration peuvent changer.

La conception d'un outil de visualisation générique s'est vite avérée importante, et les différentes méthodes de visualisation définies dans ce document utilisent toutes cet outil. De plus, l'interactivité proposée par cet outil de visualisation est vite devenue un point clé dans la résolution des problèmes de calculs énoncés précédemment.

Points Marquants / Apports

• Modèle de programmation

Un premier effort a été fait dans la conception d'un modèle de programmation adapté aux calculs en nombres complexes. Ce modèle est basé sur la notion d'*indétermination*, telle qu'elle peut être définie par la théorie des fonctions multiformes. Le principe consiste essentiellement à séparer la partie générale d'un calcul de sa partie spécifique. La mise au point de ce modèle est passée par trois étapes :

- détermination des opérateurs pouvant entraîner des indéterminations,
- mise au point de moyens logiciels de détection des indéterminations au cours d'un calcul,
- mise au point de moyens logiciels de traitement à effectuer en cas d'indétermination.

D'un point de vue technique, ce modèle a été rendu possible grâce à l'utilisation des exceptions C++, ainsi que par les facilités de surcharge d'opérateurs apportées par ce langage. Ainsi, ce modèle de programmation objet permet d'apporter des moyens naturels et efficaces de gérer les indéterminations.

• Estimation de la qualité de calcul

Un deuxième effort a été fait dans la détection des erreurs *pendant le déroulement d'un calcul*. Ainsi, la détection des indéterminations à la volée décrite plus haut, de même que la visualisation des estimations d'erreur pendant les intégrations numériques d'équations différentielles permettent d'avoir en plus des résultats du calcul une idée de la qualité numérique d'un calcul. Le principe de détection des indéterminations permet d'agir sur le calcul au moment où une indétermination se produit, et donc sans avoir à recommencer l'intégralité du calcul. La visualisation des données, en plus de renseignements de type localisation d'erreurs, que cette localisation soit sur les arguments des primitives élémentaires (cas des *clearcut regions*, chapitre 4) ou sur les résultats d'une intégration numérique (cas de la visualisation de l'erreur

globale, chapitre 11), permet dans tous les cas de modifier les paramètres initiaux d'un calcul afin d'éviter que des erreurs se produisent au cours d'un calcul suivant.

• Interactivité des expériences

L'interactivité des expériences permet ici d'apporter des améliorations à un calcul (précision et correction des résultats) grâce à une intervention humaine. Les indéterminations qui se produisent pendant un calcul peuvent être ainsi résolues par une réponse de l'utilisateur lorsqu'il n'est pas possible à l'environnement de calculs de les résoudre (chapitre 4). Des méthodes de visualisation efficaces peuvent aussi aider l'utilisateur à comprendre la nature des paramètres à changer. Cette interactivité est rendue possible par un couplage serré entre calculs et visualisation, qui lui même est facilité par la séparation de la partie visualisation d'une expérience de la partie purement calculatoire. De plus, cet utilisateur a de nombreuses méthodes à sa disposition (chapitre 10), qui peuvent être spécifiées dynamiquement (chapitre 9). Enfin, lorsque le résultat d'un calcul n'est pas satisfaisant (comme notamment lorsque l'estimation de l'erreur globale calculée pendant une intégration est supérieure à un certain seuil, chapitre 11), l'utilisateur peut facilement re-spécifier certains paramètres et relancer le calcul.

• Méthodes de visualisation

Les méthodes de visualisation introduites dans le chapitre 10 pour la visualisation des solutions d'équations différentielles permettent de mieux saisir le comportement global de la solution d'une équation différentielle. Ces méthodes de visualisation sont basées sur un principe commun (les portraits de phase étendus), et permettent de voir le comportement d'une solution en fonction du temps, de comparer plusieurs solutions sur une même figure, puis de superposer des données additionnelles aux solutions. Le formalisme des données primaires et secondaires permet aussi de visualiser de manière conjointe deux fonctions de la variable complexe, dont une est déclarée plus significative que l'autre, et ainsi de comparer rapidement les résultats de deux intégrations proches.

Enfin, ces méthodes de visualisation sont appliquées au problème de l'estimation de la qualité numérique des solutions d'une équation différentielle. Une mauvaise qualité numérique peut avoir deux causes de nature assez différentes : elle peut être due soit à des problèmes numériques (proximité d'une singularité, équations instables), soit au caractère multiforme des solutions considérées.

• Extensibilité

L'extensibilité de l'environnement de calculs et de visualisation présenté dans ce document est facilitée par sa conception modulaire (chapitre 3) : de nouveaux modules peuvent y être intégrés, que ceux-ci soient des modules de calcul ou de visualisation. En particulier, des modules issus d'autres domaines du Calcul Scientifique peuvent y être rattachés, comme cela fut le cas avec la réalisation d'un outil permettant

de composer des surfaces à partir de données complexes, issues de la thermodynamique (travail en commun avec Loïc Benayoun). De plus, les différentes interfaces de programmation présentées dans le chapitre 8 rendent ce travail particulièrement aisé, puisque les différents modules à intégrer dans l'environnement suivent tous le même modèle de programmation. Enfin, l'utilisation du chargement dynamique de code (chapitre 7) peut même rendre efficaces l'exploitation graphique des nouveaux modules, avec relativement peu de travail de programmation.

Perspectives

Perspectives techniques

Les perspectives techniques concernent essentiellement la partie serveur graphique, et il serait intéressant d'en appliquer le principe sur des machines accélérées. Ceci devrait pouvoir se faire dans des temps assez courts, tout d'abord grâce à un travail de portage du serveur GANJ sur des Silicon Graphics, puis sur du matériel plus générique, dès que les extensions en trois dimensions de Xfree86 seront disponibles au public : ceci devrait permettre de disposer d'une solution graphique complète et surtout bon marché, puisque ce serveur X fonctionnerait sur des PC, qui peuvent eux-mêmes exploiter les cartes graphiques accélératrices 3D disponibles sur le marché.

Les directions d'extension du serveur GANJ les plus intéressantes nous semblent concerner la modularité du serveur lui-même. Ainsi, la généralisation des mécanismes de chargement dynamique aux mécanismes de sélection ou de gestion d'erreur permettrait de rendre particulièrement intéressante la solution de la plate-forme, car dans ce cas un outil de calcul possédant des méthodes de visualisation spécifique peut aussi fournir des méthodes spécifiques de sélection sur les scènes qui sont composées.

Enfin, il serait intéressant de finaliser la version parallèle de l'interface de programmation du serveur GANJ utilisant le support exécutif Athapascan0$_{mp}$ [Riv97]. Une première version montrant la faisabilité de la tâche a déjà été réalisée, exploitant notamment la notion de groupe de processus. Cette interface de programmation permettrait notamment d'exploiter graphiquement des résultats issus de calculs menés sur des machines parallèles et sur des réseaux de stations.

Perspectives mathématiques

Parmi les perspectives mathématiques induites directement par le travail présenté dans le cadre de cette thèse, nous pouvons envisager des couplages entre les données issues des intégrations avec les résultats de la détermination des *clearcut regions* pour une fonction multiforme particulière, ou alors des mécanismes de composition automatique de chemins grâce à la donnée des *clearcut regions* Dans le même registre, des *differential clearcuts regions* pourraient être envisagées, qui permettraient de voir comment un numéro de feuillet est transformé après intégration sur un chemin fermé : ce principe serait à rapprocher des applications de Poincarré, en considérant le flot associé à une équation et un chemin.

Générales

Les perspectives mathématiques qui sont envisagées dans le point précédent profiteraient toutes de la notion de partage d'objets mathématiques entre plusieurs modules : si elles n'ont pas été réalisées dans le cadre de cette thèse, c'est parce que les mécanismes de partage d'objets n'ont pas encore été implémentés. Une des premières priorité d'amélioration de la plate-forme présentée ici serait donc l'exploitation de données par plusieurs modules travaillant en parallèle. Ceci passerait par la prise en compte de formats d'échanges de données mathématiques (OpenMath), échange situé au-dessus d'un environnement robuste permettant de distribuer des calculs. Un autre point qui nous paraît fondamental serait d'ouvrir l'exploitation graphique à d'autres outils, commerciaux ou shareware, afin de faciliter l'utilisation d'une telle plate-forme et de permettre à des utilisateurs potentiels ayant la chance de posséder de bons outils de visualisation de les intégrer facilement. Un dernier point serait de faciliter la sauvegarde des scènes dans un format de description répandu, VRML par exemple. Cela permettrait le partage, sur le Web par exemple, des résultats issus d'une expérience particulière. Ces résultats pourraient même être obtenus "à la demande", au vu de l'évolution d'un langage comme Java, qui permettrait d'intégrer à distance des modules et de les instancier avec des données particulières, afin qu'un utilisateur du Web puisse composer des expériences, puis exploiter visuellement ses résultats sur le Web. Notons quand même que le temps nécessaire pour la réalisation d'une plate-forme aussi générale, qui soit sécurisée et performante, devrait être particulièrement élevé...