COMPOSANTS LOGICIELS POUR LE DIMENSIONNEMENT EN GENIE ELECTRIQUE.APPLICATION A LA RESOLUTION D'EQUATIONS DIFFERENTIELLES.

Vincent Fischer

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
/_/_/_/_/_/_/_/_/_/_/_/

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : « Génie électrique »

Préparée au Laboratoire d'Electrotechnique de Grenoble

dans le cadre de l'Ecole Doctorale
« Electronique, Electrotechnique, Automatique, Télécommunication, Signal **»**

présentée et soutenue publiquement par

Vincent FISCHER

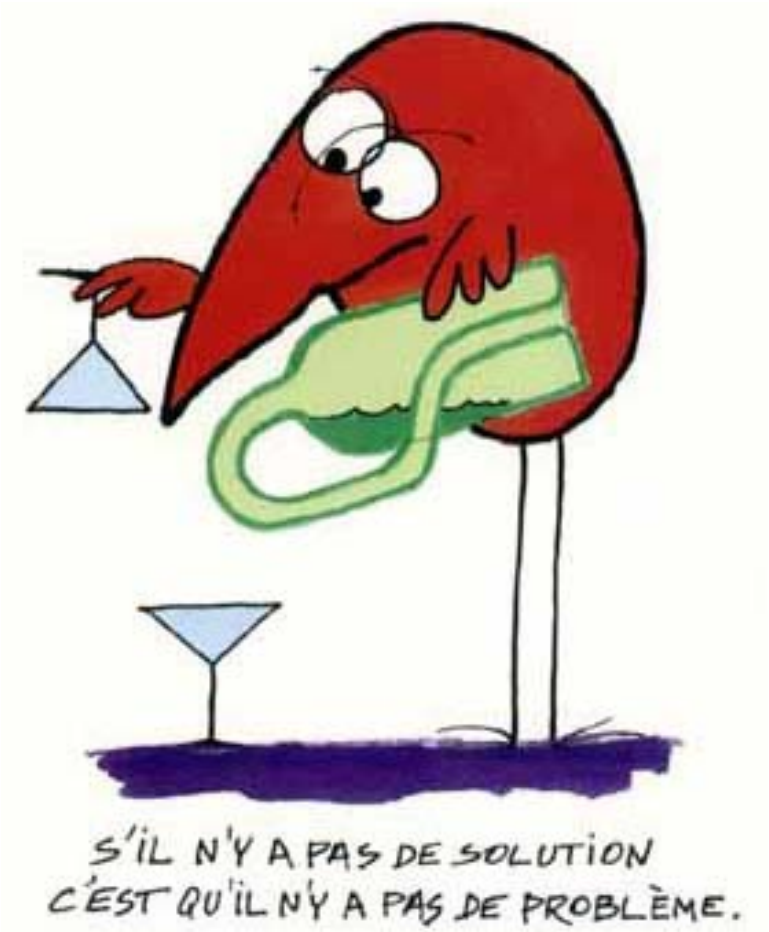
Ingénieur ENSIEG

LE 12 OCTOBRE 2004

COMPOSANTS LOGICIELS POUR LE DIMENSIONNEMENT EN GENIE ELECTRIQUE. APPLICATION A LA RESOLUTION D'EQUATIONS DIFFERENTIELLES.

Directeur de thèse : **Laurent GERBAUD**

Monsieur	JL. COULOMB	Président
Messieurs	H. PIQUET	Rapporteur
	L. NICOLAS	Rapporteur
	J. DELLA DORA	Examinateur
	L. HASCOET	Examinateur
	L. GERBAUD	Directeur de thèse



DEVISE SHADOK

A Karina,

Tu es partie trop tôt, mais tu restes à jamais dans mon coeur

REMERCIEMENTS

Le moment de conclure trois années exceptionnelles au sein du Laboratoire d'Electrotechnique de la Galaxie est venu. Je profite donc de cette occasion pour remercier les personnes sans lesquelles cette aventure n'aurait été ni possible, ni aussi formidable.

Tout d'abord, je tiens à remercier Monsieur Jean-Pierre Rognon pour m'avoir accueilli au sein du LEG et en avoir assumé la direction pendant les premiers mois que j'y ai passé. Je remercie également Monsieur Yves Brunet pour avoir pris la suite de Monsieur Rognon, ainsi que pour avoir animé les séances (parfois longues, voire très longues…) du Conseil de Labo.

Je tiens également à remercier Messieurs Piquet et Nicolas pour l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de mes travaux et de l'intérêt qu'ils y ont porté. Un grand merci également à Monsieur Jean-Louis Coulomb, qui a présidé ce jury, ainsi qu'à Messieurs Della Dora et Hascoet, qui ont eu le courage de lire ce manuscrit en quelques jours avant la soutenance.

Je tiens à exprimer ma gratitude à celles et ceux sans qui le labo marcherait sur la tête. Un grand merci à Jacqueline Delaye, Elise Riado, Monique Boizard et Danielle Collin qui assurent le fonctionnement de notre labo au quotidien. Je remercie tout particulièrement Danielle qui a toujours été là quand j'avais besoin d'un coup de main avec toutes ces paperasses qui font le charme de notre beau système. Je profite de l'occasion pour remercier le système administratif de l'INPG, toujours un grand moment à vivre… Un grand merci aussi à nos mécaniciens et électroniciens, Jean Barbaroux (le spécialiste du transfo qui chante), Claude Brun, François Blache, Stéphane Catellani, Bruno Mallet et Daniel Ogier, qui assurent le soutien technique de nos travaux. Spéciale dédicace à Djidge pour les coups de mains lors des bricolages sur la moto (belle technique pour faire sauter un Neiman…). Enfin, et tout particulièrement, merci à nos trois courageux admins du Service Info, Corine Marcon, Vincent Danguillaume et Patrick Eustache. Sans eux, le fonctionnement informatique serait plus proche du chaos que d'autre chose. Entre les coupures d'eau industrielle qui transforment leur local en une soufflerie digne de tester des voilures d'avions, et les méchants virus qui nous attaquent, ils arrivent à faire en sorte que notre réseau fonctionne tant bien que mal. Bon courage, car la situation ne vas pas aller en se simplifiant…

La gratitude et la reconnaissance que je ressens envers mon directeur de thèse sont difficiles à exprimer au travers de quelques mots, mais je vais quand même essayer. Merci à toi, Laurent, pour tout ce que tu as su m'apporter durant ces trois années que tu as largement contribué à rendre extraordinaires. Merci d'avoir accepté de diriger mes travaux. Ce ne fut assurément pas une tâche facile, surtout lors de la rédaction, durant laquelle tu as fait preuve d'une patience extraordinaire. Merci aussi pour la confiance que tu m'as accordée dans les choix scientifiques. J'ai trouvé en toi un encadrant qu'une qualité scientifique certaine et d'une humanité exceptionnelle. Merci également d'avoir assuré la direction de l'équipe CDI (et ça non plus ce n'est pas une tâche facile) ces deux dernières années. Que les dieux de la science soient avec toi…

Que l'ensemble de l'équipe CDI soit remercié, pour m'avoir accueilli en son sein, ainsi que pour les discussions et les débats passionnés que nous avons pu avoir. Les débats sur la sémantique et la syntaxe, les réunions épistémologie de Fred, les présentations équipe, les grands débats sur les graphes de calcul ou les réunions composants resteront à jamais gravés dans ma mémoire.
Merci à Jaime Fandino pour sa bonne humeur, pour sa vision décalée de la vie et de notre société, et pour toutes les discussions que nous avons eues autour d'un café. Bon chance pour la suite de tes travaux sur tes "petites machines pourries".
Merci également à Alain Bolopion pour la compagnie durant les longues heures de cours de C durant lesquelles il m'a régalé de ses récits de voyages dans de lointains pays. Merci également pour ton aide et tes conseils judicieux sur l'utilisation et l'encapsulation d'ORO, et pour les astuces en C (y'a qu'à agiter les bons gri-gris et les erreurs de compil' s'en iront). Bon courage pour la suite, c'est bientôt la quille !!
Enfin, un merci tout particulier à Fred pour toutes les discussions enflammées que nous avons eues, tout particulièrement sur les composants et la notion de service, ainsi que sur la persistance du graphe de calcul pour la dérivation de code. Fred, j'te jure, y'a des cas où c'est vraiment pas dérivable. C'est triste, mais c'est la dure loi de la mathématique numérique.
Merci à Laule, le sniper fou à Counter, pour l'accent du sud-ouest, la bonne humeur perpétuelle, l'expertise en matériel audio et vidéo haut de gamme, pour les trajets en Clio S (attention, pas 16S) quand j'étais à pied. Merci aussi pour les discussions sur l'alcool, sur les effets de l'alcool, sur les apéros, sur les différences entre Ricard et 51, sur les bienfaits de la Chartreuse (mais oui, tu comprends, c'est du médicament), sur les spécialités culinaires du sud-ouest et surtout pour les fous rires qu'elles ont provoqués. Sans toi, ces années auraient été bien tristes. Bonne chance pour la suite à Guyancourt, garde ta foi envers le TFC et continue à parler vite, mais pas trop quand même.

Merci à Nuts, notre dictateur maison, pour le tranchant du verbe, pour les cris d'animaux, pour l'animation des discussions autours des cafés (belle technique pour lancer le Laule). Merci aussi pour la collaboration sur RAMA, pour toutes les idées, les conseils et les échanges que nous avons pu avoir autours de Java. Bon courage pour la suite, et décolle vite de Cluzes, c'est pas bon le fond des vallées, il fait humide et on finit par s'enrhumer.

Merci au P'tit Ben, le tout nouveau MC, pour m'avoir supporté comme voisin de PC pendant deux ans et demi, pour les multiples collaborations sur Java, pour tous les échanges et les confrontations de points de vue que nous avons eus autours des composants, ça a toujours été enrichissant, et pour avoir fait un bon lapin à Counter. C'est quand tu veux pour un tour un moto, faut pas laisser le Bandit se recouvrir de poussière et de toiles d'araignées. Une moto faut que ça roule…

Merci à David Magot (C'est qui Alain ?), dit Psycho, pour les discussions enflammées, les théories décalées sur l'actualité, et ses points de vue éclairés sur la programmation. Un grand merci tout particulier pour la formidable alternative qu'a constitué CDI Optimizer à un monde verrouillé, clos, entravé, contraint par des gens bercés par la douce illusion d'avoir la science infuse, et d'avoir toujours raison. La science en général, et la recherche en particulier, repose sur la remise en question de ce que l'on croit savoir et sur l'ouverture d'esprit. Malheureusement certains se permettent de l'oublier et croient qu'ils sont détenteurs de l'absolue vérité. La dure loi du marché se chargera de les ramener à la réalité. Je tiens au passage à remercier le Frelon Vert, savant mélange de duplicité et de fanfaronnade assaisonné avec beaucoup de prétention, pseudo-motard et carnaval ambulant déguisé en être humain, pour m'avoir montré le côté obscur de la recherche et la voie qu'il ne fallait surtout pas suivre.

Merci au Francky, notre gars du Nord à nous. Toujours prêt pour une partie de Counter, de OuicheBase, ou de Blobby, il arrive malgré tout à faire onduler quelques modèles. Merci aussi pour les nombreux kilomètres que nous avons partagés, aussi bien en 125 (spéciale dédicace pour l'après-midi des grands cols, le Lautaret, le Galibier, la Croix de Fer et le Glandon en Virago, belle perf), qu'en gros cube. Attention quand même à pas trop faire frotter les cale-pieds du Bandit, tu vas finir par les bouffer.

Merci à Bertrand, le p'tit jeune, pour son calme qui contraste avec l'ambiance de la salle, pour ses idées en programmation et pour m'avoir aidé à formaliser mes services et mes interfaces, et pour son soutien dans les réunions composants. Promis, la calculette sera bientôt finie. N'hésite pas, la zone rouge du Fazer est à 14000 tours, t'as de l'allonge, faut en profiter (c'est quand t'es dans les tours que ça devient vraiment bon).

Merci à Jean-Mich, l'Ancien, pour toutes les discussions que nous avons eues sur tous les sujets possibles et imaginables, et pour avoir tenu bon dans la guerre des formations.
Merci à Edouard pour tous les développements qu'il a pu mener autours de la dérivation de code, de JavaDiff et par là même pour la plus-value qu'il a donné à CoreLab. Bon courage pour la thèse qui commence.
Merci aussi à Hichem pour les traductions en Arabe, à Imen pour apporter une touche de féminité dans la salle, et à Lalao pour la bonne humeur.
Merci à Sylvain, le Marseillais, pour sa vision éclairée sur le monde, son accent et sa bonne humeur.
Merci à Raph' pour les conseils en bricolage, pour ses talents de cuistot, surtout derrière un barbeuk', et les prêts de matériel (vous cherchez une disqueuse ? demandez Raph'). Merci à Guillaume V. pour les day movies, à Franck pour les tours en moto (comme quoi, un GSX-F, ça marche), à Kiki pour la simulation quotidienne de la pile électrique, à Guillaume R. pour les soirées guitare, à Goubs pour les goubseries et le vin au citron, à Babe pour les astuces sous la mule, à Martin pour la modélisation du convert', à BigBen pour la blaguasse du vendredi aprèm, à Hervé pour la compagnie en Conseil de Labo, à Rico, ancien binôme, à Mariya, Jpeg, Manuela, Malik, Adi, Olivier, Roger, Ianko, Thierry, Chabi, Valdo, Seb, Reno Raines, et tous les autres…
Merci aussi à Matthew Bellamy, Chris Wolstenholme et Dominic Howard pour l'ambiance musicale durant la rédaction.

Un grand merci tout spécial à Guile, un véritable ami, pour tout ce que nous avons partagés ces cinq dernières années, pour les tentatives de roue arrière en TDR, les glisses, les kilomètres par moins dix à en avoir les doigts gelés, les roues avant au feu rouge à faire retomber le pneu sur le capot des bagnoles, les chutes dans la boue à la Croix au dessus de la petite descente de Lancey, pour les roues arrière réussies en Fazer, pour avoir supporté les incongruités de Quelqu'un, et pour tout le reste… Ce sont des moments que je n'oublierai pas.
Merci à Gigi, le musicien infographiste, pour les nombreuses soirées autours d'un verre de rhum, pour le soutient dans les moments délicats, pour les longues discussions durant lesquelles on refait le monde, pour les soirées télé, bref pour son amitié.
Merci à Jean-Phi, le cuisinier motard, pour sa bonne humeur, ses astuces culinaires, pour les soirées DVD ou match. Merci à Gé pour les commentaires des matchs et pour son indéfectible soutien aux Verts (ça rassure de savoir qu'il y a certaines choses qui ne changeront jamais).

Enfin, je veux remercier mes parents et ma sœur, sans lesquels rien n'aurait été possible, pour leur soutien sans faille tout au long de ces années.

SOMMAIRE

SOMMAIRE

GLOSSAIRE 1

INTRODUCTION 9

CHAPITRE I : CADRE DE TRAVAIL 13

I.1. Contexte de travail : la conception en Génie Electrique 15
I.1.1. Le processus de conception 15
I.1.2. Le dimensionnement du dispositif 16
I.1.3. La conception assistée par ordinateur 17
I.2. L'optimisation sous contraintes 19
I.3. La résolution des modèles de dimensionnement 20
I.3.1. La modélisation des systèmes physiques 21
I.3.2. Le système d'état 22
I.3.2.1. Etat d'un système 22
I.3.2.2. Vecteur d'état 22
I.3.2.3. Espace d'état 22
I.3.2.4. Trajectoire d'état 22
I.3.2.5. Equation d'état 22
I.3.2.6. Exemple de modélisation d'un système par une équation d'état 23
I.3.3. Résolution du système d'état dans le cas des systèmes linéaires 24
I.3.3.1. Résolution de l'équation homogène 24
I.3.3.2. Solution complète de l'équation d'état 25
I.3.4. Résolution du système d'état dans le cas non linéaire 26
I.4. Le besoin d'un environnement pour le dimensionnement 26
I.5. Le composant logiciel 28
I.5.1. Le composant : abstraction logicielle 28
I.5.2. Code et composant 29
I.5.2.1. Réutilisation ouverte ou fermée 29
I.5.2.2. Objet et composant 30
I.5.3. Les différentes interactions avec les composants 31
I.5.3.1. La création des composants 31
I.5.3.2. La projection 32
I.5.3.3. La composition des composants 33
I.5.4. Les différents types de composants existant 34
I.6. Solution proposée 35
I.6.1. La résolution des modèles de dimensionnement 35
I.6.2. L'approche composant pour le dimensionnement 35

CHAPITRE II : LA RESOLUTION DES MODELES DE DIMENSIONNEMENT 37

II.1. La résolution symbolique des modèles 39
II.1.1. Valeur du terme intégral 40
II.1.1.1. Valeur du terme intégral pour une source constante 40
II.1.1.2. Valeur du terme intégral pour une source polynomiale 41
II.1.1.3. Valeur du terme intégral pour une source sinusoïdale 41
II.1.2. Solution générale de l'équation d'état 42
II.1.3. Evaluation de la trajectoire d'état 43
II.1.3.1. Introduction sur le calcul d'exponentielles de matrices 43
II.1.3.2. Algorithme de calcul d'exponentielle basé sur le développement de Taylor 46

II.1.3.3. Algorithme de calcul d'exponentielle basé sur les approximants de Padé 47
II.1.3.4. Amélioration des algorithmes par "scaling and squaring" 48
II.1.3.5. Importance du conditionnement des matrices 49
II.1.3.6. Conclusion sur l'évaluation de la trajectoire d'état 50
II.1.4. Calcul des critères de dimensionnement 51
II.1.4.1. Introduction sur le calcul des dérivées partielles 51
II.1.4.2. Première approche symbolique : dérivée de l'exponentielle 52
II.1.4.3. Deuxième approche symbolique : méthode de la matrice semi-circulante 53
II.1.4.4. Troisième approche symbolique : recombinaison du système d'état 53
II.1.4.5. Calcul de la valeur moyenne de la trajectoire d'état 54
II.2. La résolution numérique des modèles 55
II.2.1. Principe de la dérivation de code 55
II.2.1.1. Les fonctions représentées par des programmes 55
II.2.1.2. Modélisation d'une fonction 56
II.2.1.3. La différentiation automatique en mode direct 58
II.2.2. Différentiation automatique en mode inverse 61
II.2.2.1. Approche par le graphe de calcul 61
II.2.2.2. Approche par substitution régressive 64
II.2.2.3. Approche par dualité 66
II.2.2.4. Implémentations algorithmiques 68
II.2.2.4.1. Algorithme ReG : approche par le graphe de calcul 68
II.2.2.4.2. Algorithme ReS : approche par substitutions régressives 69
II.2.2.4.3. Algorithme ReD : approche par dualité 69

CHAPITRE III : APPROCHE COMPOSANT POUR LE DIMENSIONNEMENT 71
III.1. Vers un nouveau standard de composant 73
III.1.1. Hétérogénéité des composants 73
III.1.1.1. Hétérogénéité de constitution 74
III.1.1.2. Hétérogénéité de fonctionnement 74
III.1.1.2.1. Chargement des composants 74
III.1.1.2.2. Utilisation des composants 74
III.1.1.3. Conséquences de cette hétérogénéité 75
III.1.2. Besoins établis et émergents des composants 76
III.1.2.1. Mécanisme de chargement 76
III.1.2.2. Mécanisme d'introspection 77
III.I.2.3. Support du schizomorphisme 77
III.1.2.4. Evolutivité du standard 79
III.1.2.5. Portabilité du standard 79
III.1.3. Spécification d'un standard de composants pour la conception 79
III.1.3.1. Choix du langage de base du standard 80
III.1.3.2. Jeu d'interfaces structurant le standard 80
III.1.3.2.1. Interface Component 80
III.1.3.2.2. Interface Service 81
III.1.3.3. Norme des fichiers 83
III.1.3.4. Outils de lecture, d'écriture, et d'introspection directe 84
III.1.3.4.1. Outil d'écriture des composants 84
III.1.3.4.2. Outil de lecture des composants 84
III.1.3.4.3. Outil d'introspection directe 85
III.1.4. Spécification des services dans le cadre d'ICAr 86
III.1.4.1. Service de résolution des modèles de dimensionnement 86
III.1.4.1.1. Service de résolution des systèmes d'état 86

III.1.4.1.2. Service de résolution des modèles pour l'optimisation 87
III.1.4.2. Service de visualisation et de post-processing 89
III.1.4.3. Service de gestion des algorithmes 90
III.1.4.4. Autres services 90
III.2. Réalisation d'un environnement d'aide à la conception 91
III.2.1. Spécifications de l'environnement 91
III.2.1.1. Fiabilité, maintenabilité, évolutivité, modularité et portabilité 91
III.2.1.1.1. Fiabilité 91
III.2.1.1.2. Maintenabilité 91
III.2.1.1.3. Evolutivité 92
III.2.1.1.4. Modularité 92
III.2.1.1.5. Portabilité 92
III.2.1.2. Ouverture 92
III.2.1.3. Principe d'utilisation de CoreLab 93
III.2.2. Architecture générale 94
III.2.3. Module de génération des composants de calcul 95
III.2.3.1. Principe de fonctionnement 95
III.2.3.2. Fonctionnement de nos générateurs dédiés 97
III.2.3.2.1. Organisation générale 97
III.2.3.2.2. Analyse du modèle 97
III.2.3.2.3. Génération du composant de calcul 98
III.2.4. Module de composition 100
III.2.4.1. Différentes compositions possibles 100
III.2.4.2. Composition hétérogène 101
III.2.4.2.1. Exemple de composition hétérogène 101
III.2.4.2.2. Intérêts de la composition hétérogène 102
III.2.4.3. Architecture du module de composition 103
III.2.5. Module de projection 103

CHAPITRE IV : APPLICATIONS 105
IV.1. Présentation de CoreLab 107
IV.2. Génération des composants de résolution 108
IV.2.1. Génération d'un composant de résolution des équations différentielles 108
IV.2.1.1. Présentation de l'exemple 108
IV.2.1.2. Génération du composant 110
IV.2.1.3. Résultats obtenus 112
IV.2.2. Génération d'un composant de résolution de modèles analytiques 114
IV.2.2.1. Présentation de l'application 114
IV.2.2.2. Génération du composant 116
IV.3. Utilisation de la composition hétérogène 119
IV.3.1. Présentation de l'application 119
IV.3.2. Composition du modèle 120
IV.4. Comparaison des systèmes de différentiation automatique 121
IV.4.1. Comparaison des performances des différentiateurs 123
IV.4.2. Limites d'utilisation des différentiateurs 125
IV.4.3. Conséquences des limites d'utilisation de la dérivation de code 126
IV.5. Avantages et limites de la démarche proposée 126

CONCLUSION 129

ANNEXE I : OUTIL DE TRAITEMENT FORMEL 133

I.1. Fonctionnement général de RAMA **135**
I.2. Le modèle MOM **137**
I.3. Ecriture des règles **138**
I.3.1. Attributs du ruleset 138
I.3.2. Format des règles 138
I.3.3. Description du contexte 139
I.3.4. Description du résultat 139
I.3.5. Formalisme MOMPath 140
I.3.6. Exemple de règle 140
I.4. Fonctionnement détaillé de RAMA **141**
I.4.1. Structure et fonctionnement du parser d'expressions 141
I.4.2. Structure du ROM 144
I.4.3. Les actions élémentaires 145
I.4.4. La création de la liste d'actions à partir du résultat 145
I.4.5. Fonctionnement du moteur d'application de règles 146
I.5. Conclusion **146**
I.6. Exemple d'un fichier de règles **147**

ANNEXE II : ANALYSE INVERSE D'ERREUR DE L'APPROXIMATION DE PADE **153**

ANNEXE III : STRUCTURE XML DE DESCRIPTION DES SYSTEMES D'ETAT **159**

ANNEXE IV : MODELISATION ANALYTIQUE D'UN TRANSFORMATEUR **171**

ANNEXE V : DERIVATEUR DE CODE JAVA : JAVADIFF **177**
V.1. Utilisation de JavaDiff pour dériver du code **179**
V.1.1. Préparation d'une section de code à dériver 179
V.1.1.1. Déclaration des variables actives 179
V.1.1.2. Délimitation des sections de code actives 180
V.1.1.3. Sélection des variables dépendantes et indépendantes 180
V.1.1.4. Opérateurs et fonctions mathématiques 181
V.1.2. Procédure de modification d'un code de calcul 182
V.1.2.1. Balisage des sections de code 183
V.1.2.2. Spécification des variables actives 183
V.1.2.3. Exploitation des résultats 184
V.2. Fonctionnement de JavaDiff **185**

ANNEXE VI : GENERATION DES COMPOSANTS DE RESOLUTION **187**
VI.1. Formalisme de description des modèles analytiques **189**
VI.1.1. Description de l'exemple 189
VI.1.2. Formalisme de description 190
VI.1.2.1. Equations simples 190
VI.1.2.2. Ajout de commentaires 191
VI.1.2.3. Statut des variables 191
VI.1.2.4. Utilisation de fonctions 191
VI.1.3. Modèle du dispositif 193
VI.2. Les différents générateurs de composants **194**
VI.2.1. Générateurs JavaDiff Direct Jacobian et JavaDiff Reverse Jacobian 194
VI.2.2. Générateur RAMA Differentials 194
VI.2.3. Générateur Adol-C Jacobian 195

INDEX 197

BIBLIOGRAPHIE 203

GLOSSAIRE

GLOSSAIRE

Algorithme : Un algorithme est une méthode de résolution d'un problème énoncée sous la forme d'une série d'opérations à effectuer. La mise en œuvre de l'algorithme consiste en l'écriture de ces opérations dans un langage de programmation et constitue alors la brique de base d'un programme informatique. Les informaticiens utilisent fréquemment l'anglicisme implémentation pour désigner cette mise en œuvre.

API : Acronyme pour **A**pplication **P**rogramming **I**nterface. Une API définit la manière dont une entité informatique peut communiquer avec une autre. Dans le cas typique d'une bibliothèque, il s'agit généralement d'une liste de fonctions considérées comme utiles pour d'autres programmes. Une API en tant que telle est quelque chose d'abstrait; les entités fournissant une API étant des implémentations de celle-ci. Idéalement, il peut y avoir plusieurs implémentations pour une même API. Par exemple, sous UNIX, la librairie libc définit des fonctions de base utilisées par pratiquement tous les programmes et est fournie par des implémentations propriétaires comme par des implémentations libres, sous différents systèmes d'exploitation.

Bytecode : Le bytecode est un code de programmation qui est exécuté par un programme, généralement appelé machine virtuelle, plutôt que par le processeur matériel de la machine. La machine virtuelle traduit chaque instruction du bytecode en instructions adaptées au système d'exploitation et à l'ordinateur sur lequel elle fonctionne. Le bytecode est le résultat d'une compilation d'un langage de programmation supportant cette approche. L'intérêt principal du bytecode est que le même bytecode peut être exécuté sur des ordinateurs d'architectures et de systèmes d'exploitations différents, pourvu que la machine virtuelle adaptée fonctionne sur cet ordinateur. Le langage le plus connu fonctionnant sur ce principe est le langage Java, dans lequel le bytecode est contenu dans les fichiers .class, résultats de la compilation des fichiers sources. D'autre plateformes se développent actuellement, avec comme enjeu un bytecode unifié pour la compilation de sources dans des langage différents. La plateforme ".NET" de Microsoft a été conçue dans cet esprit.

CAO : Acronyme pour **C**onception **A**ssistée par **O**rdinateur. C'est un ensemble de logiciels et de techniques apportant des aides aux concepteurs de dispositifs, de bâtiments, … Ces logiciels permettent par exemple de créer des pièces mécaniques, de les assembler et de simuler leur

comportement. Ils permettent également la conception de bâtiments puis d'en tirer les plans facilement ou de créer une maquette en trois dimensions. Dans le cadre du Génie Electrique, on trouve des logiciels de modélisation et simulation (Matlab, Flux3D, PSpice, Flowterm, …), des logiciels de calcul (MathCad, Maple, …), des logiciels de dimensionnement (Pascosma, CDIOptimizer, CoreLab, …).

COB : Acronyme pour **C**omputational **Ob**ject. Modèle simple de composant de calcul pour le dimensionnement.

Corba : CORBA (pour **C**ommon **O**bject **R**equest **B**roker **Ar**chitecture) est une architecture logicielle, client-serveur, orientée objet, standardisée par l'OMG (Object Management Group, http://www.omg.org). Son but est de permettre à des applications développées dans des langages différents de communiquer, même si elles ne sont pas sur le même ordinateur.

Flop : Acronyme pour **Fl**oating Point **Op**eration. Cela désigne une opération arithmétique de base sur un ou deux nombres en précision flottante pour un microprocesseur. On les distingue des Binary Integer Operation, opérations sur des entiers (qui prennent donc moins de temps).
Dans le langage informatique, on se sert des flops pour caractériser la vitesse d'un processeur (en millions de flops par seconde). En algorithmique, on caractérise la longueur d'un algorithme par le nombre de flops nécessaires à son exécution.

Encapsulation : L'encapsulation pour les développeurs en informatique est l'idée de cacher l'information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet. Ainsi les propriété et axiomes associés au informations contenue dans l'objet seront assurés et validés par les méthodes de l'objet et ne seront plus de la responsabilité de l'utilisateur extérieur. L'utilisateur extérieur ne pourra pas modifier directement l'information et risquer de mettre en péril les axiomes et les propriétés comportementales de l'objet. L'objet est alors vu comme une boite noire, dont on ne connaît que les fonctionnalités, mais pas la constitution ni le fonctionnement.

Environnement graphique : Un environnement graphique, est en informatique ce qui est affiché en mode pixel à l'écran de l'ordinateur et sur lequel l'utilisateur peut agir avec différents périphériques d'entrée comme le clavier, la souris, … Des images, des animations (en 2 ou 3 dimensions), et même des vidéos peuvent être rendues à l'écran. Ce type d'interface homme-

machine s'oppose à la notion de ligne de commande où la majorité de l'interaction entre l'utilisateur et l'ordinateur se fait au clavier, sans visualisation élaborée, dans un terminal ou dans une fenêtre de terminal, comme par exemple dans le cas de l'antique DOS. En anglais, GUI est l'abréviation de Graphical User Interface, soit Interface Utilisateur Graphique. Elle s'oppose à CLI pour Command Line Interface, soit Interface en Ligne de Commande.

Héritage : L'approche objet introduit deux types d'héritage : le sous-typage (héritage d'interface) et la sous-classification (l'héritage d'implémentation). L'héritage est la faculté d'une sous-classe ou d'un sous-type d'hériter des propriétés de son parent et de les raffiner. Le sous-typage est donc le processus par lequel on restreint l'espace des valeurs du type parent, et la sous-classification est le processus par lequel on récupère et on spécialise l'implémentation.

IHM : Acronyme pour Interface Homme Machine. Bien qu'il puisse être étendu à n'importe quel moyen de contrôle d'un mécanisme, c'est un terme qui est principalement utilisée en informatique. On y fait la distinction entre les environnements graphiques qui sont des ensembles graphiques affichés sur un écran qui permettent de visualiser ce que l'ordinateur fait et les IHM en ligne de commande. Il existe cependant beaucoup d'autres types d'interfaces utilisateurs. Ainsi les premiers ordinateurs était utilisé sous forme de traitement par lots: ils étaient alimenté en entrée par des instructions encodées sur des cartes perforées et fournissaient les données de sortie sur des imprimantes. En informatique industrielle, les automates sont encore très souvent pilotés par des baies équipées de boutons poussoirs et de voyants.

Implémentation : L'implémentation d'un concept ou d'un algorithme informatique désigne sa mise en application. L'algorithme ou le concept implémentés sont généralement indépendants de bien des contraintes. Là où un algorithme va mettre l'accent sur la complexité, l'implémentation va devoir jouer sur différents tableaux :

- la plate-forme matérielle
- le langage de programmation
- la généricité
- la modularité
- la portabilité
- les performances

Certains concepts sont tellement particuliers, par leur complexité ou leurs exigences matérielles, qu'il n'en existe pas d'implémentation pendant une longue période, comme les ordinateurs quantiques, dont le concept est né en 1994 et dont les premiers balbutiements datent de 2002.

JNI : Acronyme pour **J**ava **N**ative **I**nterface. Mécanisme permettant d'exécuter du code natif en Java.

Machine virtuelle : Environnement d'exécution servant d'interface entre le système d'exploitation et des instructions en bytecode.

Manifeste : Document de bord d'un bateau ou d'un avion, répertoriant la cargaison de celui-ci. Par analogie, on retrouve cette notion relativement aux fichiers d'archives. Le manifeste est alors la liste de la "cargaison" de l'archive, c'est-à-dire son contenu.

Natif (langage) : Un langage de programmation est dit natif par opposition à un langage portable. Un langage natif est donc dépendant de l'architecture matérielle et du système d'exploitation sur lesquels il va être exécuté.

Objet (Approche) : Dans l'approche objet, les systèmes sont uniquement constitués d'entités appelées objet. Ces objets sont définis par des types (voir la théorie des types de Liskov et celle de Cardelli par exemple). Un type, en programmation orientée objet, définit de façon syntaxique et sémantique les propriétés que présenteront les objets (les valeurs) du type ; ces propriétés permettent de délimiter, de qualifier les objets. Il définit l'interface visible de l'objet au travers duquel on peut communiquer avec lui. L'implémentation de ces propriétés est ensuite fournie par une classe, structure de données qui lie à une propriété une implémentation possible.
L'implémentation des propriétés par une classe peut être représentée soit sous forme d'emplacement mémoire (champs de donnée) appelé attribut, soit sous forme de calcul (opération) appelé méthode. Selon Cook (théorie F-Bound), une classe est l'implémentation d'une famille polymorphique de types ; une classe dans ce cas n'est donc pas nécessairement l'implémentation d'un seul type.
En objet, si un type définit donc l'interface de l'objet, la classe lui fournit une implémentation. A ce titre, la classe est le moule par lequel est construit un objet. On dit alors d'un objet qu'il est une instance de telle classe.
Enfin, la notion d'héritage est caractéristique de l'approche objet.

Phases de vie du logiciel : On distingue deux phases principales dans la vie d'un logiciel. La phase de design-time, ou phase de développement, durant laquelle le logiciel est programmé et développé, et la phase de run-time, ou phase d'exécution, durant laquelle le logiciel est exécuté.

Portabilité : Pour un programme informatique, sa portabilité lui permet d'être utilisé sous différents systèmes d'exploitation (Windows, GNU/Linux, ...).
Certains langages comme Java sont entièrement portables grâce à la présence d'une machine virtuelle qui sert d'interface entre le système d'exploitation et le programme lui-même. D'autres langages sont spécifiques à une plate-forme de développement (le C#, développé par Microsoft, et qui n'est pas libre, ne peut fonctionner que sous Windows pour l'instant).

Programmation Orientée Objet : La programmation orientée objet (ou POO) est née des travaux de recherche sur l'intelligence artificielle dans les années 70-80. Elle consiste à combiner au sein d'une même structure de données, appelée Classe, opérations et données. Le concept de classe a été introduit avec le langage Simula, ceux d'encapsulation, d'héritage et de polymorphisme avec le langage Smalltalk. Les langages orientés objets les plus connus sont Java et C++. Voir Objet.

RAMA : Acronyme pour **R**ule **A**pplicator for **M**athematical **A**nalysis. Outil de traitement formel des expressions mathématiques et de génération de code à base de règles. Voir Annexe I.

Sérialisation : L'un des fondements de la Programmation Orientée Objet réside dans l'idée de réutilisation. La plateforme Java collecte pour sa part les objets en mémoire afin qu'ils puissent être accédés à partir de n'importe quelle application Java. Cependant, cette réutilisabilité n'est valable que tant que la machine virtuelle est lancée: si celle-ci est arrêtée, le contenu de la mémoire disparaît. C'est ici que la sérialisation intervient : elle permet de stocker l'état des objets, par exemple sur le disque dur, ou lors d'un transfert réseau, ainsi que la manière de recréer cet objet pour plus tard. Un objet peut ainsi exister entre deux exécutions d'un programme, ou entre deux programmes : c'est la persistance objet.

Système d'exploitation : Un système d'exploitation (SE ou OS en anglais pour Operating System) est un ensemble cohérent de logiciels permettant d'utiliser un ordinateur et tous ses éléments (ou périphériques). Il assure le démarrage de celui-ci et fournit aux programmes applicatifs les interfaces pour contrôler les éléments de l'ordinateur. Les programmes applicatifs

n'ont traditionnellement pas vocation à être considérés comme partie intégrante du système, bien que ce point de vue soit en train d'évoluer. Sur les architectures PC, de nombreux systèmes d'exploitation existent mais les deux systèmes rencontrés majoritairement sont Windows et Linux.

XML : Acronyme pour e**X**tended **M**arkup **L**anguage. Langage à balisage unifié de description de données, standardisé par le W3C (World Wide Web Consortium, http://www.w3c.org).

INTRODUCTION

INTRODUCTION

Confronté à l'accélération du rythme de vie de notre société, et à ses répercussions dans le monde scientifique et industriel, le concepteur doit faire face de plus en plus à un compromis entre réactivité et performance des solutions à fournir. Il est donc contraint de s'appuyer sur des méthodes, des environnements, et des outils de conception, afin de prendre au plus vite les décisions judicieuses. La Conception Assistée par Ordinateur propose une large gamme de logiciels permettant d'épauler le concepteur au cours des différentes phases du processus de conception.

En particulier, le dimensionnement des solutions envisagées s'appuie sur des outils de calcul numérique (simulation temporelle ou fréquentielle, simulation par éléments finis) et des modèles analytiques. Les premiers permettent d'aider le concepteur à comprendre les phénomènes physiques et ainsi affiner sa conception. Les seconds lui procurent un support pour la prise de décision et subséquemment la réactivité nécessaire, notamment durant les premières phases du dimensionnement. Dans le cadre de notre travail, nous nous intéressons plus spécifiquement à cette dernière problématique.

Ces dernières années, il apparaît de plus en plus judicieux d'utiliser des modèles analytiques des solutions envisagées, combinés à de l'optimisation, ce qui permet d'agir sur de nombreux paramètres tout en conservant des temps de réponse rapides. Les algorithmes d'optimisation sont éminemment variés, et certains peuvent requérir l'évaluation des sensibilités des sorties du modèle de dimensionnement en fonction de ses entrées.

Dans nos travaux, nous nous sommes particulièrement attachés à traiter la résolution des modèles de dimensionnement, ceux-ci pouvant inclure des systèmes d'équations différentielles. Dans cette optique, nous nous sommes focalisés sur la résolution des systèmes d'état linéaires. Nous avons aussi proposé un standard de composants pour le dimensionnement, ainsi qu'un environnement de gestion de ces composants.

Dans le premier chapitre, après une présentation de la problématique du dimensionnement, nous rappelons les éléments fondamentaux des systèmes d'état, leur formulation, ainsi que leur résolution. Nous développons notamment la résolution symbolique des systèmes d'état linéaires. Les modèles devant être utilisés dans des environnements souples et ouverts, nous nous appuyons sur le concept de composant logiciel que nous présentons, ainsi que la manière de les utiliser : création, composition et projection.

Dans le second chapitre, nous développons différentes méthodes de résolution des modèles de dimensionnement. La contrainte essentielle, due à l'utilisation de certains algorithmes d'optimisation, est l'évaluation des gradients des sorties du modèle en fonction de ses entrées.
Nous commençons par expliciter des méthodes symboliques pour la résolution des systèmes d'état linéaires s'appuyant sur l'exponentielle de matrice. Afin d'évaluer cette dernière, deux méthodes numériques sont utilisées et comparées. Nous introduisons ensuite des formulations supplémentaires basées sur les systèmes d'état afin que leur résolution fournisse aussi les gradients de leurs solutions en fonction des paramètres d'entrée du modèle.
Afin de traiter les modèles reposant sur des algorithmes numériques, nous avons utilisé la dérivation de code. Dans ce cadre, nous nous sommes appuyés sur une librairie existante, Adol-C, permettant la dérivation de code en C / C++, et nous avons développé JavaDiff qui est un premier pas vers la dérivation de code en Java.

Dans le troisième chapitre, nous proposons un standard de composants pour le dimensionnement, ICAr, dans l'optique de faciliter l'utilisation complémentaire de services hétérogènes, développés indépendamment jusqu'à présent. Afin d'aider le concepteur à gérer ses composants, nous avons développé un environnement : CoreLab. Cet environnement a été conçu afin que des développeurs puissent enrichir ses fonctionnalités pour la gestion des composants (génération, composition, projection).

Dans le quatrième chapitre, nous illustrons les différentes possibilités de notre environnement. Nous commençons par la création d'un composant de résolution de systèmes d'état linéaires. Ensuite, nous explicitons la démarche permettant au concepteur de passer d'un modèle de dimensionnement à un composant de calcul utilisable dans un environnement d'optimisation. Nous faisons aussi un bilan des différentes méthodes de dérivation des modèles permettant d'obtenir leurs gradients. Nous terminons ce chapitre par les avantages procurés par l'utilisation des concepts, des méthodes et des outils proposés, avant de conclure.

CHAPITRE I

Cadre de travail

I.1. Contexte de travail : la conception en Génie Electrique **15**
I.1.1. Le processus de conception 15
I.1.2. Le dimensionnement du dispositif 16
I.1.3. La conception assistée par ordinateur 17
I.2. L'optimisation sous contraintes **19**
I.3. La résolution des modèles de dimensionnement **20**
I.3.1. La modélisation des systèmes physiques 21
I.3.2. Le système d'état 22
I.3.2.1. Etat d'un système 22
I.3.2.2. Vecteur d'état 22
I.3.2.3. Espace d'état 22
I.3.2.4. Trajectoire d'état 22
I.3.2.5. Equation d'état 22
I.3.2.6. Exemple de modélisation d'un système par une équation d'état 23
I.3.3. Résolution du système d'état dans le cas des systèmes linéaires 24
I.3.3.1. Résolution de l'équation homogène 24
I.3.3.2. Solution complète de l'équation d'état 25
I.3.4. Résolution du système d'état dans le cas non linéaire 26
I.4. Le besoin d'un environnement pour le dimensionnement **26**
I.5. Le composant logiciel **28**
I.5.1. Le composant : abstraction logicielle 28
I.5.2. Code et composant 29
I.5.2.1. Réutilisation ouverte ou fermée 29
I.5.2.2. Objet et composant 30
I.5.3. Les différentes interactions avec les composants 31
I.5.3.1. La création des composants 31
I.5.3.2. La projection 32
I.5.3.3. La composition des composants 33
I.5.4. Les différents types de composants existant 34
I.6. Solution proposée **35**
I.6.1. La résolution des modèles de dimensionnement 35
I.6.2. L'approche composant pour le dimensionnement 35

CHAPITRE I

CADRE DE TRAVAIL

I.1. Contexte de travail : la conception en Génie Electrique

Notre contexte de travail se situe dans le cadre de la conception en Génie Electrique. La conception d'un dispositif électrique se déroule en plusieurs phases. Un processus de conception permet de passer de l'expression du besoin, et donc de la nécessité d'un dispositif afin de répondre à ce besoin, à la réalisation des plans du dispositif.

I.1.1. Le processus de conception

On définit généralement le processus de conception comme étant l'exploration d'espaces de solutions, la simulation et la vérification des solutions trouvées [DEL]. Mais la possibilité de revenir en arrière lors d'une étape de la conception est toujours envisageable. En effet, chaque étape du processus peut remettre en cause une ou plusieurs étapes précédentes, obligeant le concepteur à revenir sur ses pas. Le processus de conception évolue donc de manière imprévisible.

La première étape du processus de conception consiste généralement en une analyse fonctionnelle, permettant de définir les différentes fonctionnalités que doit offrir le dispositif afin de répondre au besoin exprimé. Un cahier des charges commence alors à être défini puis complété au fur et à mesure du processus de conception. A partir de ce cahier des charges, le concepteur définit et choisit une structure, puis évalue les différents paramètres du dispositif, suivant les différentes contraintes et les objectifs spécifiés pour le dimensionnement.

Mais le concepteur peut aussi réutiliser une structure existante et ne faire que la redimensionner en spécifiant de nouvelles contraintes et de nouveaux objectifs au dimensionnement. C'est notamment le cas pour des produits déclinés dans plusieurs gammes (comme une gamme de moteurs de différentes puissances par exemple).

Dans les deux cas, une fois le dimensionnement achevé, il reste à vérifier et valider le dispositif par simulation, en établir les plans, avant de passer à la construction des prototypes, faire de l'expérimentation, et enfin attaquer la production en série. Dans ce projet, nous nous intéressons particulièrement au dimensionnement.

I.1.2. Le dimensionnement du dispositif

Le dimensionnement est une étape complexe du processus de conception. Les différentes phases du processus de dimensionnement tel que nous l'appréhendons peuvent être définies de la manière suivante :

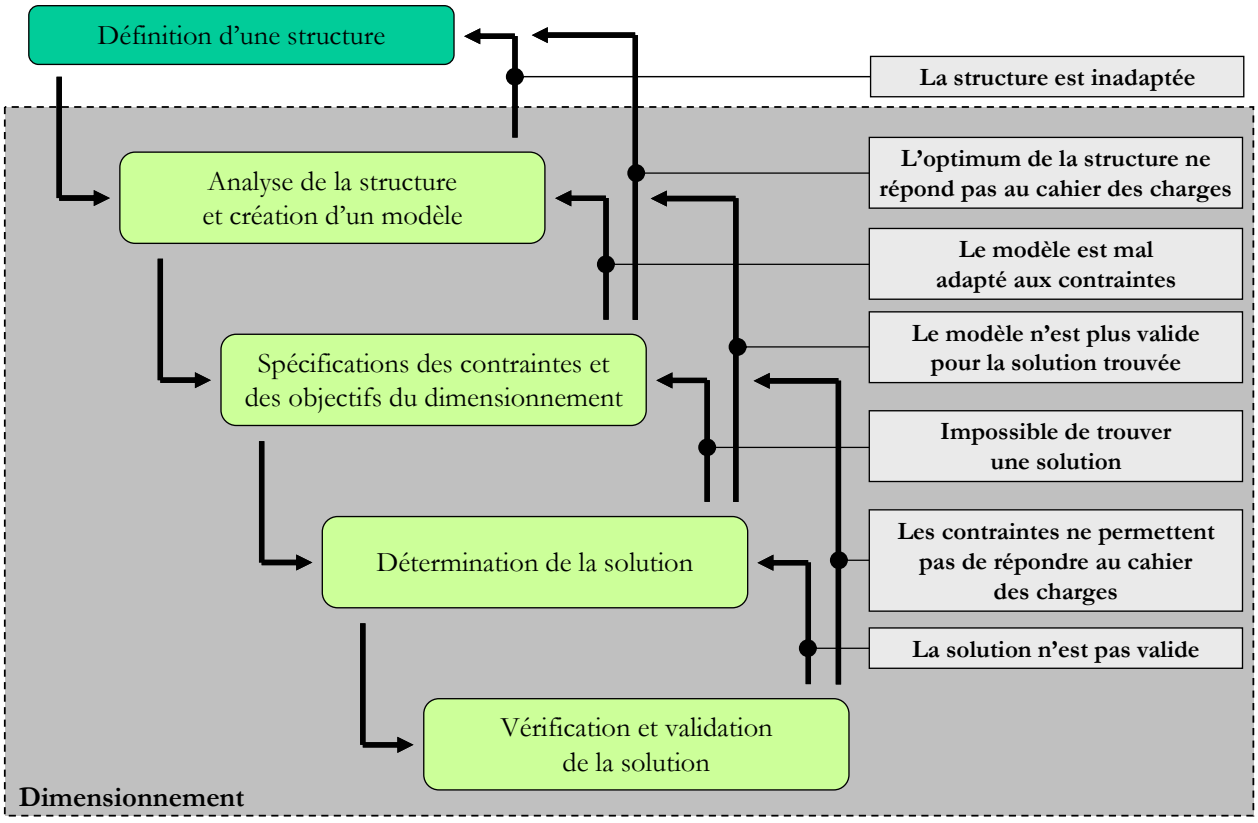


Figure I.1 : Principe du dimensionnement

Tout comme pour le processus global de conception, chaque étape du processus de dimensionnement peut remettre en cause une ou plusieurs étapes précédemment effectuées. L'étape du dimensionnement peut aussi remettre en cause le choix de structure effectué par le concepteur.

Le dimensionnement doit respecter les contraintes exprimées par le cahier des charges, qui peuvent être de nature variée, par exemple :

- Contraintes d'encombrement : les dimensions géométriques du dispositif doivent respecter certains gabarits, son poids doit être inférieur à une certaine limite, …
- Contraintes de compatibilité : le dispositif doit être compatible avec son environnement (respect des normes CEM, diffusion de chaleur, …)
- Contraintes économiques : la production et / ou l'utilisation du dispositif doivent être inférieures à un coût critique.
- Contraintes de qualité du résultat : le résultat fournit par le dispositif doit être suffisamment précis, rapide, stable, …

La problématique du dimensionnement est d'atteindre un objectif tout en respectant toutes les contraintes exprimées par le cahier des charges. Par exemple, dans le cas d'un convertisseur statique, cela consiste à obtenir le meilleur rendement possible, et / ou le volume minimal.
Le dimensionnement utilise un modèle du dispositif à concevoir. Ce modèle de dimensionnement peut être considéré comme une boite noire[1] (voir la figure I.2), et possède comme entrées les différents paramètres du dispositif (paramètres géométriques, paramètres physiques, paramètres économiques, …). Les sorties de ce modèle sont les différents critères de dimensionnement. Ces critères peuvent concerner des grandeurs physiques (courants ou tensions à une certaine date, valeurs moyennes ou maximales, temps de réponse, norme CEM, rendement, pertes, …), des grandeurs économiques (coût de production, coût d'utilisation, durée de vie, amortissement, …), ou autres.

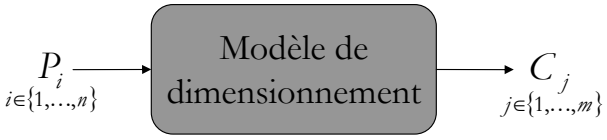


Figure I.2 : Le modèle de dimensionnement, avec ses entrées P_i et ses sorties C_j

Dans ce cadre, deux types de problèmes peuvent être envisagés : le problème direct et le problème inverse. Le problème direct est le fait de calculer, à partir de la connaissance des entrées P_i, la valeur des sorties C_j. Le problème posé par le dimensionnement est différent. A partir des valeurs désirées pour les différents critères de dimensionnement (valeurs imposées par le cahier des charges), il faut calculer la valeur de chaque paramètre d'entrée, tout en s'assurant qu'il appartienne à son domaine de validité (contraintes imposées par le cahier des charges). L'obtention des P_i est généralement impossible directement à partir des C_j sans le recours à des méthodes numériques. Il va donc falloir utiliser un processus itératif afin de déterminer quelles sont les valeurs des entrées correspondant aux valeurs désirées des sorties. La solution que nous proposons ici est l'utilisation d'algorithmes d'optimisation afin de déterminer le jeu de valeurs optimal des paramètres d'entrée.

I.1.3. La conception assistée par ordinateur

La conception est une activité complexe qui nécessite un support informatique. En effet, le but de la CAO[2] est d'aider au maximum le concepteur. Il existe plusieurs outils dont la fonction est d'apporter un support au processus de conception de manière générale, ou à telle ou telle étape du processus de conception. L'utilisation de ces outils impose aussi des contraintes, comme la

[1] Boite noire : entité dont on connaît les fonctionnalités, mais pas le fonctionnement. Voir glossaire : encapsulation.
[2] CAO : Conception Assistée par Ordinateur. Voir glossaire.

gestion des informations techniques (modèle produit [HAR] par exemple), ou la standardisation des méthodes de conception [LAV].

De plus, la conception est une activité qui nécessite des connaissances dans différents domaines de la physique (électronique, électromagnétisme, thermique, mécanique) ou d'autres domaines (comme par exemple l'économie).

Il faut donc des outils capables d'assister le concepteur dans ces domaines variés, qui épargnent au maximum au concepteur les tâches fastidieuses ou répétitives. De plus, un des enjeux émergents de ces dernières années est la gestion de la complexité et du compliqué. En effet, l'accélération du rythme de vie du monde scientifique et industriel font ressentir le besoin de gestion du compliqué. Le complexe se gère pas une meilleure prise en compte des besoins du concepteur et du savoir des concepteurs dans les méthodes et les outils de conception. Le compliqué se caractérise par la prise en charge des outils de CAO[1] de tout ce qui est répétitif et laborieux pour le concepteur, sans être complexe, comme par exemple l'expertise.

Un moyen de gérer certains aspects du compliqué, d'assurer une réutilisabilité des connaissances capitalisées et ainsi d'augmenter la réactivité du concepteur est d'utiliser une description du dispositif, à partir de laquelle on va générer les codes de calcul correspondant au dispositif. Le principe de cette approche est illustré sur la figure I.3 :

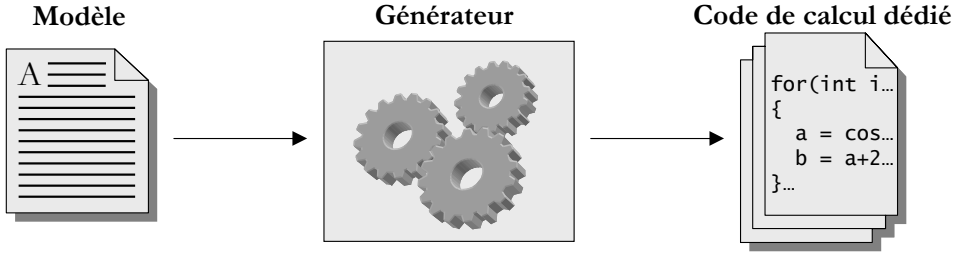


Figure I.3 : Approche par génération de code

On épargne ainsi au concepteur toute programmation, lui permettant donc de se consacrer pleinement à la conception même du dispositif. Le concepteur décrit son dispositif dans un langage adapté (par exemple, des équations analytiques pour décrire un modèle analytique), puis cette description est analysée, et à partir de cette analyse, le code de calcul correspondant est généré automatiquement, spécifiquement dédié à l'environnement de conception ou de simulation choisi. Cette approche a déjà été utilisée dans de nombreux logiciels de CAO, comme Gentiane [GE2], Pascosma [WUR] ou MAEL [ALL]. C'est sur cette approche que nous baserons nos travaux

[1] CAO : Conception Assistée par ordinateur. Voir glossaire.

I.2. L'optimisation sous contraintes

L'utilisation d'un algorithme d'optimisation afin de résoudre le problème inverse donne généralement de bons résultats mais impose un certain nombre de contraintes sur l'écriture et le calcul du modèle de dimensionnement. Nous allons ici expliquer les principes généraux de fonctionnement d'un algorithme d'optimisation afin d'en déduire les contraintes imposées au modèle de dimensionnement.

L'optimisation est en fait la minimisation (ou la maximisation) d'une fonction dépendante d'une ou plusieurs variables [POL]. Cette fonction est appelée fonction objectif. Dans le cas du dimensionnement, cette fonction objectif est une fonction des paramètres d'entrée du modèle et des critères de dimensionnement en sortie du modèle. L'algorithme d'optimisation doit donc trouver un jeu de paramètres d'entrée satisfaisant aux contraintes données par les cahiers des charges (contraintes sur les entrées P_i et sur les sorties C_j) et minimisant la fonction objectif [PRE]. Ce fonctionnement est illustré sur la figure I.4 :

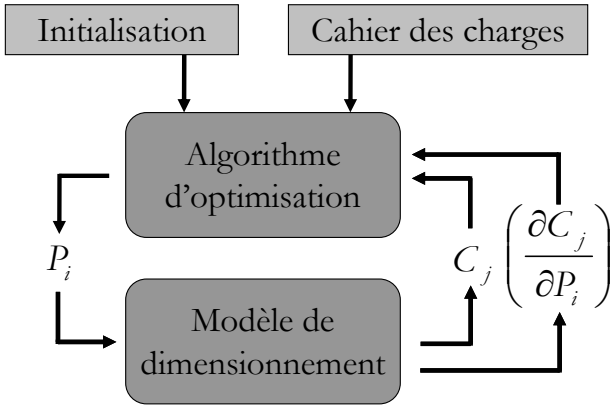


Figure I.4 : Le fonctionnement général de l'optimisation

Le fonctionnement d'un algorithme d'optimisation est itératif : pour un jeu donné de valeurs des paramètres d'entrée du modèle, le modèle est calculé, les critères de dimensionnement sont analysées, et, éventuellement, leurs dérivées par rapport aux paramètres d'entrée sont évaluées, puis l'algorithme d'optimisation propose un nouveau jeu de paramètres. Ce nouveau jeu est utilisé pour le calcul du modèle, et le cycle recommence. Par exemple, un algorithme d'optimisation utilisant les gradients explore ainsi l'espace des solutions de proche en proche et détermine le minimum de la fonction objectif.

On distingue deux familles d'algorithmes d'optimisation. Les premiers utilisent uniquement l'évaluation de la fonction objectif afin de trouver le minimum. On trouve par exemple dans cette famille les algorithmes stochastiques (les algorithmes génétiques en font partie). La deuxième famille, afin de se diriger à travers l'espace des solutions vers le minimum de la fonction le plus efficacement possible, utilise quant à elle les gradients de la fonction objectif. Cela minimise le

nombre d'itérations nécessaires pour trouver la solution, et donc le nombre de résolutions du modèle de dimensionnement. Pour des applications avec beaucoup de paramètres, ces algorithmes sont généralement beaucoup plus efficaces que ceux qui utilisent uniquement les évaluations de la fonction objectif, et c'est pourquoi nous nous appuierons par la suite sur ces algorithmes. Cependant, ils peuvent se faire piéger par des optimums locaux.

Une contrainte forte liée à l'utilisation d'algorithmes d'optimisation par gradients est donc qu'il faut leur fournir un modèle du dispositif calculant non seulement la valeur de la fonction objectif, mais aussi les gradients de cette fonction par rapport aux différents paramètres d'entrée du modèle.

L'autre contrainte forte liée à l'utilisation d'algorithmes d'optimisation est qu'il faut connecter le modèle de dimensionnement à l'algorithme d'optimisation. Or on ne peut pas demander au concepteur d'écrire le code informatique qui fera le lien entre l'algorithme et le modèle. Nous nous appuierons sur une structure informatique qui permet de connecter facilement un algorithme d'optimisation et un modèle de dimensionnement, de spécifier les différentes contraintes qui vont s'appliquer aux paramètres du modèle ainsi qu'aux critères de dimensionnement, et de lancer l'optimisation.

I.3. La résolution des modèles de dimensionnement

Comme on l'a vu, le pré-requis essentiel des modèles de dimensionnement que nous allons utiliser est de fournir la valeur des critères de dimensionnement C_j à partir des valeurs des paramètres d'entrée P_i, ainsi que les dérivées partielles de ces critères par rapport aux entrées $\frac{\partial C_j}{\partial P_i}$.

Comme le modèle de dimensionnement est destiné à être utilisé en interaction avec un algorithme d'optimisation, sa résolution doit répondre à plusieurs contraintes du point de vue informatique. Tout d'abord, la résolution doit être rapide et occuper aussi peu de mémoire que possible. En effet, comme l'utilisation de l'algorithme d'optimisation est un processus itératif, le nombre de résolutions du modèle peut devenir très important, et une résolution du modèle lente ou demandant beaucoup de mémoire devient donc rapidement un point bloquant majeur à l'optimisation. Ensuite, la résolution du modèle doit être robuste[1].

[1] Robuste : qui ne diverge pas numériquement et qui ne provoque pas d'erreurs lors du calcul.

Un modèle de dimensionnement peut contenir plusieurs types de calculs, par exemple :

- Des calculs analytiques, basés sur des formulations analytiques de type $a = b + c * d$
- Des résolutions de systèmes différentiels de type $f(x, \dot{x}, u, t) = 0$
- Des résolutions de systèmes implicites de type $f(a, b, c) = 0$
- Des simulations numériques de type intégration temporelle (type Matlab / Simulink)
- Des simulations par éléments finis (type Flux2D)
- …

Nous allons ici nous intéresser plus particulièrement aux deux premiers types de calculs, à savoir les calculs basés sur des modèles analytiques, et les résolutions d'équations différentielles appelées à partir de modèles analytiques. En effet, ces types de calculs sont souvent rencontrés lors de la modélisation de dispositifs, et le second est pourtant peu abordé actuellement

I.3.1. La modélisation des systèmes physiques

Afin de pouvoir optimiser un dispositif, un modèle de celui-ci est nécessaire. Un dispositif est un ensemble d'éléments interagissant entre eux. Cette définition reste forcément très générale afin de pouvoir englober les divers systèmes que l'on peut rencontrer. Le point important de cette définition est l'interaction entre les différents éléments, qui devra être prise en compte pour la modélisation, plutôt que de traiter chaque élément séparément.

Il y a plusieurs façons de représenter mathématiquement le comportement dynamique d'un système physique. Le choix d'une représentation plutôt qu'une autre dépend à la fois des objectifs de la modélisation, et des outils à disposition pour effectuer la modélisation. En particulier, certaines formes sont plus adaptées à la représentation de certains systèmes plutôt que d'autres. Les différentes formes communément recensées sont [MUT] :

- La représentation sous forme de système d'état
- La représentation sous forme de système différentiel (entre les entrées et les sorties du système)
- La représentation sous forme d'un système de fonctions de transfert
- La représentation sous forme d'un schéma-bloc (très utilisée en automatique)
- La représentation sous forme de Bond Graphs.

Nous allons ici nous intéresser plus particulièrement à la représentation sous forme de système d'état, particulièrement utile dans le cadre des systèmes linéaires, qui sont très utilisés dans le domaine du Génie Electrique.

I.3.2. Le système d'état

I.3.2.1. Etat d'un système

L'état d'un système est défini par Kalman [KAL] comme une structure mathématique contenant un jeu de *n* variables $x_1(t), x_2(t), \ldots, x_n(t)$. Ces variables sont appelées variables d'état, et sont telles que leurs valeurs initiales $x_i(t_0)$ et les entrées du système $u_j(t)$ sont suffisantes pour décrire de manière unique la réponse future du système pour $t \geq t_0$. Un jeu minimum de variables d'état est requis pour représenter le système correctement.

Les variables d'état ne sont pas nécessairement des valeurs physiques observables ou des quantités mesurables. Elles peuvent avoir une existence uniquement mathématique.

I.3.2.2. Vecteur d'état

Le jeu de variables d'état représente les éléments d'un vecteur de dimension *n*, appelé vecteur d'état et noté $X(t)$:

$$X(t) = \begin{bmatrix} x_1(t) \\ \vdots \\ x_2(t) \end{bmatrix} \tag{I.1}$$

L'ordre de l'équation caractéristique du système est *n*, et l'équation d'état représentant le système est constituée de *n* équations différentielles du premier ordre.

I.3.2.3. Espace d'état

L'espace d'état est défini comme étant l'espace de dimension *n* dans lequel les composantes du vecteur d'état représentent les coordonnées de ce vecteur.

I.3.2.4. Trajectoire d'état

La trajectoire d'état est définie comme le chemin emprunté par le vecteur d'état dans l'espace d'état au cours du temps.

I.3.2.5. Equation d'état

L'équation d'état d'un système est en fait un jeu de *n* équations différentielles du premier ordre. L'équation d'état générale se présente sous la forme suivante :

$$f\left(X(t), \dot{X}(t), U(t), t\right) = 0 \tag{I.2}$$

où $U(t)$ représente les entrées du système.

Dans de nombreuses applications, on peut se ramener à une équation d'état (aussi appelée équation de transition), qui peut se présenter sous la forme suivante :

$$\dot{X}(t) = A \cdot X(t) + B \cdot U(t) \tag{I.3}$$

où A et B sont des matrices caractéristiques du système et dépendent des paramètres du système. A est appelée matrice d'état et B matrice d'entrée (appelée aussi matrice de contrôle). Comme les états du système ne sont pas forcément les sorties du système, une deuxième équation, appelée équation de mesure, vient compléter l'équation d'état afin de former le système d'état :

$$Y(t) = C \cdot X(t) + D \cdot U(t) \tag{I.4}$$

où C est appelée matrice de sortie et D matrice d'entrée-sortie.

I.3.2.6. Exemple de modélisation d'un système par une équation d'état

Afin d'illustrer le concept de modélisation par équation d'état, nous allons modéliser le circuit suivant :

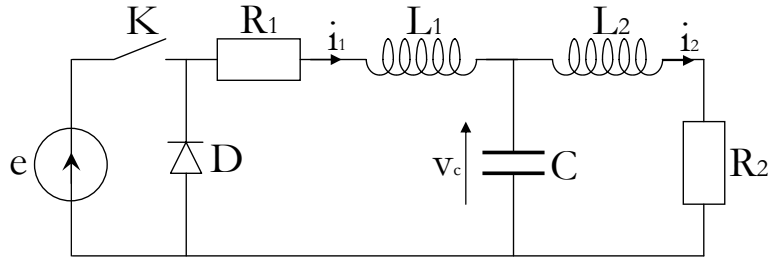


Figure I.5 : Le circuit électrique à modéliser

Les équations régissant ce circuit sont les suivantes :

$$\begin{cases} R_1 x_1 + L_1 \dot{x}_1 + x_3 = u \\ L_2 \dot{x}_2 + R_2 x_2 - x_3 = 0 \\ -x_1 + x_2 + C\dot{x}_3 = 0 \end{cases} \tag{I.5}$$

avec $i_1 = x_1$, $i_2 = x_2$, $v_C = x_3$, $u = \begin{cases} e & \text{si K fermé et D ouvert} \\ 0 & \text{si K ouvert et D fermé} \end{cases}$

Nous pouvons donc déduire à partir de ces trois équations différentielles l'équation d'état correspondante :

$$\begin{cases} \dot{X} = \begin{bmatrix} -\frac{R_1}{L_1} & 0 & -\frac{1}{L_1} \\ 0 & -\frac{R_2}{L_2} & \frac{1}{L_2} \\ \frac{1}{C} & -\frac{1}{C} & 0 \end{bmatrix} \cdot X + \begin{bmatrix} \frac{1}{L_1} \\ 0 \\ 0 \end{bmatrix} \cdot u \\ Y = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \cdot X \end{cases} \tag{I.6}$$

Cette modélisation peut donc s'écrire sous la forme suivante :

$$\begin{cases} \dot{X} = A \cdot X + B \cdot u \\ Y = C \cdot X \end{cases} \tag{I.7}$$

I.3.3. Résolution du système d'état dans le cas des systèmes linéaires

La modélisation d'un système par son équation d'état présente plusieurs avantages. Dans des systèmes de grande taille (c'est-à-dire comportant beaucoup d'entrées, de sorties, ou d'états), cette approche est beaucoup plus adaptée que les autres à une résolution numérique du fait de la formulation du problème sous forme matricielle. Dans le cas où le système est linéaire, cette approche permet d'envisager la résolution du système d'équations modélisant le dispositif de manière symbolique. Nous allons ici nous intéresser à la résolution d'un système linéaire modélisé par une équation d'état, comportant *m* entrées, *l* sorties et *n* états, comme le montre la figure I.6 :

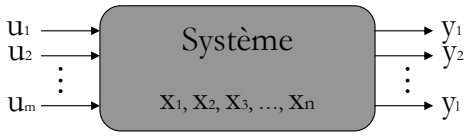


Figure I.6 : Représentation générale du système

L'équation d'état modélisant ce système est donc :

$$\begin{cases} \dot{X}(t) = A \cdot X(t) + B \cdot u(t) \\ Y(t) = C \cdot X(t) + D \cdot u(t) \end{cases} \tag{I.8}$$

Les variables $x(t)$, $y(t)$ et $u(t)$ sont des vecteurs colonne et A, B, C et D sont des matrices à coefficients constants.

I.3.3.1. Résolution de l'équation homogène

L'équation homogène est tirée de l'équation d'état générale (I.8) où les entrées ont été annulées. L'équation d'état devient donc :

$$\dot{X}(t) = A \cdot X(t) \tag{I.9}$$

Ici, A est une matrice $n \times n$ et $x(t)$ est un vecteur colonne de taille *n*.

Considérons le cas de l'équation différentielle scalaire du premier ordre suivante :

$$\dot{x}(t) = ax(t) \tag{I.10}$$

La solution de cette équation est donnée par :

$$x(t) = x(t_0)e^{a(t-t_0)} \tag{I.11}$$

En comparant l'équation différentielle scalaire (I.10) et l'équation d'état (I.9), on déduit que la solution de l'équation d'état doit être analogue à la solution exprimée en (I.11). On est donc

amené à introduire l'opérateur exponentiel dans le cadre matriciel, ce qui permet d'écrire la solution de l'équation d'état :

$$X(t) = e^{A(t-t_0)} \cdot X(t_0) \tag{I.12}$$

L'opérateur exponentiel de matrice est défini par analogie avec le développement en série entière de l'exponentielle scalaire :

$$e^{at} = 1 + \frac{at}{1!} + \frac{(at)^2}{2!} + \cdots + \frac{(at)^k}{k!} + \cdots = \sum_{k=0}^{+\infty} \frac{(at)^k}{k!} \tag{I.13}$$

Soit :

$$e^{At} = I + \frac{t}{1!} A + \frac{t^2}{2!} A^2 + \cdots + \frac{t^k}{k!} A^k + \cdots = \sum_{k=0}^{+\infty} \frac{t^k}{k!} A^k \tag{I.14}$$

L'exponentielle d'une matrice carrée est donc une matrice carrée de même taille. On peut déduire quelques propriétés de cette matrice :

$$e^{At_1} e^{At_2} = e^{A(t_1+t_2)} \tag{I.15}$$

$$e^{At} \cdot e^{At} \cdot \cdots \cdot e^{At} = \left(e^{At}\right)^q = e^{Aqt} \tag{I.16}$$

$$e^{A*0} = I \tag{I.17}$$

De plus, cette matrice est non singulière quelque soit *t*.

Enfin, on remarque que les valeurs propres λ_i associées à A sont des solutions de l'équation :

$$\det(A - \lambda_i I) = 0 \tag{I.18}$$

On peut donc relier les valeurs propres de A avec les fréquences caractéristiques du système modélisé par cette équation d'état.

I.3.3.2. Solution complète de l'équation d'état

Quand une entrée est présente, la solution de l'équation d'état est obtenue à partir de l'équation (I.8). Le point de départ afin d'obtenir cette solution est l'égalité suivante, obtenue en appliquant la règle de la dérivation du produit de deux matrices :

$$\frac{d}{dt}\left[e^{-At} \cdot X(t)\right] = e^{-At}\left[\dot{X}(t) - A \cdot X(t)\right] \tag{I.19}$$

En injectant l'équation (I.8) dans cette égalité, on obtient :

$$\frac{d}{dt}\left[e^{-At} \cdot X(t)\right] = e^{-At} \cdot Bu \cdot (t) \tag{I.20}$$

En intégrant cette expression entre *0* et *t*, il vient :

$$\left[e^{-At} \cdot X(t)\right] - \left[e^{-A\times 0} \cdot X(0)\right] = \int_0^t e^{-A\tau} \cdot B \cdot u(\tau) d\tau \tag{I.21}$$

En multipliant par e^{At} à gauche et en réarrangeant les termes, l'expression devient :

$$X(t) = e^{At} \cdot X(0) + \int_0^t e^{A(t-\tau)} \cdot B \cdot u(\tau) d\tau \quad \text{(I.22)}$$

Cette forme est la solution générale de l'équation d'état. C'est sur cette forme que nous baserons nos algorithmes de résolution de systèmes différentiels.

I.3.4. Résolution du système d'état dans le cas non linéaire

Pour les systèmes différentiels non linéaires, la résolution symbolique du système d'état n'est pas possible. Il nous faudra donc nous appuyer sur d'autres méthodes afin de résoudre ces systèmes. Ces méthodes seront des méthodes numériques, comme par exemple :

- Les méthodes d'intégration numériques pour la simulation (trapèzes, Runge-Kutta, …).
- Les méthodes de simulation par élément finis.

Nous verrons au Chapitre III comment intégrer ces différentes méthodes lors de la résolution des modèles de dimensionnement.

I.4. Le besoin d'un environnement pour le dimensionnement

Une fois la modélisation du système accomplie, il reste encore à traduire cette modélisation mathématique en un code informatique calculant les sorties du modèle à partir de la connaissance de la valeur des entrées. Afin d'être utilisé par des algorithmes d'optimisation par gradients, le code informatique calculant le modèle doit aussi fournir la valeur des dérivées des sorties par rapport aux entrées. Il faudra donc que, pour tous les calculs intervenant lors de la résolution du modèle (calculs basés sur des modèles analytiques, résolution de systèmes différentiels, calculs et simulations numériques), le calcul des gradients (ou du jacobien) soit mené en parallèle.

De plus, l'implantation informatique de ces calculs doit être facilement utilisable par le concepteur, qui ne doit pas perdre son temps en manipulations informatiques inutiles ou automatisables, comme nous le verrons au Chapitre III. En effet, tout le temps consacré par le concepteur aux manipulations informatiques nécessaires au déroulement du processus de conception est du temps perdu. Il faut donc essayer de réduire ce temps au maximum. De plus, chacune de ces manipulations est potentiellement source d'erreurs. Il faut donc essayer d'automatiser un maximum de tâches du processus de conception, afin que le concepteur puisse se consacrer uniquement aux tâches où son expertise et son intelligence sont requises.

On voit donc apparaître ici le besoin d'un environnement informatique pour supporter cette phase de dimensionnement. Certains environnements existent et sont commercialement disponibles. La plupart de ces environnements sont dédiés à l'optimisation d'un certain type de dispositif (comme par exemple ARNO, un logiciel d'optimisation des réseaux de radio-téléphonie

mobile [ARN]). Quelques logiciels de dimensionnement à visée généraliste existent (comme par exemple Pascosma [WUR], ou Pro@Design [ATI]), mais ils ne proposent pas tous les services permettant d'assister le concepteur dans la phase de dimensionnement, comme par exemple des services de gestion des algorithmes entrant en jeu lors de la résolution des modèles.

De manière générale, l'environnement informatique, pour supporter la phase de dimensionnement, doit avoir plusieurs caractéristiques :

- Il doit supporter les différentes phases du dimensionnement et minimiser autant que possible les manipulations nécessaires pour passer d'une phase à l'autre.
- Il doit épargner au maximum les manipulations informatiques que le concepteur doit accomplir afin de créer ses modèles et de les coupler avec un algorithme d'optimisation.
- Il doit proposer une écriture des modèles la plus naturelle et intuitive possible pour le concepteur.
- Il doit être ouvert car le processus de conception étant par nature imprévisible, il faut laisser la possibilité au concepteur, moyennant une manipulation informatique aussi minimale que possible, de traiter des problèmes qui n'avaient pas été prévus à la base.

Dans ce cadre général, nous nous appuierons sur une architecture logicielle modulaire permettant au concepteur d'étendre les possibilités existantes de l'environnement et d'en ajouter de nouvelles, afin d'offrir une ouverture maximale, tant au niveau de l'écriture des modèles que des problèmes traités. L'environnement proposé sera basé sur un concept informatique permettant au concepteur de limiter au maximum les manipulations informatiques : les composants informatiques. Cette approche nous permettra de satisfaire aux différentes contraintes évoquées pour l'environnement de dimensionnement.

L'approche que nous proposons devra donc se baser sur le processus suivant :

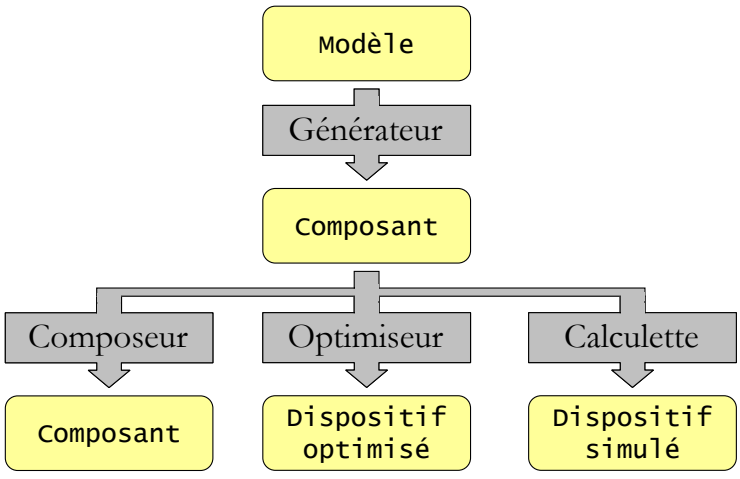


Figure I.7 : Processus lié à l'approche composant

A partir d'un modèle de dimensionnement du dispositif, un composant résolvant le modèle de dimensionnement de ce dispositif est créé. Ce composant peut ensuite être utilisé de différentes manières :

- Tout d'abord, il peut servir à une procédure d'optimisation.
- Il peut aussi servir dans une calculette à simuler le dispositif et à obtenir la valeur des critères de dimensionnement pour un jeu de paramètres d'entrée donné.
- Enfin, au sein d'applications spécifiques pour les composants (composeurs, projeteurs, …), il peut servir afin de créer de nouveaux composants.

Cette approche est donc basée sur les composants informatiques. Nous allons maintenant voir plus en détail la nature des composants informatiques, ainsi que leur utilisation.

I.5. Le composant logiciel

I.5.1. Le composant: abstraction logicielle

La notion de composant logiciel est apparue dans les années 90 par analogie avec les composants électroniques. Le composant logiciel a été défini [MEI] comme une abstraction d'une structure logicielle donnée, utilisée pour construire une structure plus grande, tout en cachant les détails de l'implémentation de la structure qu'il encapsule. On peut aussi le définir [SZY] comme étant une unité autonome de déploiement d'un code informatique qu'il encapsule, et décrivant par des interfaces les différentes interactions possibles avec les autres composants ou des outils logiciels. L'idée de base est que l'on peut construire un logiciel comme on construit un circuit électronique : on utilise des composants que l'on relie entre eux afin d'obtenir le comportement désiré.

L'intérêt de cette approche est double. D'abord, le composant permet de réutiliser du code informatique déjà écrit pour une autre application, et d'obtenir un comportement différent en le connectant différemment. De plus, il permet d'utiliser de manière sécurisée un code informatique déjà écrit, tout en cachant les détails de ce code. Le composant fournit une interface d'utilisation, qui définit précisément tout ce qu'il peut faire, et tout ce dont il a besoin pour fonctionner. On peut donc représenter un composant de la manière suivante :

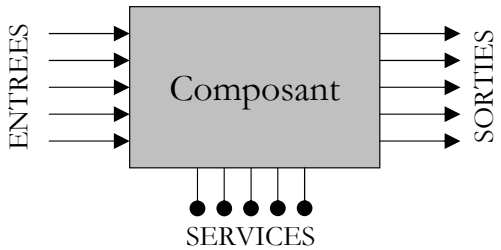


Figure I.8 : Représentation du composant logiciel

De même que la classe d'un objet définit sa structure et son comportement, l'interface d'un composant définit ses entrées et ses sorties, ainsi que les services qu'il fournit (par exemple, un service peut être la résolution d'un modèle de dimensionnement). Par contre, à la différence de la classe d'un objet, l'interface d'un composant n'impose en rien sa structure interne. L'implémentation interne du composant reste donc totalement libre de toute contrainte.

I.5.2. Code et composant

I.5.2.1. Réutilisation ouverte ou fermée

Afin de cerner les différences entre un composant logiciel contenant un code informatique, et du code informatique sans encapsulation, il faut préciser les notions de code et de composant.

On peut voir un programme informatique comme une structure composée d'instructions, de procédures, de méthodes, de classes, … Dans la forme la plus basique du développement logiciel, le programmeur doit créer cette structure à partir de zéro.

Un programmeur à qui l'on fournit certaines pièces logicielles qui peuvent être adaptées et combinées (comme par exemple une séquence d'instructions, ou un groupe de classes coopérantes) peut déjà atteindre un certain degré de réutilisation. Nous appellerons cela une réutilisation "ouverte", ou de type "boîte blanche", puisque les structures réutilisées ne sont pas encapsulées, et peuvent être adaptées suivant les besoins du programmeur.

Adapter des structures de type boîte blanche peut toutefois être extrêmement difficile, car le programmeur doit comprendre ce que chaque élément de la structure fait, et comment il interagit avec les autres éléments de la structure. La complexité d'adaptation dépend bien évidemment de la complexité de la structure qui doit être adaptée. De plus, mettre ensemble plusieurs structures complexes afin de former un système encore plus gros (comme par exemple fusionner des groupes d'instructions différents) est aussi délicat. C'est ici que le composant logiciel peut apporter une aide considérable.

Comme on l'a vu précédemment, le composant logiciel est une unité autonome qui cache les détails de son implémentation. Connecter les composants est simple, puisque chaque composant possède un certain nombre de connecteurs, avec des règles fixées qui spécifient comment ce composant peut être relié à d'autres composants. A la place d'adapter une pièce logicielle afin de modifier sa fonctionnalité, l'utilisateur a juste à brancher les connecteurs et donner les paramètres du composant pour obtenir le comportement désiré. Nous appellerons cette approche la réutilisation "fermée", ou de type "boîte noire".

La différence entre les deux approches est illustré sur la figure suivante :

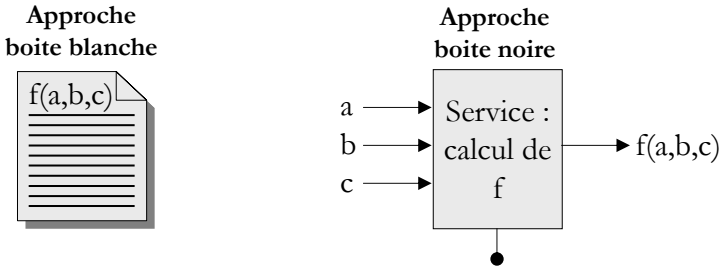


Figure I.9 : Différence entre boite blanche et boite noire

I.5.2.2. Objets et composants

Comme on vient de le voir, le composant logiciel est donc une structure informatique permettant de connecter et d'utiliser facilement des structures logicielles afin de créer un programme complet, chaque composant proposant une ou plusieurs fonctionnalités (appelées services), et des paramètres d'entrée et de sortie afin de fournir les données au composant et de récupérer les résultats. Mais les possibilités offertes par le composant logiciel sont beaucoup plus nombreuses. Le composant logiciel est une structure permettant d'encapsuler un code informatique, et de s'assurer que toutes les conditions sont réunies afin que le code fonctionne correctement. Cette encapsulation du code et cette sûreté d'exécution sont aussi des caractéristiques que l'on retrouve dans les objets.

Quelles sont alors les différences entre objets et composants ? Tout d'abord, les objets encapsulent des services, tandis que les composants sont des abstractions qui peuvent être utilisées pour construire des systèmes orientés objet .

On peut illustrer la différence entre objets et composants de la manière suivante :

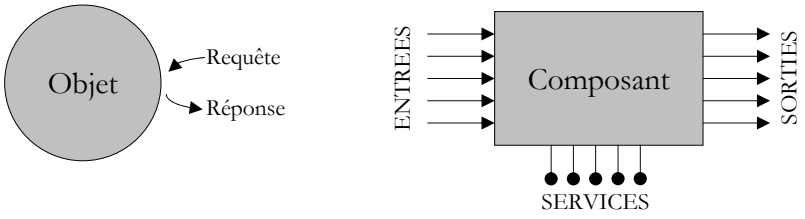


Figure I.10 : Objets et composants

Les objets ont une identité, un état et un comportement et sont toujours des entités n'existant que durant la phase de run-time[1]. Les composants sont des entités qui peuvent être utilisées lors de la phase de design-time[2], mais qui n'existent pas forcément lors du run-time. En effet, lorsqu'ils sont utilisés pour construire une application, les composants ne sont pas forcément instanciés et fournissent donc des services comme entité statique de l'application. Mais certains

[1] Run-time : phase d'exécution. Voir glossaire : phases de vie du logiciel.
[2] Design-time : phase de développement. Voir glossaire : phases de vie du logiciel.

composants peuvent aussi être instanciés et ainsi devenir des entités non statiques. Un objet peut être vu comme un type spécial de composant qui serait disponible au run-time.
Il est possible d'implémenter beaucoup de types de composants comme des objets, ce qui est source de confusion. En encapsulant les composants comme des objets, on atteint une grande flexibilité, puisque les composants peuvent être configurés et substitués lors du run-time. Cette notion est fondamentale pour tous les environnements interactifs basés sur des composants. D'un autre côté, cela ne veut pas dire que tous les objets sont utilisables en tant que composants, ou que tous les composants doivent être implémentés comme des objets. Des fonctions, des modules, et même des applications entières peuvent être vus comme des composants, et encapsulés comme tels. Des objets qui n'ont pas été conçus pour être connectés à d'autres objets ne peuvent pas être vus comme des composants.
Le composant informatique est donc la structure qui est la mieux adaptée pour supporter la phase de dimensionnement.

I.5.3. Les différentes interactions avec les composants

Il existe plusieurs aspects dans le cycle de vie d'un composant. Tout d'abord, le composant doit être créé. C'est ce qu'on appelle ici la génération ou l'encapsulation, suivant la manière dont le composant est crée.
Plusieurs utilisations du composant créé sont alors possibles. Ce composant peut être connecté avec d'autres composants, afin de créer soit de nouveaux composants plus complexes, soit être utilisés en coopération. C'est la composition. Un composant peut aussi être modifié pour être compatible avec de nouvelles interfaces. C'est la projection d'une interface vers une autre. Enfin, un composant est toujours amené à fournir les services qu'il propose. C'est la phase d'utilisation proprement dite du composant.

I.5.3.1. La création de composants

La création de composants peut être envisagée de deux manières différentes. Tout d'abord, le composant peut être crée de toutes pièces : c'est la génération complète. D'autre part, le composant peut être crée à partir d'un code déjà existant. On dit alors qu'on encapsule le code existant dans un composant.
Dans les deux cas, la structure du composant généré sera toujours la même, comme montré sur la figure I.11. Un composant est en effet composé de deux parties principales, le cœur de calcul et le code composant. Le cœur de calcul est l'entité qui fournit les différentes fonctionnalités dont le composant peut avoir besoin afin de satisfaire aux services qu'il doit fournir, et qui sont imposés par son interface. Afin de faire le lien entre le cœur de calcul et l'interface du composant, un code

spécifique au composant est nécessaire. Cette "glue" assure le passage des infos nécessaires au calcul entre les entrées du composant et le cœur de calcul (et éventuellement les adapte si besoin est), lance les calculs suivant le service du composant qui est appelé, puis récupère les résultats (et les adapte si besoin) puis les passe en sortie du composant.

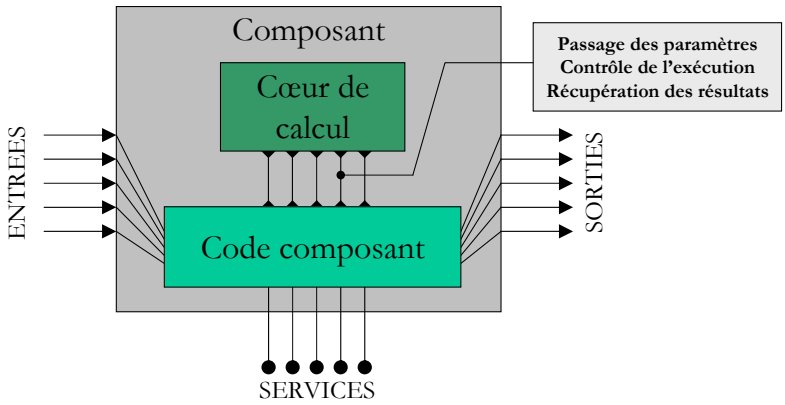


Figure I.11 : Structure interne d'un composant

La différence entre les deux façons de créer un composant se traduit par le fait que dans un cas, le cœur de calcul est à créer, alors que dans l'autre cas ce cœur est déjà disponible, et seul le code composant doit être créé. Dans le cas de l'encapsulation, plusieurs problèmes peuvent se poser, venant du fait que le cœur de calcul n'a pas forcément été créé pour répondre aux besoins spécifiques exprimés par l'interface du composant. Ainsi, des problèmes de sémantique, d'abstraction, de traduction, d'adaptation, ou autres peuvent apparaître [DEL]. Ces problèmes doivent être solutionnés dans le code composant car c'est lui qui assure l'adaptation entre le monde extérieur au composant (spécifié par l'interface du composant) et son monde intérieur (imposé par le cœur de calcul).

I.5.3.2. La projection de composants

La projection d'un composant vers une autre interface est en fait un changement de norme de ce composant. Le principe de la projection est illustré sur la figure I.12 :

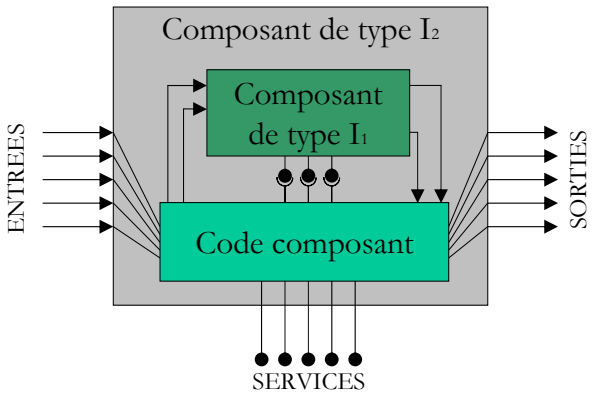


Figure I.12 : Principe de la projection

On part d'un composant satisfaisant à une certaine interface I_1, et on veut projeter ce composant vers une nouvelle interface I_2. Les problèmes posés par la projection sont en fait proches de ceux que l'on peut rencontrer dans le cadre de l'encapsulation d'un code de calcul existant. En effet, le changement de norme d'un composant peut être vu comme l'encapsulation de ce composant dans un nouveau composant, régi par une autre norme. On retrouve donc des problèmes de sémantique, de traduction, d'adaptation lors de la projection de composants [DEL].
On peut voir ici que le composant encapsulé est géré au travers de ses services et de ses entrées par le code du composant encapsulant. Puis les sorties du composant encapsulées sont récupérées et traitées par le code composant.

I.5.3.3. La composition de composants

La composition est l'idée forte qui a amené au concept de composant. En effet, chaque composant est considéré comme une brique de base, que l'on va relier à d'autres briques afin de concevoir un ensemble évolué capable d'effectuer des opérations complexes. L'idée de base est que l'on peut construire une application (un logiciel) comme on construit un circuit électronique. On connecte les composants les uns aux autres afin d'obtenir le comportement désiré. Dans notre approche, nous ne cherchons pas à construire des applications avec nos composants. Par contre, nous sommes intéressés par la composition pour pouvoir par exemple construire des modèles de dimensionnement par parties.
Il faut donc pouvoir relier les composants les uns aux autres. On peut ainsi créer des ensembles plus évolués. Par exemple, dans le cas de la modélisation d'un transformateur, plusieurs aspects différents peuvent être modélisés [PO2] :

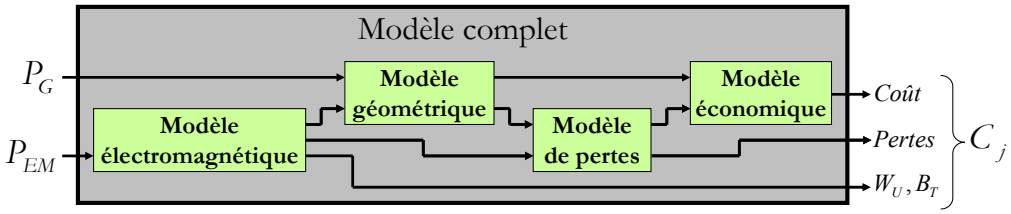


Figure I.13 : Le modèle composé du transformateur

- La partie géométrique de la modélisation est assez simple, et peut être exprimée de manière analytique.
- La partie électromagnétique, plus complexe, peut nécessiter une simulation éléments finis, même si un modèle analytique peut être suffisant dans certains cas.
- Le calcul de pertes peut être mené de manière simple et analytique une fois les résultats de la simulation électromagnétique connus.

- Le calcul économique du coût capitalisé du transformateur est un modèle analytique simple.

Le modèle de ce transformateur peut donc être décomposé en quatre sous modèles, chacun étant indépendant des autres.

Afin de créer un composant comprenant le modèle complet, on peut commencer par créer les quatre composants contenant chacun un sous modèle spécifique. La création séparée de chaque modèle est en effet plus facile que la création en un seul bloc du modèle complet. En effet, l'établissement de chaque modèle requiert des compétences différentes (électromagnétisme et électrotechnique pour le modèle électromagnétique, le modèle de pertes et le modèle géométrique, économiques pour le modèle économique). Des concepteurs différents peuvent donc établir chaque sous-modèle complètement séparément. Puis les composants contenant ces modèles sont centralisés, connectés les uns aux autres, et finalement le composant contenant le modèle complet est créé. De même, si on a commencé par utiliser un modèle analytique pour la partie électromagnétique, remplacer ce modèle analytique par un composant contenant une simulation par éléments finis est facile et rapide [DEL].

La composition apporte donc deux avantages décisifs au niveau de l'établissement des modèles, et au niveau de la capitalisation et de la réutilisation de ceux-ci.

I.5.4. Les différents types de composants existant

Au cours des travaux effectués dans l'équipe CDI du LEG, plusieurs types de composants sont apparus :

- Les composants de résolution des modèles de dimensionnement, existant avec différentes normes (COB[1], CoRe, CoSi [ALL]) répondant aux différents besoins apparus au cours des années.
- Les composants contenant des optimiseurs, afin de pouvoir mener les procédures d'optimisation [MAG].
- Les composants de visualisation et de post-processing, qui permettent de suivre l'évolution d'une structure durant l'optimisation et d'analyser les résultats obtenus.

Tous ces composants doivent pouvoir être pris en compte dans l'application qui sera développée durant les travaux. De plus, elle doit également pouvoir proposer certains services, comme par exemple un service de gestion des algorithmes numériques entrant en jeu lors de la résolution des modèles de dimensionnement.

[1] COB : acronyme pour Computational Object. Voir glossaire.

I.6. Solution proposée

Lors du dimensionnement, des outils doivent pouvoir aider le concepteur à obtenir des solutions d'une manière relativement simple et rapide. De nombreux travaux ont prouvé l'utilité de la modélisation analytique couplée à de l'optimisation [WUR] [ALB]. Pour cela, nous allons nous baser sur une approche de génération de code à partir d'un modèle du dispositif.
Afin de répondre à cette problématique, notre travail va se baser sur deux axes :

- La résolution des modèles de dimensionnement.
- L'approche composant pour le dimensionnement.

I.6.1. La résolution des modèles de dimensionnement

Nous allons nous attacher à résoudre des modèles de dimensionnement analytiques, pouvant notamment comprendre des systèmes d'équations différentielles.
En raison de l'utilisation de certains algorithmes d'optimisation, nous devons fournir non seulement la valeur des sorties du modèle de dimensionnement, mais aussi les sensibilités de ses sorties par rapport à ses entrées (le jacobien du modèle).
Nous proposons donc différentes approches pour la résolution des systèmes d'équations différentielles :

- Approches basées sur la résolution des systèmes d'état par exponentielle de matrice dans le cas des systèmes d'équations différentielles linéaires.
- Approches basées sur la dérivation de code pour les systèmes non linéaires et les modèles analytiques, et pour l'utilisation de code existant.

I.6.2. L'approche composant pour le dimensionnement

L'utilisation de composants logiciels durant la phase d'optimisation est une approche qui s'est imposée ces dernières années [DEL] [ALL] [MAG]. Il n'existe pas aujourd'hui, à notre connaissance, d'approche unifiée des composants et de leur utilisation pour le dimensionnement.
Le premier objectif de notre travail sur l'approche composant est donc de proposer une approche unifiée des composants pour le dimensionnement au travers de la spécification d'un standard de composants. Ce standard doit fournir la base de manipulation des composants, tout en restant suffisamment ouvert afin de pouvoir concilier les divers besoins relatifs au dimensionnement :

- Résolution des modèles de dimensionnement.
- Gestion des différents algorithmes numériques entrant en jeu dans la résolution des modèles.
- Support d'analyse et de post-processing.

Le deuxième objectif sera de développer un environnement logiciel permettant de gérer les différents aspects de l'utilisation des composants pour le dimensionnement, notamment au travers de trois aspects fondamentaux :

- La génération des composants.
- La composition.
- La projection.

CHAPITRE II

II.1. La résolution symbolique des modèles **39**
II.1.1. Valeur du terme intégral 40
II.1.1.1. Valeur du terme intégral pour une source constante 40
II.1.1.2. Valeur du terme intégral pour une source polynomiale 41
II.1.1.3. Valeur du terme intégral pour une source sinusoïdale 41
II.1.2. Solution générale de l'équation d'état 42
II.1.3. Evaluation de la trajectoire d'état 43
II.1.3.1. Introduction sur le calcul d'exponentielles de matrices 43
II.1.3.2. Algorithme de calcul d'exponentielle basé sur le développement de Taylor 46
II.1.3.3. Algorithme de calcul d'exponentielle basé sur les approximants de Padé 47
II.1.3.4. Amélioration des algorithmes par "scaling and squaring" 48
II.1.3.5. Importance du conditionnement des matrices 49
II.1.3.6. Conclusion sur l'évaluation de la trajectoire d'état 50
II.1.4. Calcul des critères de dimensionnement 51
II.1.4.1. Introduction sur le calcul des dérivées partielles 51
II.1.4.2. Première approche symbolique : dérivée de l'exponentielle 52
II.1.4.3. Deuxième approche symbolique : méthode de la matrice semi-circulante 53
II.1.4.4. Troisième approche symbolique : recombinaison du système d'état 53
II.1.4.5. Calcul de la valeur moyenne de la trajectoire d'état 54
II.2. La résolution numérique des modèles **55**
II.2.1. Principe de la dérivation de code 55
II.2.1.1. Les fonctions représentées par des programmes 55
II.2.1.2. Modélisation d'une fonction 56
II.2.1.3. La différentiation automatique en mode direct 58
II.2.2. Différentiation automatique en mode inverse 61
II.2.2.1. Approche par le graphe de calcul 61
II.2.2.2. Approche par substitution régressive 64
II.2.2.3. Approche par dualité 66
II.2.2.4.1. Algorithme ReG : approche par le graphe de calcul 68
II.2.2.4.2. Algorithme ReS : approche par substitutions régressives 69
II.2.2.4.3. Algorithme ReD : approche par dualité 69

CHAPITRE II

LA RESOLUTION DES MODELES DE DIMENSIONNEMENT

Comme on l'a vu précédemment, le dimensionnement d'un dispositif est basé sur l'interaction entre un algorithme d'optimisation et un modèle de dimensionnement. Nous allons ici nous intéresser à la résolution des modèles de dimensionnement. Les contraintes associées à cette résolution sont de plusieurs types. Tout d'abord, la résolution doit être fiable et robuste. En effet, si ce n'est pas le cas, la procédure d'optimisation n'est plus valide. De plus, la résolution doit être aussi rapide que possible, et ne doit utiliser que le minimum de mémoire. Ces contraintes viennent du fait qu'un processus d'optimisation est par nature itératif, et donc le nombre de résolutions du modèle va être élevé. Si le modèle est trop lent à calculer, ou occupe trop de mémoire, la procédure d'optimisation va s'allonger d'autant plus, et cela finit par devenir contre-productif. Enfin, la contrainte principale vient de l'utilisation possible d'algorithmes d'optimisation utilisant les gradients, ce qui impose le calcul de toutes les dérivées partielles des sorties du modèle de dimensionnement par rapport à ses entrées, c'est-à-dire le jacobien du modèle.

Afin d'obtenir la plus grande efficacité possible du calcul, le calcul d'une solution symbolique qui est ensuite évaluée aux points intéressants pour l'optimisation semble être bien adaptée. Le problème posé par cette technique est qu'il faut pouvoir calculer la solution symbolique. Quand l'approche symbolique ne peut pas être utilisée, on s'intéressera alors aux méthodes de résolution numériques (intégration pas à pas, éléments finis, simulations, …). Ces méthodes sont généralement plus lentes car elles nécessitent une quantité de calculs beaucoup plus importante que les méthodes symboliques. Elles occupent donc aussi beaucoup plus de mémoire. Mais elles ont l'avantage de résoudre des problèmes pour lesquels l'approche symbolique s'est révélée inefficace.

Ces deux approches pour la résolution sont complémentaires et il faudra donc les étudier toutes les deux.

II.1. La résolution symbolique des modèles

Afin de pouvoir résoudre symboliquement le modèle de dimensionnement, nous allons voir comment traiter les parties de la modélisation qui sont sous forme d'état. Comme on l'a vu au Chapitre I, un système d'état se présente sous la forme suivante :

$$f\left(X(t), \dot{X}(t), U(t), t\right) = 0 \qquad \text{(II.1)}$$

Sous cette forme, le système d'état ne peut absolument pas être résolu. Il existe alors deux possibilités. L'équation d'état peut être linéaire ou pas. Afin que nous puissions le résoudre, le système d'état doit se présenter sous la forme suivante :

$$\dot{X}(t) = A \cdot X(t) + B \cdot U(t) \tag{II.2}$$

Quand le système d'état peut se mettre sous cette forme, cela signifie alors que le modèle du dispositif est linéaire, c'est-à-dire que toutes les équations différentielles le composant sont des équations linéaires. Dans ce cas là, nous pouvons alors exploiter la résolution présentée au I.3.3. Cette résolution nous donne la forme générale de la solution de l'équation (II.2) :

$$X(t) = e^{At} \cdot X(0) + \int_0^t e^{A(t-\tau)} \cdot B \cdot U(\tau) d\tau \tag{II.3}$$

Nous allons nous baser sur cette formule, que nous évaluerons aux points demandés par l'algorithme d'optimisation.

Cette résolution va se décomposer en deux étapes. La première est une étape de calcul symbolique. En effet, le terme non homogène de la trajectoire d'état n'est pas directement exploitable en termes numériques. Il faut d'abord calculer symboliquement la valeur du terme intégral, puis évaluer la solution complète aux points spécifiés.

II.1.1. Valeur du terme intégral

Comme on ne peut pas calculer pour une source quelconque la valeur du terme intégral, nous allons calculer ce terme pour tous les différents types de sources que nous pouvons rencontrer dans le cadre de la conception de dispositifs électriques.

II.1.1.1. Valeur du terme intégral pour une source constante

On considère ici une seule source constante, c'est-à-dire :

$$U(t) = U \tag{II.4}$$

A partir de la solution générale de l'équation d'état (II.3), on peut en déduire :

$$X(t) = e^{At} \cdot X(0) + \int_0^t e^{A(t-\tau)} \cdot B \cdot U \cdot d\tau \tag{II.5}$$

On peut donc sortir tous les termes constants de l'intégrale, ce qui conduit à :

$$X(t) = e^{At} \cdot X(0) + \int_0^t e^{A(t-\tau)} d\tau \cdot B \cdot U \tag{II.6}$$

Après intégration, on obtient finalement :

$$X(t) = e^{At} \cdot X(0) + \left[A^{-1} \cdot e^{At} - A^{-1}\right] \cdot B \cdot U \tag{II.7}$$

Nous avons donc ici la solution complète de l'équation d'état dans le cas d'une seule source constante.

II.1.1.2. Valeur du terme intégral pour une source polynomiale

On considère maintenant une source polynomiale, c'est-à-dire :

$$U(t) = \sum_{k=0}^{d_U} c_k t^k \tag{II.8}$$

Ici, d_U est le degré du polynôme. Cette forme de source n'est pas rencontrée de manière classique dans la conception des dispositifs électriques. Par contre, savoir traiter ce genre de source permet de prendre en compte un grand nombre de sources constituées de polynômes définis par morceaux, comme par exemple une source délivrant un signal en triangle, ou en dents de scie. De plus, savoir traiter les polynômes permet, dans le cas où l'on rencontre une source inhabituelle, ou délivrant un signal singulier, de faire un développement en séries entières du signal délivré par la source, et ainsi de se ramener à une expression polynomiale.

En injectant l'expression de la source dans la solution de l'équation d'état, on obtient :

$$X(t) = e^{At} \cdot X(0) + \int_0^t e^{A(t-\tau)} \cdot B \cdot \left[\sum_{k=0}^{d_U} c_k \tau^k \right] \cdot d\tau \tag{II.9}$$

En inversant la somme et l'intégrale (par linéarité de l'intégrale), on obtient alors :

$$X(t) = e^{At} \cdot X(0) + \sum_{k=0}^{d_U} \int_0^t e^{A(t-\tau)} \cdot B \cdot c_k \tau^k d\tau \tag{II.10}$$

En sortant les paramètres constant de l'intégrale et en factorisant, il vient alors :

$$X(t) = e^{At} \cdot X(0) + \left[\sum_{k=0}^{d_U} c_k \int_0^t e^{A(t-\tau)} \tau^k d\tau \right] \cdot B \tag{II.11}$$

En intégrant, on obtient finalement :

$$X(t) = e^{At} \cdot X(0) + \left[\sum_{k=0}^{d_U} \left[-c_k A^{-(k+1)} \sum_{n=0}^{k} \left[k!\, e^{At} + \frac{(At)^n k!}{n!} \right] \right] \right] \cdot B \tag{II.12}$$

Nous avons donc exprimé la solution complète de l'équation d'état dans le cas d'une seule source polynomiale.

II.1.1.3. Valeur du terme intégral dans le cas d'une source sinusoïdale

On considère ici le cas d'une source sinusoïdale, c'est-à-dire :

$$U(t) = \sin(\omega t + \varphi) \tag{II.13}$$

En injectant l'expression (II.13) dans la solution générale de l'équation d'état et en intégrant, il vient :

$$X(t) = e^{At} \cdot X(0) + \left[A^2 + \omega^2 I \right]^{-1} \left[\begin{array}{c} e^{At} \left[I \cdot \omega \cos(\varphi) + A \cdot \sin(\varphi) \right] \\ - \left[I \cdot \omega \cos(\omega t + \varphi) + A \cdot \sin(\omega t + \varphi) \right] \end{array} \right] \cdot B \tag{II.14}$$

II.1.2. Solution générale de l'équation d'état

Nous allons maintenant nous placer dans le cas général, c'est-à-dire plusieurs sources de types différents, et nous allons exprimer la solution générale de l'équation d'état. On considère ici N états et N_S sources, chacune de ces sources pouvant être constante, sinusoïdale, ou polynomiale. Le vecteur d'entrée se présente donc de la manière suivante :

$$U(t) = \left[u_i(t)\right]_{i=\{1..N_S\}} \tag{II.15}$$

En injectant cette expression dans la solution générale de l'équation d'état, on obtient :

$$X(t) = e^{At} X(0) + \int_0^t e^{A(t-\tau)} \cdot B \cdot \left[u_i(t)\right]_{i=\{1..N_S\}} d\tau \tag{II.16}$$

Si on veut pouvoir découpler les termes liés aux différentes sources, il faut décomposer la matrice B suivant ses colonnes. On peut en effet écrire B sous la forme suivante :

$$B = \left[B_{i,j}\right]_{\substack{i=\{1..N_S\} \\ j=\{1..N\}}} \tag{II.17}$$

On peut alors poser les matrices B_{Mi} suivantes :

$$B_{Mi} = \begin{bmatrix} & B_{i,1} & \\ (0) & B_{i,j} & (0) \\ & B_{i,N} & \end{bmatrix} \tag{II.18}$$

Ces matrices sont telles que l'on peut écrire :

$$B = \sum_{i=1}^{N_S} B_{Mi} \tag{II.19}$$

En injectant cette relation dans la solution de l'équation d'état (II.16), on obtient :

$$X(t) = e^{At} \cdot X(0) + \int_0^t e^{A(t-\tau)} \cdot \left[\sum_{i=1}^{N_S} B_{Mi}\right] \cdot \left[u_i(\tau)\right]_{i=\{1..N_S\}} d\tau \tag{II.20}$$

En posant la famille de vecteurs suivante :

$$B_i = \begin{bmatrix} B_{i,1} \\ B_{i,j} \\ B_{i,N} \end{bmatrix} \tag{II.21}$$

On peut alors écrire la relation suivante :

$$\left[\sum_{i=1}^{N_S} B_{Mi}\right] \cdot \left[u_i(t)\right]_{i=\{1..N_S\}} = \sum_{i=1}^{N_S} B_i \cdot u_i(t) \tag{II.22}$$

La relation (II.20) devient donc :

$$X(t) = e^{At} \cdot X(0) + \int_0^t e^{A(t-\tau)} \cdot \left[\sum_{i=1}^{N_S} B_i \cdot u_i(\tau)\right] d\tau \tag{II.23}$$

On peut maintenant sortir la somme de l'intégrale (par linéarité de cette dernière), ce qui donne finalement :

$$X(t) = e^{At} \cdot X(0) + \sum_{i=1}^{N_S} \int_0^t e^{A(t-\tau)} \cdot B_i \cdot u_i(\tau) d\tau \qquad \text{(II.24)}$$

Cette solution est composée du terme homogène, et de la somme des termes non homogènes correspondant à chaque source traitée séparément, que l'on sait calculer, comme on l'a vu au paragraphe II.1.1. Cela revient en fait à appliquer le principe de superposition au système linéaire représenté par l'équation d'état. Grâce à cette formulation, on a obtenu l'expression symbolique de la solution complète de l'équation d'état pour le dispositif considéré.

II.1.3. Evaluation de la trajectoire d'état

Grâce au calcul formel présenté précédemment, nous avons obtenu une expression symbolique de la solution exploitable numériquement. Nous avons donc accompli la première étape de la résolution symbolique. Il nous reste à effectuer la deuxième étape, c'est-à-dire évaluer de manière numérique la valeur de la solution. La solution est une expression matricielle. Si la plupart des calculs matriciels ne posent aucun problème particulier, l'évaluation de l'exponentielle de matrice, ou de l'inversion d'une matrice peuvent se révéler problématiques à plusieurs niveaux. Tout d'abord, les résultats du calcul peuvent facilement être faux, ou du moins extrêmement imprécis, ce qui fait perdre toute validité au résultat final. De plus, certains algorithmes de calcul matriciel peuvent se révéler gourmands en temps de calcul ou en occupation mémoire.

II.1.3.1. Introduction sur le calcul d'exponentielles de matrices

Il existe un grand nombre d'algorithmes permettant d'évaluer l'exponentielle d'une matrice [VAN]. Parmi ces algorithmes, nous en avons sélectionné quelques-uns qui sont stables numériquement tout en restant rapides et légers au niveau mémoire.

Comme on l'a vu, l'opérateur d'exponentielle de matrice est défini comme étant la solution de l'équation différentielle suivante :

$$\dot{X}(t) = A \cdot X(t) \qquad \text{(II.25)}$$

Comme on l'a vu au paragraphe I.3.3, la solution de cette équation différentielle est donnée par :

$$X(t) = e^{At} \cdot X(0) \text{ avec } e^{At} = I + At + \frac{A^2 t^2}{2} + \cdots = \sum_{k=0}^{+\infty} \frac{A^k t^k}{k!} \qquad \text{(II.26)}$$

Les valeurs propres de la matrice dont on calcule l'exponentielle jouent un rôle fondamental dans l'étude de cette exponentielle, même si elles n'entrent pas directement en jeu dans tous les

algorithmes de calcul. Par exemple, si toutes les valeurs propres de A sont dans le demi-plan complexe de gauche, c'est-à-dire si leurs parties réelles sont négatives, alors on peut écrire :

$$e^{At} \xrightarrow[t\to+\infty]{} 0 \tag{II.27}$$

Cette propriété est souvent appelée "stabilité" en automatique, mais nous réserverons ce terme pour décrire les propriétés numériques d'un algorithme.

Il existe bien sûr des algorithmes adaptés à une classe particulière de matrices (comme par exemple des méthodes basées sur les décompositions par valeurs propres pour les matrices symétriques) mais comme dans notre cadre de travail nous n'avons pas à faire à une ou plusieurs classes particulières de matrices, ces algorithmes ne nous intéressent pas.

La difficulté inhérente à trouver des algorithmes de calcul d'exponentielle de matrice efficaces vient du problème suivant. Si on essaye d'exploiter les propriétés particulières venant de l'équation différentielle (II.24), on est naturellement amené à considérer les valeurs propres λ_i et les vecteurs propres v_i de A, ainsi que la représentation suivante :

$$X(t) = \sum_{i=1}^{n} a_i e^{\lambda_i t} v_i \tag{II.28}$$

Toutefois, il n'est pas toujours possible d'exprimer $X(t)$ de cette manière. S'il existe des valeurs propres confluentes[1], alors les coefficients a_i de la combinaison linéaire (II.28) peuvent avoir à être exprimés sous la forme de polynômes en t. En pratique, la zone grise dans laquelle les valeurs propres sont quasiment confluentes amène à des pertes de précision du calcul.

D'un autre côté, les algorithmes qui évitent l'utilisation explicite des valeurs propres ont tendance à requérir considérablement plus de temps de calcul. Ils peuvent aussi être affectés par des erreurs d'arrondi dans les cas où la matrice possède des éléments numériquement grands.

Ces difficultés peuvent être illustrées par un exemple simple. Si on pose la matrice A suivante :

$$A = \begin{bmatrix} \lambda & a \\ 0 & \mu \end{bmatrix} \tag{II.29}$$

Alors l'exponentielle de cette matrice est donnée par :

$$e^{At} = \begin{bmatrix} e^{\lambda t} & a\dfrac{e^{\lambda t} - e^{\mu t}}{\lambda - \mu} \\ 0 & e^{\mu t} \end{bmatrix} \tag{II.30}$$

Bien sûr, dans le cas où $\lambda = \mu$, cette représentation doit être remplacée par :

$$e^{At} = \begin{bmatrix} e^{\lambda t} & ate^{\lambda t} \\ 0 & e^{\lambda t} \end{bmatrix} \tag{II.31}$$

[1] Valeurs propres confluentes : valeurs propres proches mais non égales.

Il n'y a pas de difficultés sérieuses quand λ et μ sont exactement égales, ou même quand leur différence peut être considérée comme négligeable. Le problème peut être détecté et la forme adéquate de la solution alors utilisée. La difficulté survient lorsque $\lambda - \mu$ est petit mais non négligeable. Dans ce cas, la différence divisée suivante peut être difficile à calculer :

$$\frac{e^{\lambda t} - e^{\mu t}}{\lambda - \mu} \tag{II.32}$$

En effet, si elle est calculée de la manière la plus triviale, le résultat peut être calculé avec une erreur relative importante. Quand on le multiplie par a, le résultat final peut être extrêmement imprécis. Bien sûr, dans cet exemple, le terme non diagonal peut être écrit de différentes manières, qui sont beaucoup plus stables numériquement. Néanmoins, quand le même type de difficulté se présente dans des problèmes non triangulaires, ou des problèmes d'une taille supérieure à 2×2, sa détection et son traitement n'est absolument pas facile.

Cet exemple permet aussi d'illustrer une autre propriété de l'exponentielle qui doit être prise en compte par tout algorithme. Quand t croît, les éléments de l'exponentielle peuvent croître avant de diminuer. Dans notre exemple, si λ et μ sont négatives et que a est relativement grand, alors la courbe suivante est typique :

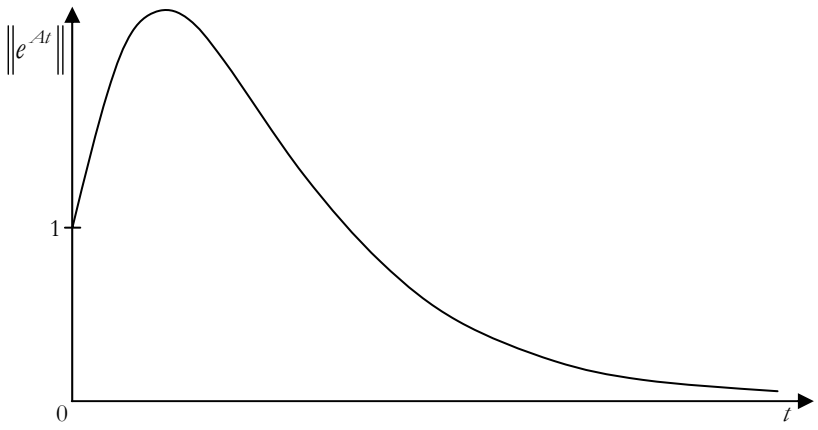


Figure II.1 : La "bosse" de l'exponentielle

Beaucoup d'algorithmes utilisent (directement ou indirectement) l'identité suivante :

$$e^{sA} = \left(e^{sA/m}\right)^m \tag{II.33}$$

La difficulté apparaît quand s/m est avant la bosse mais que s est après. En effet, dans ce cas là :

$$\left\|e^{sA}\right\| << \left\|e^{sA/m}\right\|^m \tag{II.34}$$

Malheureusement, les erreurs d'arrondi de la $m^{\text{ième}}$ puissance d'une matrice, par exemple P^m, sont généralement petites relativement à $\|P\|^m$ plutôt qu'à $\|P^m\|$. En conséquence, tout algorithme qui essayera de passer par dessus la bosse par des multiplications répétées sera en difficulté.

II.1.3.2. Algorithme de calcul d'exponentielle basé sur le développement de Taylor

Comme on l'a vu au paragraphe I.3.3, l'exponentielle d'une matrice peut se développer en séries entières de la manière suivante :

$$e^{A} = \sum_{k=0}^{\infty} \frac{A^k}{k!} \tag{II.35}$$

On peut se servir de ce développement afin d'évaluer numériquement la valeur de l'exponentielle d'une matrice. Comme cette somme comporte un nombre infini de termes, on ne peut évidemment pas la calculer entièrement. Il faut donc nous fixer une limite afin d'évaluer l'exponentielle, tout en s'assurant que le calcul est conduit de manière suffisamment précise. Il existe une relation entre la norme de la matrice dont on calcule l'exponentielle, le rang du calcul (c'est-à-dire l'indice jusqu'auquel est calculé la somme) et la précision finale du calcul [LIO]. C'est à partir de cette relation que nous allons calculer le rang optimal afin d'obtenir la précision souhaitée. Cette relation est la suivante :

$$0 \le \left(\frac{\|A\|^{N+1}}{(N+1)!} \right) \left(1 - \frac{\|A\|}{N+2} \right)^{-1} \le \varepsilon \tag{II.36}$$

Ici, A est la matrice dont on calcule l'exponentielle, N le rang du calcul et ε la précision du calcul. Connaissant A et ε, il suffit de choisir N assez grand pour que cette relation soit vérifiée. Le plus petit N qui vérifiera cette relation sera le rang optimal de calcul de l'exponentielle.

Au niveau de la conduite du calcul, il faut commencer par calculer le $N^{\text{ième}}$ terme de la somme en premier et remonter jusqu'au terme 0. En effet, les derniers termes de la somme (ceux de rang élevé) sont numériquement beaucoup plus petits que les premiers. Si on commence par additionner les premiers, il va arriver un moment où chaque terme qu'on ajoutera sera gommé par les imprécisions numériques de la machine. On appelle ce phénomène l'annulation numérique. Si au contraire, on commence par ajouter les derniers termes entre eux, l'accumulation peut se faire et la précision numérique du calcul sera bien meilleure.

Au final, la formule étant utilisée pour calculer l'exponentielle de matrice en se basant sur le développement de Taylor est la suivante :

$$e^{A} = \frac{A^N}{N!} + \frac{A^{N-1}}{(N-1)!} + \cdots + I = \sum_{k=N}^{0} \frac{A^k}{k!} \text{ avec } 0 \le \left(\frac{\|A\|^{N+1}}{(N+1)!} \right) \left(1 - \frac{\|A\|}{N+2} \right)^{-1} \le \varepsilon \tag{II.37}$$

Cet algorithme de calcul des exponentielles donne de bons résultats du point de vue numérique tant que la norme de la matrice dont on calcule l'exponentielle reste petite, c'est-à-dire inférieure à 1.

II.1.3.3. Algorithme de calcul d'exponentielle basé sur les approximants de Padé

L'approximation (p,q) de Padé de e^A est définie par :

$$R_{p,q}(A) = \left[D_{p,q}(A)\right]^{-1} N_{p,q}(A) \tag{II.38}$$

avec :

$$N_{p,q}(A) = \sum_{i=0}^{p} \frac{(p+q-i)!\, p!}{(p+q)!\, i!\,(p-i)!} A^i \tag{II.39}$$

$$D_{p,q}(A) = \sum_{j=0}^{q} \frac{(p+q-j)!\, q!}{(p+q)!\, j!\,(q-j)!} (-A)^j \tag{II.40}$$

Ici, la non singularité de $D_{p,q}(A)$ est assurée si p et q sont assez grands ou si les valeurs propres de A sont négatives. De manière générale, les approximants de Padé donnent de bons résultats, mais les erreurs d'arrondis incitent toujours à prendre les résultats avec précaution. Pour de grandes valeurs de p, l'approximant du numérateur $N_{p,p}(A)$ approche la série représentant $e^{A/2}$ et l'approximant du dénominateur $D_{p,p}(A)$ approche la série représentant $e^{-A/2}$. En conséquence, l'erreur d'arrondi peut empêcher la détermination précise de ces matrices. Des considérations similaires s'appliquent aux approximants généraux (p,q). En plus du problème d'arrondi, l'approximant du dénominateur peut être extrêmement mal conditionné en regard de l'inversion. Ceci est particulièrement vrai lorsque les valeurs propres de A sont largement distribuées. En effet, considérons à nouveaux les approximants diagonaux (p,p). Il n'est pas difficile de montrer que, pour des valeurs de p assez grandes, on a :

$$cond\left[D_{p,p}(A)\right] \approx cond\left[e^{-A/2}\right] \geq e^{(a_1 - a_n)/2} \tag{II.41}$$

où $a_1 \geq \cdots \geq a_n$ sont les parties réelles des valeurs propres de A et $cond(\ )$ représente le conditionnement d'une matrice. Les approximants de Padé peuvent être utilisés si la norme de la matrice dont on calcule l'exponentielle n'est pas trop grande. Dans ce cas, il y a plusieurs raisons pour lesquelles les approximants diagonaux sont préférés aux approximants non diagonaux. Supposons par exemple que $p < q$ (cas d'un approximant sous diagonal). L'évaluation de $R_{p,q}(A)$ requiert approximativement $q \cdot n^3$ flops[1] si A est de taille n, et cette évaluation est d'un ordre $p+q$. Toutefois, la même quantité de travail est nécessaire afin d'évaluer $R_{q,q}(A)$, et cette évaluation là est d'un ordre $2q$, qui est supérieur à $p+q$. Le même type d'argumentation peut s'appliquer aux approximants sur diagonaux ($p > q$).

[1] Flop : Floating point operation. Opération arithmétique de base dans un ordinateur. Voir glossaire.

Il y a encore d'autres raisons de préférer les approximants diagonaux. Si toutes les valeurs propres de A sont dans la partie gauche du plan, alors les approximants calculés avec $p > q$ ont tendance à avoir des erreurs d'arrondis plus grandes dues à l'annulation numérique, alors que les approximants calculés avec $p < q$ ont tendance à avoir des erreurs d'arrondi dues au mauvais conditionnement de la matrice $D_{p,q}(A)$.

II.1.3.4. Amélioration des algorithmes par "scaling and squaring"

Les difficultés dues aux erreurs d'arrondi et les coûts en termes de temps de calcul des méthodes de Taylor et de Padé s'accroissent quand la norme de la matrice dont on veut calculer l'exponentielle augmente, ou quand la distribution des valeurs propres de A s'élargit. Ces deux difficultés peuvent être contrôlées en exploitant une propriété fondamentale de l'exponentielle :

$$e^{A} = \left(e^{A/m}\right)^{m} \tag{II.42}$$

L'idée de base est de choisir m parmi les puissances de 2, et tel que $e^{A/m}$ puisse être calculé de manière fiable et efficace (c'est le "scaling", c'est-à-dire la mise à l'échelle), puis de former la matrice $\left(e^{A/m}\right)^{m}$ par élévations au carré successives (c'est le "squaring"). Un critère couramment utilisé pour le choix de m est de prendre la plus petite puissance de 2 qui soit telle que $\|A\|/m \leq 1$. Avec cette restriction, $e^{A/m}$ peut être calculé de manière satisfaisante, soit en utilisant le développement de Taylor, soit en utilisant les approximants de Padé.

Si l'exponentielle de la matrice mise à l'échelle ($A/2^{j}$) est calculée par l'approximant de Padé $R_{q,q}\left(A/2^{j}\right)$, alors il faut choisir deux paramètres, q et j. En Annexe II, on montre que si $\|A\| \leq 2^{j-1}$, alors :

$$\left[R_{qq}\left(A/2^{j}\right)\right]^{2^{j}} = e^{A+E} \tag{II.43}$$

avec :

$$\frac{\|E\|}{\|A\|} \leq 8\left[\frac{\|A\|}{2^{j}}\right]^{2q}\left(\frac{(q!)^{2}}{(2q)!(2q+1)!}\right) \tag{II.44}$$

Cette analyse inverse d'erreur peut être utilisée pour déterminer q et j de différentes manières. Par exemple , si ε est une tolérance d'erreur, on peut alors choisir entre les différentes paires (q, j) celle qui assurent, grâce à l'inégalité (II.44), la condition suivante :

$$\frac{\|E\|}{\|A\|} \leq \varepsilon \tag{II.45}$$

Puisque l'évaluation de l'approximant de Padé requiert approximativement $(q+j+1/3)n^3$ flops, il est logique d'essayer de choisir la paire pour laquelle $q+j$ est minimal. La table 1 spécifie les paires optimales pour différentes valeurs de ε et de $\|A\|$.

$\|A\|$ \ ε	10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-15}
10^{-2}	(1,0)	(1,0)	(2,0)	(3,0)	(3,0)
10^{-1}	(1,0)	(2,0)	(3,0)	(4,0)	(4,0)
10^{0}	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
10^{1}	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
10^{2}	(2,8)	(3,8)	(4,8)	(5,8)	(6,8)
10^{3}	(2,11)	(3,11)	(4,11)	(5,11)	(6,11)

Table II.1 : Paires optimales *(q,j)* pour différentes valeurs de précision et de norme

Ces paires ont été calculées à l'aide du corollaire du théorème présenté en Annexe II.

De manière générale, on s'aperçoit que le calcul d'exponentielle par les approximants de Padé est plus efficace que le calcul basé sur le développement de Taylor. Quand la norme de la matrice est petite, les approximants de Padé requièrent approximativement moitié moins de travail que le développement de Taylor pour obtenir la même précision. Cet avantage décroît lorsque la norme de la matrice augmente, à cause de la mise à l'échelle qui devient plus importante.

II.1.3.5. Importance du conditionnement des matrices

Comme on l'a vu, les algorithmes de calcul des exponentielles de matrices sont très sensibles à la norme des matrices, mais aussi à leur conditionnement. Le conditionnement d'une matrice peut se calculer de différentes manières. On peut par exemple le calculer à partir des valeurs singulières de la matrice. La décomposition en valeurs singulières d'une matrice est basée sur la relation suivante :

$$A = U^* \cdot D \cdot V \tag{II.46}$$

Ici, U et V sont des matrices ligne orthonormées, et U^* représente la transformée hermitienne de U. D est une matrice diagonale, dont les éléments diagonaux sont les valeurs singulières de A. En notant les valeurs singulières σ_i, et en classant ces valeurs singulières $\sigma_1 \le \sigma_2 \le \cdots \le \sigma_n$, alors on peut calculer le conditionnement de A de la manière suivante :

$$cond(A) = \left|\frac{\sigma_n}{\sigma_1}\right| \tag{II.47}$$

Le conditionnement de A traduit en fait la disparité au niveau numérique de ses éléments. Lorsque le conditionnement d'une matrice est trop grand, cela signifie que les éléments de cette

matrice sont largement distribués. Cela implique qu'au moment de la mise à l'échelle de la matrice (lors du "scaling and squaring"), certains éléments vont être annulés à cause d'une précision trop faible du calcul (phénomène d'annulation numérique). Le calcul de l'exponentielle se fait donc avec des erreurs beaucoup trop importantes, et donc le résultat n'est absolument pas correct.

Dans notre cadre de travail, l'obtention de matrices mal conditionnées vient directement de la modélisation du système étudié. Typiquement, le mauvais conditionnement des matrices peut provenir d'un mélange entre différents niveaux de modélisation. Plus on modélise un système finement, et plus on risque d'obtenir des éléments matriciels numériquement plus petits, mais aussi plus nombreux. Si on mélange deux niveaux de modélisation (une partie du système modélisé grossièrement et une partie beaucoup plus finement), on va accroître la disparité numérique des éléments, les plus grands provenant principalement de la modélisation grossière et les plus petits provenant de la modélisation fine. Cela va donc être la cause d'une forte disparité des éléments des matrices d'état, et donc de leur mauvais conditionnement. Il existe des méthodes permettant d'améliorer le conditionnement des matrices, comme par exemple la technique de l'équilibrage ou le raffinement itératif [TIT], mais ces méthodes n'ont pas été appréhendées ici. Il faudra donc que le niveau de modélisation des dispositifs étudiés soit cohérent, afin de ne pas introduire une trop grande disparité dans les éléments.

II.1.3.6. Conclusion sur l'évaluation de la trajectoire d'état

Comme on vient de le voir, les deux points bloquants pour l'évaluation de la trajectoire d'état, c'est-à-dire l'évaluation de l'exponentielle de matrice et de l'inverse de matrice, ont été résolus. L'utilisation d'algorithmes comme ceux de Taylor ou de Padé pour le calcul d'exponentielle, combinés avec la technique de "scaling and squaring", donne de bons résultats. A titre indicatif, le calcul des exponentielles de matrices dans Matlab est effectué par les approximants de Padé. L'utilisation de la méthode du pivot ou de l'inverse généralisé donne de bons résultats pour le calcul d'inversion. Comme nous nous sommes particulièrement intéressés au calcul d'exponentielles de matrices, nous avons récupéré les algorithmes d'inversion dans la bibliothèque HSL[1]. La fonction d'inversion par la méthode du pivot est la fonction MB01 et l'inversion généralisée est assurée par la fonction MB10.

Nous avons donc développé une méthode d'évaluation de la solution de l'équation d'état dans le cas linéaire qui est fiable et rapide. Mais ce n'est pas suffisant pour pouvoir dimensionner les dispositifs modélisés par cette représentation d'état. Il nous faut encore calculer les différentielles

[1] HSL : Harwell Subroutine Library, bibliothèque de fonctions de calcul numérique en Fortran77 développée par le Numarical Analysis Group, CSE, CCLRC

de la trajectoire d'état, de même que la valeur moyenne des états. En effet, la valeur moyenne des états est fréquemment utilisée comme critère de dimensionnement.

II.1.4. Calcul des critères de dimensionnement

Comme on l'a vu dans le Chapitre I, l'utilisation d'algorithmes d'optimisation par gradients impose que le modèle de dimensionnement fournisse non seulement la valeur des critères de dimensionnement mais aussi leurs dérivées partielles par rapport aux paramètres d'entrée du modèle. De même, certains autres critères de dimensionnement pouvant concerner des valeurs moyennes (de courant ou de tension par exemple), nous avons aussi besoin de pouvoir calculer la valeur moyenne des états du système. Un des autres critères de dimensionnement potentiellement intéressant peut être la valeur efficace des états. Malheureusement, il n'existe pas de méthode rendant ces valeurs efficaces directement accessibles depuis l'équation d'état. Nous allons donc étudier les méthodes permettant d'accéder aux dérivées partielles des états, ainsi qu'à leurs valeurs moyennes.

II.1.4.1. Introduction sur le calcul des dérivées partielles

Comme on l'a vu lors de l'évaluation de la trajectoire d'état, le point bloquant à l'évaluation des dérivées partielles de la trajectoire d'état va être le calcul de la dérivée de l'exponentielle. En effet, cette dérivée n'est pas calculable directement. La première technique envisageable afin de calculer cette dérivée est d'utiliser la méthode des différences finies. Cette méthode, purement numérique, a l'avantage d'être facilement programmable et de fonctionner sur tous les types de fonctions que l'on peut rencontrer. Mais elle est aussi extrêmement délicate à régler.
La méthode des différence finies est basée sur la formule suivante :

$$\frac{\partial f(x, y_i)}{\partial x} = \lim_{h \to 0} \left(\frac{f(x+h, y_i) - f(x, y_i)}{h} \right) \tag{II.48}$$

Il faut donc choisir un h suffisamment petit. Mais prendre une valeur trop petite pose aussi des problèmes. En effet, une valeur trop petite donne des résultats numériquement instables, souvent à cause de la différence qui est égale à 0, ce qui est dû au phénomène d'annulation numérique. Un point délicat lié à l'utilisation de cette méthode est donc le réglage du pas de différentiation h.
De plus, cette technique requiers au moins deux évaluations du modèle pour chaque point auquel on veut calculer la dérivée partielle, et donc elle est peu efficace en terme de rapidité de calcul. Nous allons donc devoir considérer d'autres approches afin d'évaluer la dérivée de l'exponentielle de matrices.

II.1.4.2. Première approche symbolique : dérivée de l'exponentielle

Dans une première approche symbolique, le principe est, à partir de la solution symbolique complète obtenue au II.1.2, de calculer, au moyen d'un dérivateur formel, la dérivée symbolique de la trajectoire d'état. Classiquement, en reprenant la solution (II.24) et en la dérivant par rapport à un paramètre d'entrée, on obtient :

$$\frac{\partial X(t)}{\partial P_i} = \frac{\partial e^{At}}{\partial P_i} \cdot X(0) + e^{At} \cdot \frac{\partial X(0)}{\partial P_i} + \sum_{i=1}^{N_S} \frac{\partial \int_0^t e^{A(t-\tau)} \cdot B_i \cdot u_i(\tau) d\tau}{\partial P_i} \tag{II.49}$$

Dans cette formulation, rien ne pose de problèmes particuliers à l'évaluation sauf la dérivée de l'exponentielle de matrice. En effet, on ne peut évaluer directement l'exponentielle. Ceci vient de la non commutativité du produit matriciel. Repartons du développement de Taylor de l'exponentielle :

$$\frac{\partial e^{At}}{\partial P_i} = \frac{\partial \sum_{k=0}^{+\infty} \frac{A^k t^k}{k!}}{\partial P_i} = \sum_{k=0}^{+\infty} \frac{t^k}{k!} \frac{\partial A^k}{\partial P_i} \tag{II.50}$$

Pour écrire ceci, on considère que $P_i \neq t$. En effet, si on veut calculer la dérivée temporelle de $X(t)$, il nous suffit d'appliquer l'équation d'état (II.2).

En explicitant l'expression de la dérivée des puissances de A, il vient :

$$\frac{\partial A^k}{\partial P_i} = \frac{\partial A}{\partial P_i} \cdot A^{k-1} + A \cdot \frac{\partial A^{k-1}}{\partial P_i} = \frac{\partial A}{\partial P_i} \cdot A^{k-1} + A \cdot \left(\frac{\partial A}{\partial P_i} \cdot A^{k-2} + A \cdot \frac{\partial A^{k-2}}{\partial P_i} \right) \tag{II.51}$$

Au final, en développant ainsi de terme en terme, on obtient :

$$\frac{\partial A^k}{\partial P_i} = \sum_{l=0}^{k-1} A^l \cdot \frac{\partial A}{\partial P_i} \cdot A^{k-l-1} \tag{II.52}$$

A partir des expressions (II.50) et (II.52), on peut obtenir l'expression de la dérivée de l'exponentielle :

$$\frac{\partial e^{At}}{\partial P_i} = \sum_{k=0}^{+\infty} \frac{t^k}{k!} \sum_{l=0}^{k-1} A^l \cdot \frac{\partial A}{\partial P_i} \cdot A^{k-l-1} \tag{II.53}$$

L'évaluation d'une telle somme ne posera pas de problèmes particuliers tant que les normes de la matrice A et de sa dérivée sont inférieures à 1. Malheureusement, la méthode du "scaling and squaring" ne peut être utilisée ici, car aucune relation du type de celle utilisée dans cette méthode ne peut être trouvée.

Cette approche directe ne peut donc pas être utilisée pour l'évaluation des dérivées partielles de la trajectoire d'état. Il nous faut donc nous orienter vers d'autres méthodes symboliques.

II.1.4.3. Deuxième approche symbolique : méthode de la matrice semi-circulante

Une méthode permettant d'obtenir les dérivées partielles de l'exponentielle de matrice consiste à utiliser la propriété suivante de l'exponentielle [NAJ] :

$$\text{Si on pose } S_A = \begin{bmatrix} A & \frac{\partial A}{\partial P_i} \\ 0 & A \end{bmatrix}, \text{ alors } e^{S_A} = \begin{bmatrix} e^A & \frac{\partial e^A}{\partial P_i} \\ 0 & e^A \end{bmatrix} \tag{II.54}$$

Grâce à cette méthode, on obtient donc directement l'exponentielle de la matrice ainsi que sa dérivée en un seul calcul. La matrice S_A est une matrice semi-circulante. En injectant les valeurs trouvées pour l'exponentielle et sa dérivée, on peut alors évaluer la valeur de la dérivée partielle de la trajectoire d'état.

Cette approche donne de bons résultats. Cependant, on peut pousser son principe plus loin et, au lieu de juste recombiner la matrice A afin d'obtenir une matrice semi-circulante, on peut recombiner le système entièrement et ainsi calculer directement la trajectoire d'état et sa dérivée.

II.1.4.4. Troisième approche symbolique : recombinaison du système d'état

Dans cette approche, on va directement dériver l'équation d'état (II.2) :

$$\frac{\partial \dot{X}(t)}{\partial P_i} = \frac{\partial A}{\partial P_i} \cdot X(t) + A \cdot \frac{\partial X(t)}{\partial P_i} + \frac{\partial B}{\partial P_i} \cdot U(t) + B \cdot \frac{\partial U(t)}{\partial P_i} \tag{II.55}$$

Avec cette formulation, et en prenant comme nouveau vecteur d'état le vecteur formé de la manière suivante :

$$\tilde{X}(t) = \begin{bmatrix} X(t) \\ \frac{\partial X(t)}{\partial P_i} \end{bmatrix} \tag{II.56}$$

Alors on peut recombiner les équations (II.2) et (II.55) afin de former un nouveau système d'état :

$$\begin{bmatrix} \dot{X}(t) \\ \frac{\partial \dot{X}(t)}{\partial P_i} \end{bmatrix} = \begin{bmatrix} A & 0 \\ \frac{\partial A}{\partial P_i} & A \end{bmatrix} \cdot \begin{bmatrix} X(t) \\ \frac{\partial X(t)}{\partial P_i} \end{bmatrix} + \begin{bmatrix} B & 0 \\ \frac{\partial B}{\partial P_i} & B \end{bmatrix} \cdot \begin{bmatrix} U(t) \\ \frac{\partial U(t)}{\partial P_i} \end{bmatrix} \tag{II.57}$$

Ce système d'état peut se ramener l'équation d'état recombinée suivante :

$$\dot{\tilde{X}}(t) = \tilde{A} \cdot \tilde{X}(t) + \tilde{B} \cdot \tilde{U}(t) \tag{II.58}$$

Cette équation est linéaire et peut donc être résolue par les méthodes et techniques présentées tout au long de ce chapitre. On peut noter que les matrices $\tilde{A}$ et $\tilde{B}$ sont des matrices semi-circulantes similaires à celle rencontrée en II.1.4.3.

Cette troisième approche permet, en recombinant tout le système d'état, de calculer en une seule fois la trajectoire d'état ainsi qu'une de ses dérivées partielles. Cette technique peut évidemment être étendue pour mener de front le calcul de plusieurs dérivées partielles :

$$\begin{bmatrix} \dot{X}(t) \\ \dfrac{\partial \dot{X}(t)}{\partial P_1} \\ \dfrac{\partial \dot{X}(t)}{\partial P_2} \end{bmatrix} = \begin{bmatrix} A & 0 & 0 \\ \dfrac{\partial A}{\partial P_1} & A & 0 \\ \dfrac{\partial A}{\partial P_2} & 0 & A \end{bmatrix} \cdot \begin{bmatrix} X(t) \\ \dfrac{\partial X(t)}{\partial P_1} \\ \dfrac{\partial X(t)}{\partial P_2} \end{bmatrix} + \begin{bmatrix} B & 0 & 0 \\ \dfrac{\partial B}{\partial P_1} & B & 0 \\ \dfrac{\partial B}{\partial P_2} & 0 & B \end{bmatrix} \cdot \begin{bmatrix} U(t) \\ \dfrac{\partial U(t)}{\partial P_1} \\ \dfrac{\partial U(t)}{\partial P_2} \end{bmatrix} \tag{II.59}$$

Au niveau de l'efficacité du calcul, composer un système trop gros réduit l'efficacité en allongeant le temps de calcul. On va donc se limiter à une recombinaison de systèmes simple (comme montré sur l'équation II.56) ou double (comme sur l'équation II.59) au maximum. Au-delà, le gain apporté par la résolution d'un système unique est annulé par l'allongement du temps de calcul dû à la taille du système. C'est donc cette approche que nous utiliserons par la suite.

II.1.4.5. Calcul de la valeur moyenne de la trajectoire d'état

Comme on l'a vu, certains critères de dimensionnement peuvent aussi concerner des valeurs moyennes de certains états. Afin d'assurer le calcul de ces critères de dimensionnement, il faut donc aussi pouvoir assurer le calcul de la valeur moyenne de la trajectoire d'état sur une durée T. Pour mener ce calcul, nous allons utiliser une approche similaire à la recombinaison du système d'état pour le calcul des dérivées. Pour ce faire, on va se baser sur la relation suivante :

$$\frac{\partial \int_0^t X(\tau) d\tau}{\partial t} = X(t) \tag{II.60}$$

En utilisant cette relation et en recombinant le système d'état, il vient :

$$\begin{bmatrix} \dot{X}(t) \\ \dfrac{\partial \int_0^t X(\tau) d\tau}{\partial t} \end{bmatrix} = \begin{bmatrix} A & 0 \\ I & 0 \end{bmatrix} \cdot \begin{bmatrix} X(t) \\ \int_0^t X(\tau) d\tau \end{bmatrix} + \begin{bmatrix} B \\ 0 \end{bmatrix} \cdot U(t) \tag{II.61}$$

Ce nouveau système est lui aussi linéaire, on peut donc le résoudre avec les méthodes présentées au cours de ce chapitre. Cette méthode permet donc d'obtenir directement la valeur moyenne de la trajectoire d'état au coefficient multiplicateur $1/T$ près.

II.2. La résolution numérique des modèles

Comme on l'a vu, la résolution symbolique d'une équation d'état est une méthode efficace pour le calcul des modèles de dimensionnement. Elle permet d'obtenir rapidement et de manière fiable la valeur des critères de dimensionnement ainsi que leurs dérivées partielles grâce à des recombinaisons symboliques de l'équation d'état. Mais cette méthode ne peut être utilisée que dans le cas de systèmes linéaires. En effet, dans le cas général, on ne sait pas résoudre symboliquement l'équation d'état. Quand on se retrouve confronté à ce cas, il faut alors se tourner vers des méthodes de résolution numériques. Ces méthodes sont connues et existent depuis longtemps. Notre propos n'est pas ici de créer de nouvelles méthodes numériques ou même d'améliorer les méthodes existantes. Le gros problème posé par l'optimisation par algorithmes par gradients est qu'il faut fournir non seulement les critères de dimensionnement mais aussi leurs dérivées par rapport aux entrées du modèle. Nous allons donc nous attacher ici au calcul des dérivées des critères de dimensionnement lors de la résolution numérique des modèles de dimensionnement.

Le problème posé ici est donc de calculer la dérivée d'un résultat fourni par un calcul numérique. Ce problème n'est pas nouveau en informatique, et plusieurs solutions sont envisageables afin d'y répondre. Tout d'abord, la technique la plus ancienne est la méthode des différences finies. Peu stable numériquement et difficile à paramétrer, on lui préfèrera d'autres techniques, comme par exemple la dérivation de code. En effet, comme nous n'avons plus à dériver une fonction au sens mathématique du terme, mais un programme informatique, nous pouvons dériver directement ce code de calcul. D'autres techniques existent aussi pour calculer les dérivées. On peut très bien coder les dérivées à la main, mais c'est un travail long, fastidieux et forcément source d'erreurs. De plus, nous situant dans l'optique de génération automatique d'un composant de calcul, nous ne pouvons pas demander à l'utilisateur de coder lui-même ses dérivées. Nous nous intéresserons donc à la différentiation de code.

II.2.1. Principe de la dérivation de code

II.2.1.1. Les fonctions représentées par des programmes

Nous ne supposons pas ici que la fonction que nous avons à dériver puisse être représentée symboliquement mais seulement au moyen d'un programme informatique. Nous allons nous attacher ici à dériver le code qui représente la fonction que nous voulons dériver. Ceci est en fait beaucoup plus général que la dérivation symbolique d'une fonction symbolique. En effet, il est permis que le code puisse présenter des instructions conditionnelles, des boucles, des appels à des

sous-fonctions, à des processus itératifs. Toutes ces structures n'ont évidemment pas de représentation symbolique. La notion de fonction représentée par un programme est donc beaucoup plus large que la notion de fonction représentée symboliquement.

Afin de comprendre le principe général de la dérivation de code, commençons donc par considérer le cas d'une fonction scalaire réelle de n variables :

$$f : \Re^n \longrightarrow \Re \tag{II.62}$$

L'hypothèse de base est que cette fonction est connue par un programme de calcul numérique qui en un point $x = (x_1, \ldots, x_n)$ de $\Re^n$ fournit la valeur de $f(x)$. On cherche alors un moyen de calculer, en un point $x \in \Re^n$, la valeur numérique du gradient de f.

Idéalement, un programme de différentiation automatique de code devrait pouvoir prendre en compte toute fonction représentée par un programme. Mais, comme on le verra, c'est une exigence trop forte, en particulier quand la fonction ne peut pas être représentée symboliquement, comme par exemple une fonction implicite dont le calcul nécessiterait la résolution d'un système d'équations non-linéaires par une méthode itérative. La structure informatique du programme représentant la fonction peut contenir des instructions de contrôle particulières (comme par exemple les structures conditionnelles *if – then – else* ou les structures répétitives comme *for* ou *do – while*) ou encore des appels à d'autres fonctions, à des sous-routines, etc… On comprend aisément qu'un programme calculant une fonction puisse être extrêmement complexe. Afin d'illustrer le principe de la différentiation de code, nous allons nous placer dans le cas où l'exécution du programme se fait de manière linéaire, c'est-à-dire sans instructions conditionnelles ni bouclages.

II.2.1.2. Modélisation d'une fonction

Un programme calculant une fonction f de n variables, que nous appellerons variables indépendantes, le fait en utilisant un certain nombre de variables auxiliaires $x_{n+1}, \ldots, x_N$, appelées variables intermédiaires. Si le programme est consistant, à chaque fois qu'une nouvelle variable intermédiaire x_i est introduite, sa valeur est calculée à partir d'autres variables dont la valeur doit être connue au moment du calcul. En adoptant une numérotation adéquate des variables, on peut supposer qu'une variable x_i possède un indice supérieur à celui des variables déjà calculées. Par conséquent, quand x_i sera calculée, ce sera au moyen d'une instruction d'affectation de la forme :

$$x_i = \varphi_i(x_1, \ldots, x_{i-1}) \tag{II.63}$$

Cette fonction φ_i est appelée fonction intermédiaire. Le programme manipule donc N variables (n variables indépendantes et $N-n$ variables intermédiaires) et la valeur de la fonction est récupérée dans la variable x_N.

Le modèle simple manque de réalité sur au moins deux points. Tout d'abord, un programme comporte des instructions de contrôle (instructions conditionnelles et boucles) qui font qu'il n'est pas possible de déterminer a priori l'ordre dans lequel les variables intermédiaires seront calculées. Cet ordre peut en effet dépendre des valeurs données aux variables indépendantes. Ensuite, il est fréquent de trouver dans les programmes des variables intermédiaires qui sont redéfinies, c'est-à-dire des variables qui reçoivent des valeurs différentes en divers points du programme.

Un programme obéissant au modèle simple admet une représentation sous forme de graphe de calcul (aussi appelée graphe de Kantorovitch [KAN]) :

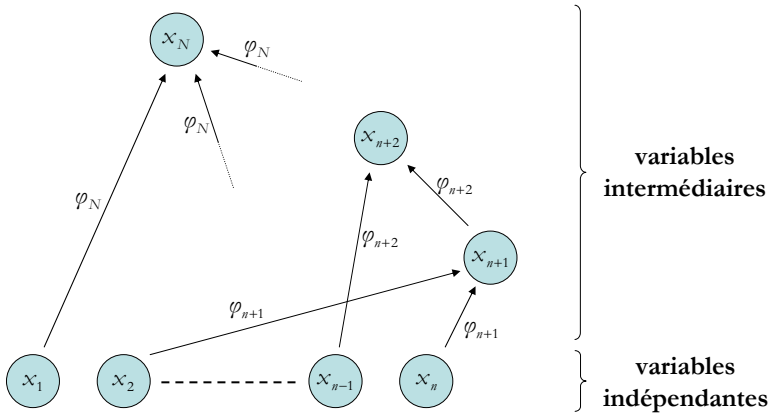


Figure II.2 : Graphe de Kantorovitch d'une fonction

Chaque nœud du graphe représente une variable (indépendante ou intermédiaire), et chaque arc allant d'un nœud x_i à un nœud x_j signifie que x_i est un paramètre de φ_j.

Un exemple (voir figure II.3) permet d'illustrer la construction du graphe de calcul :

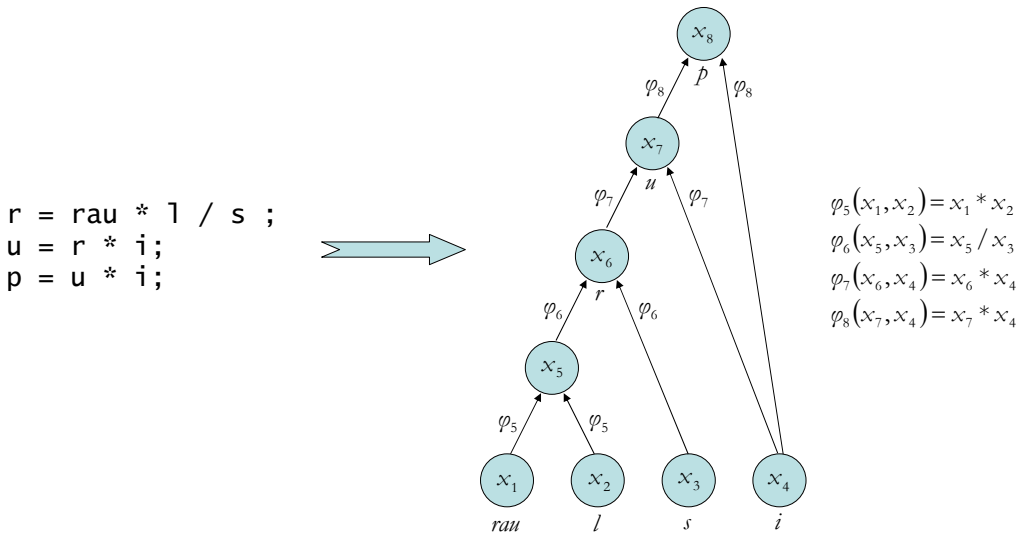


Figure II.3 : Du code au graphe de calcul

II.2.1.3. La différentiation automatique en mode direct

Le mode direct de la différentiation automatique consiste à calculer la fonction et ses gradients simultanément, en parallèle. Chaque fois qu'une variable intermédiaire est modifiée par l'instruction $x_i = \varphi_i(x_1,\ldots,x_j)$, son gradient ∇x_k est calculé en utilisant la règle de dérivation des fonctions composées. En effet, on peut écrire :

$$dx_i = \frac{\partial \varphi_i(x_1,\ldots,x_{i-1})}{\partial x_1} dx_1 + \cdots + \frac{\partial \varphi_i(x_1,\ldots,x_{i-1})}{\partial x_{i-1}} dx_{i-1} = \sum_{k=1}^{i-1} \frac{\partial \varphi_i(x_1,\ldots,x_k)}{\partial x_k} dx_k \qquad \text{(II.64)}$$

Grâce à cette formulation, on peut propager le calcul du gradient le long du graphe de calcul de la fonction, et ce calcul peut être mené en parallèle avec le calcul de la fonction. L'idée de base est la suivante. Désignons par $\delta_k x_i$ le gradient de la variable x_i (dépendante ou indépendante) par rapport à la variable indépendante x_k. On peut donc à chaque variable x_i associer le vecteur ayant n composantes ∇x_i formé des $\delta_k x_i$. Bien sûr, pour les variables indépendantes, on a :

$$\begin{cases} \delta_k x_i = 0 & \text{si } k \neq i \\ \delta_k x_i = 1 & \text{si } k = i \end{cases} \qquad \text{(II.65)}$$

Alors, en se servant de la règle de dérivation des fonctions composées, pour toutes les variables dépendantes, on peut écrire :

$$\nabla x_i = \sum_{k=1}^{i-1} \frac{\partial \varphi(x_1,\ldots,x_k)}{\partial x_k} \nabla x_k \qquad \text{(II.66)}$$

L'idée est de mener ce type de calcul pour chaque instruction du code original. Ce fonctionnement est illustré sur la figure II.4 :

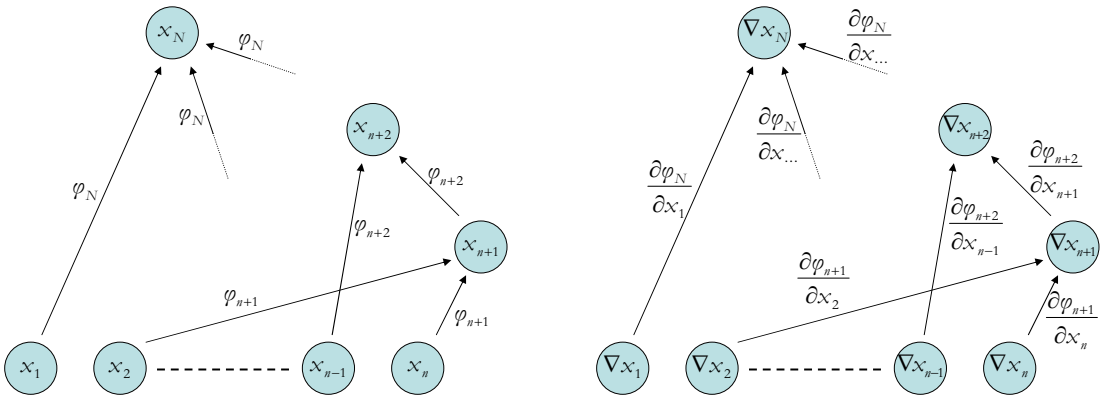


Figure II.4 : Fonctionnement du mode direct

On va donc propager le calcul de dérivée le long du graphe de calcul de la fonction, car grâce à l'hypothèse de consistance (qui est forcément assurée en mode direct) du code qui nous assure que toutes les quantités présentes dans le membre de droite de (II.66) sont connues au moment de son évaluation. On voit qu'il suffit de compléter le programme original par quelques instructions supplémentaires et sans modifier l'ordre d'exécution. On dit alors que le calcul du

gradient est propagé de façon progressive. Ce fonctionnement peut être illustré en reprenant l'exemple de la figure II.3 :

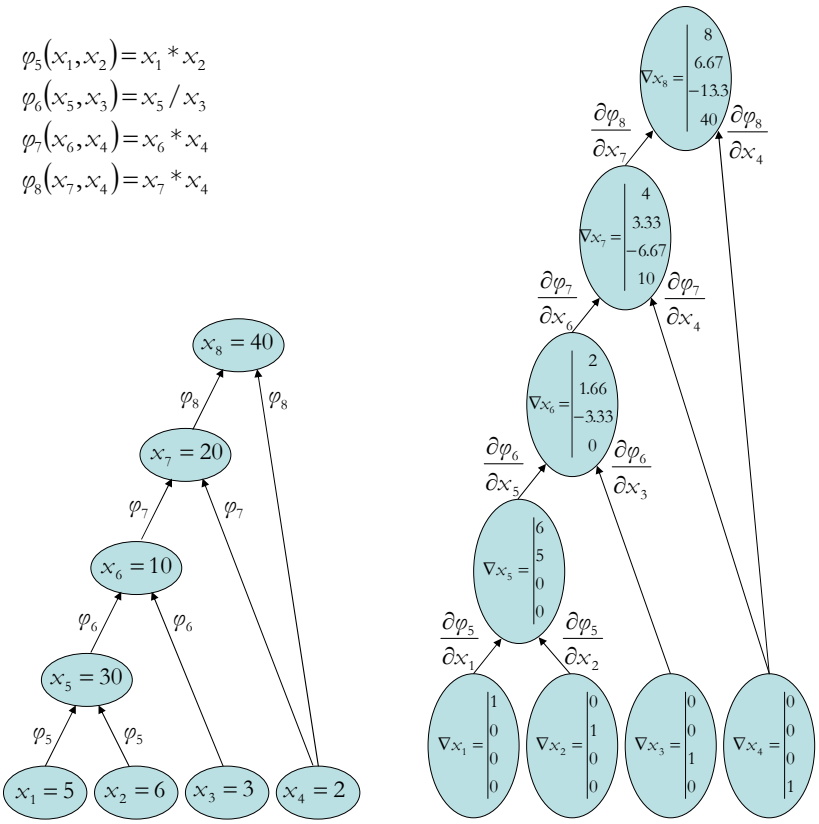


Figure II.5 : Application du mode direct

Outre sa simplicité théorique, ce système de différentiation est particulièrement adapté à la technique dite de surcharge d'opérateurs. Le principe de cette méthode est de modifier les fonctions calculant les opérations arithmétiques élémentaires (+, -, *, /, *sin*, *cos*, *tan*, *log*, …), et d'intégrer au calcul du résultat de l'opération le calcul des gradients. L'intérêt majeur de cette technique de surcharge d'opérateurs est que le code de calcul original n'a à subir que très peu de modifications pour être dérivé, puisque les fonctions ne changent pas de nom, seul leur code change et cette modification est totalement transparente du point de vue de leur utilisation. Par contre, ce système de différentiation pèche par son efficacité qui est extrêmement réduite, puisque beaucoup de calculs inutiles vont être menés.

Afin d'améliorer cet aspect de la méthode, on peut utiliser une pondération du graphe de calcul, mais cela nous interdit de mener l'évaluation du gradient en parallèle avec l'évaluation de la fonction proprement dite.

Représentons le graphe de calcul de la fonction en pondérant chaque arc par les valeurs $\varphi_{k,j}$ définies par :

$$\varphi_{k,j} = \frac{\partial \varphi_k}{\partial x_j} \tag{II.67}$$

Ce graphe de calcul pondéré est représenté sur la figure II.4 :

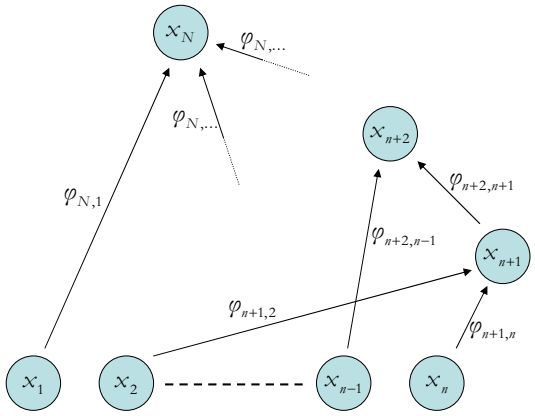


Figure II.6 : Graphe de calcul pondéré

Ainsi, à chaque nœud i, on associe la valeur de la variable x_i et son gradient ∇x_i formé de ses dérivées partielles par rapport à x_k, les $\delta_k x_i$. Au départ (en bas du graphe), les valeurs des $\delta_k x_i$ sont initialisées à $\delta_{k,i}$[1]. Mais au lieu de mener directement le calcul des gradients en parallèle avec celui de la fonction, on va se contenter de construire ce graphe pondéré pendant l'exécution de la fonction. Le temps de calcul de la i^{ème} dérivée partielle de la fonction (celle correspondant à la variable indépendante x_i) sera bien moindre si on ne parcourt que la partie du graphe dépendant de x_i. Pour cela, au lieu de parcourir le graphe en regardant pour chaque nœud de quelles variables il dépend, il suffit de regarder quelles sont les variables qu'il influence : ces variables sont repérées par le fait que l'arc qui les relie à ce nœud présente une pondération non nulle. Cela permet d'éviter toutes les opérations inutiles concernant la contribution nulle d'un nœud dans une branche du graphe.

La différence fondamentale entre ces deux algorithmes est que l'un permet de mener le calcul de la dérivée en parallèle du calcul de la fonction, tandis que le deuxième nécessite un calcul de la fonction afin de pouvoir établir son graphe de calcul, puis évalue la dérivée. Le premier algorithme est relativement économe en place mémoire tandis que le deuxième doit enregistrer la trace de toutes les opérations menées lors du calcul de la fonction, ce qui peut devenir rapidement extrêmement encombrant au niveau mémoire si les fonctions dont on calcule la dérivée sont un peu complexes. Par contre, le deuxième algorithme sera beaucoup plus efficace et plus rapide pour des fonctions complexes que le premier.

En conclusion sur le mode direct de différentiation automatique de code, c'est une technique simple et assez rapide à mettre en œuvre. Elle donne globalement de bons résultats et peut se révéler efficace sous certaines conditions (degré de linéarité du code suffisante, nombre de

[1] $\delta_{k,i}$: Symbole de Kronecker (vaut 1 si $k = i$ et 0 sinon)

variables indépendantes réduit et nombre de variables dépendantes important). Malgré cela, le mode direct n'est pas le plus prometteur de la différentiation automatique. En effet, le mode inverse de différentiation, s'il ne peut pas être mené en parallèle du calcul de la fonction, permet en revanche une analyse plus fine du graphe de calcul et se montre souvent beaucoup plus efficace que le mode direct.

II.2.2. Différentiation automatique en mode inverse

Nous venons d'illustrer le principe de la dérivation de code en explicitant le mode direct de différentiation. Ce mode est le plus simple à comprendre mais n'est pas le plus efficace. Le mode direct calcule en fait l'influence d'une entrée sur les sorties. Il existe d'autres algorithmes de dérivation de code, basé sur le mode inverse de différentiation. Le mode inverse calcule en fait la sensibilité d'une sortie par rapport aux entrées. Cette méthode, qui s'apparente au calcul des gradients par la méthode de l'état adjoint dans les problèmes de contrôle optimal, possède un intérêt majeur. Elle est plus efficace que les algorithmes s'inspirant du mode direct de différentiation. En effet, le rapport entre le temps de calcul nécessaire à l'évaluation de la fonction et de ses dérivées sur le temps de calcul de la fonction est majoré par une constante indépendante de la complexité de la fonction, ce qui signifie que le temps de calcul de la dérivée est borné, et cette borne dépend uniquement du temps de calcul de la fonction.

Plusieurs approches de la différentiation automatique en mode inverse sont possibles. Nous allons étudier ces approches ainsi que leur fonctionnement afin de pouvoir déterminer quelle approche est la plus adaptée à nos problèmes. Nous commencerons par étudier l'approche par le graphe de calcul, historiquement la plus ancienne, puis nous analyserons l'approche par substitution régressive et l'approche par dualité.

II.2.2.1. Approche par le graphe de calcul

Cette approche est l'une des premières à avoir été développée pour ce qui concerne le mode inverse [SAW] [IRI]. C'est aussi sur cette approche que sont construits les différentiateurs automatiques comme ADOL-C [GRI].

Reprenons le graphe de calcul de la fonction présenté sur la figure II.3. D'après la formule de dérivation des fonctions composées, la dérivée partielle de la fonction par rapport à une variable indépendante x_k peut être obtenue en faisant la somme sur tous les chemins menant de x_k à x_N des produits des quantités $\varphi_{k,j}$ associés aux arcs rencontrés sur chaque chemin :

$$\frac{\partial x_N}{\partial x_k} = \sum_{\substack{chemin\ \zeta \\ de\ x_k\ à\ x_N}} \prod_{\substack{arc(i,j) \\ \in \zeta}} \varphi_{i,j} \qquad \text{(II.68)}$$

Il suffit en effet d'appliquer la relation (II.66) répétitivement en n'en considérant que la $k^{\text{ième}}$ composante afin d'obtenir cette formule. Le mode inverse est basé sur le fait que lorsqu'on applique la formule (II.68) pour tous les k, des produits des quantités $\varphi_{k,j}$ peuvent être calculés plusieurs fois. Le graphe de calcul particulier montré sur la figure II.5 illustre ce cas de figure :

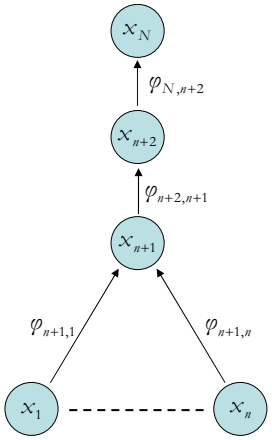


Figure II.7 : Un graphe de calcul particulier

Dans ce cas, le graphe de calcul de la fonction montre que tous les chemins menant des variables indépendantes (les nœuds de x_1 à x_n) au résultat x_N possèdent une partie commune. A cause de cette partie commune, la quantité $\varphi_{n+2,n+1}\varphi_{N,n+2}$ apparaît dans toutes les dérivées partielles de la fonction :

$$\frac{\partial x_N}{\partial x_k} = \left(\varphi_{n+2,n+1}\varphi_{N,n+2}\right)\varphi_{n+1,k} \tag{II.69}$$

Dans ce cas particulier, on peut éviter le calcul répétitif de la dérivée de la partie commune des chemins menant des variables indépendantes au résultat en accumulant non plus les seules quantités $\varphi_{k,j}$ sur les arcs mais plutôt leurs produits à partir du haut du graphe de calcul.

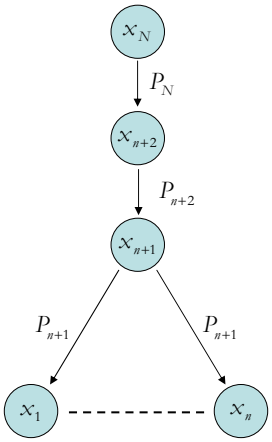


Figure II.8 : Repondération du graphe de calcul

Ici, les valeurs des pondérations sont les suivantes :

$$\begin{cases} P_N = 1 \\ P_{n+2} = \varphi_{N,n+2} P_N \\ \vdots \\ \dfrac{\partial x_N}{\partial x_k} = \varphi_{n+1,k} P_{n+1} \end{cases} \tag{II.70}$$

Cette manière de calculer correspond bien à un parcours du graphe de haut en bas, à partir de son sommet x_N. Bien entendu, cette méthode n'est pas avantageuse en permanence, en particulier dans les cas où le graphe de calcul est formé de la manière suivante :

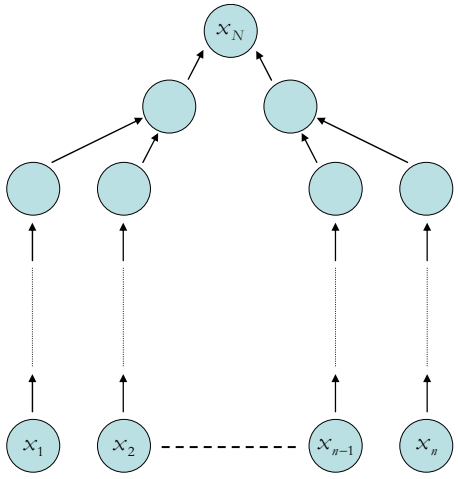


Figure II.9 : Autre graphe de calcul particulier

Mais de manière générale, on gagnera tout de même en efficacité en utilisant la dérivation de code en mode inverse.

Le fonctionnement général de cette approche est donc d'associer à chaque nœud x_k une quantité P_k définie par :

$$P_k = \sum_{\substack{\text{chemin } \zeta \\ \text{de } x_k \text{ à } x_N}} \prod_{\substack{\text{arc}(i,j) \\ \in \zeta}} \varphi_{i,j} \tag{II.71}$$

Cette formule est analogue à celle donnant les dérivées partielles de x_N par rapport à x_k. Quand k est inférieur à n (c'est-à-dire que le nœud représente une variable indépendante), la quantité P_k est alors égale à la dérivée partielle de la fonction par rapport à cette variable. Cette quantité P_k représente donc la sensibilité de la fonction par rapport à une variable intermédiaire (dépendante ou indépendante) x_k. En initialisant le graphe (c'est-à-dire en affectant la valeur de P_N à 1), on peut ensuite, grâce aux valeurs des $\varphi_{i,j}$, remonter le graphe et déterminer les valeurs des P_k de manière régressive. On commence donc par initialiser tous les P_k à 0 (sauf P_N), et ensuite il suffit de parcourir le graphe vers le bas et pour chaque nœud k rencontré, il faut

prendre en compte la contribution de P_k à P_j pour tous les nœuds j faisant partie des influents de k. Cette contribution s'écrit :

$$P_j = \sum_{\substack{k \text{ pour les} \\ j \text{ influents de } k}} \frac{\partial \varphi_k}{\partial x_j} P_k \tag{II.72}$$

Au final, la dérivée partielle de la fonction par rapport à une variable x_k se récupère dans la quantité P_k.

Ce fonctionnement peut être illustré sur l'exemple de la figure II.3 :

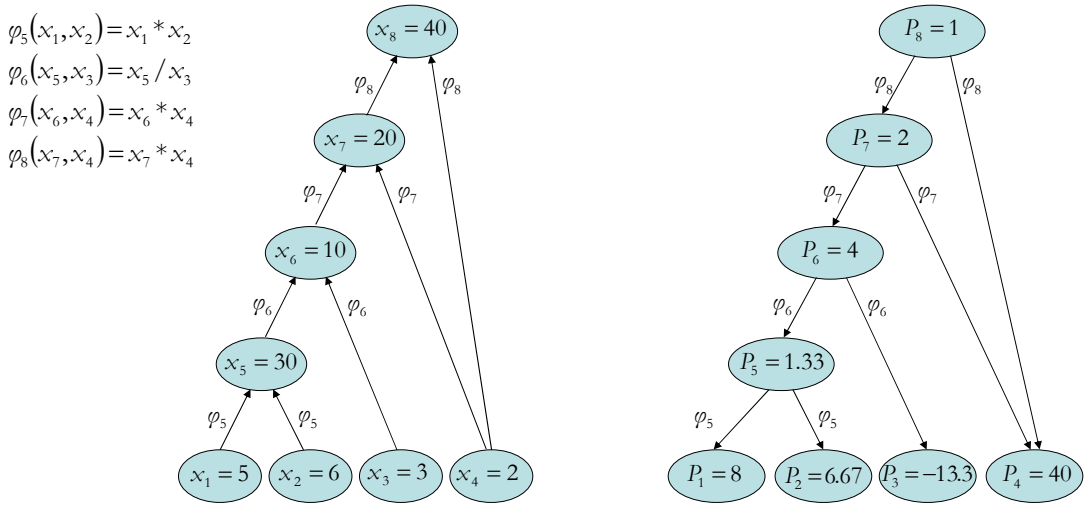


Figure II.10 : Application de l'approche par le graphe de calcul

II.2.2.2. Approche par substitutions régressives

Dans cette approche, on va s'intéresser à la manière dont la variable x_N varie lorsque la variable x_i varie, les variables $x_1,\ldots,x_{i-1}$ étant fixées à des valeurs données et la variation des variables $x_{i+1},\ldots,x_N$ se faisant par l'intermédiaire des fonctions $\varphi_{i+1},\ldots,\varphi_N$. Il va donc être intéressant pour nous d'introduire les fonctions ψ_i, définie comme étant la fonction résultant des $N-i$ dernières instructions du programme. Ces fonctions sont illustrées sur la figure II.11 :

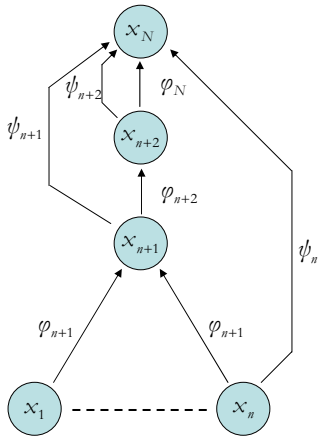


Figure II.11 : Ajout des fonctions ψ_i sur le graphe de calcul

Grâce à ces fonctions ψ_i, on va pouvoir remonter aux dérivées partielles de la fonction représentée par le graphe de calcul.

On commence par définir la fonction ψ_N :

$$\psi_N(x_1,\ldots,x_N) = x_N \tag{II.73}$$

On peut alors écrire, pour tous les $k \in \{N-1, N-2, \ldots, n\}$, les formules de récurrence régressive suivantes :

$$\psi_k(x_1,\ldots,x_k) = \psi_{k+1}(x_1,\ldots,x_k,\varphi_{k+1}(x_1,\ldots,x_k)) \tag{II.74}$$

On va alors chercher à déterminer la valeur des p_i qui sont les dérivées partielles des fonctions ψ_i définies de la manière suivante :

$$p_i(x_1,\ldots,x_i) = \frac{\partial \psi_i(x_1,\ldots,x_i)}{\partial x_i} \tag{II.75}$$

Les variables p_i représentent donc la sensibilité de x_N par rapport à x_i lorsque $x_1,\ldots,x_{i-1}$ sont fixées. Lorsque la valeur des variables $x_1,\ldots,x_n$ est donnée et que les variables intermédiaires $x_{n+1},\ldots,x_N$ ont été calculées par l'algorithme, on note $p_i \equiv p_i(x_1,\ldots,x_i)$.

Enfin, si on note $f(x_1,\ldots,x_n)$ la fonction représentée par l'algorithme, alors on peut écrire :

$$\psi_n(x_1,\ldots,x_n) = f(x_1,\ldots,x_n) \tag{II.76}$$

Les relations (II.73), (II.74) et (II.76) permettent de calculer de façon régressive p_N, p_{N-1}, …, p_{n+1} et ensuite les dérivées partielles de f par rapport à x_n, x_{n-1}, …, x_1. En effet, à partir de la relation (II.73), on tire :

$$p_N = 1 \tag{II.77}$$

Ensuite, on peut dériver (II.74) en s'assurant que $1 \le i \le N-1$ et que $max(i,n) \le k \le N-1$, et on obtient la relation suivante :

$$\frac{\partial \psi_k(x_1,\ldots,x_k)}{\partial x_i} = \frac{\partial \psi_{k+1}(x_1,\ldots,x_{k+1})}{\partial x_i} + p_{k+1} \cdot \frac{\partial \varphi_{k+1}(x_1,\ldots,x_k)}{\partial x_i} \tag{II.78}$$

Supposons que $n+1 \le i \le N-1$ (nous considérons le cas d'une variable intermédiaire). Alors, en sommant les relations (II.78) pour tous les $k \in \{i,\ldots,N-1\}$ et en notant que :

$$\frac{\partial \psi_N(x_1,\ldots,x_N)}{\partial x_i} = 0 \tag{II.79}$$

On trouve alors la relation suivante :

$$p_i = \sum_{k=i+1}^{N} \frac{\partial \varphi_k(x_1,\ldots,x_{k-1})}{\partial x_i} p_k \text{ pour } n+1 \le i \le N-1 \tag{II.80}$$

Plaçons nous maintenant dans le cas où $1 \le i \le n$ (c'est-à-dire qu'on considère une des variables indépendantes). Alors, en sommant les relations (II.78) pour tous les $k \in \{n,\ldots,N-1\}$, on obtient :

$$\frac{\partial \psi_n}{\partial x_i} = \sum_{k=n}^{N-1} \frac{\partial \varphi_{k+1}(x_1,\ldots,x_k)}{\partial x_i} p_{k+1} \tag{II.81}$$

A partir de cette relation et en utilisant (II.76), on trouve finalement :

$$\frac{\partial f(x_1,\ldots,x_n)}{\partial x_i} = \sum_{k=n+1}^{N} \frac{\partial \varphi_k(x_1,\ldots,x_{k-1})}{\partial x_i} p_k \text{ pour } 1 \le i \le n \tag{II.82}$$

Grâce aux relations (II.77), (II.80) et (II.82), on peut calculer toutes les dérivées partielles de la fonction, c'est-à-dire son gradient, à partir du calcul des p_i.

Lors de l'exécution, le graphe de calcul des gradients sera identique à celui montré sur la figure II.10. Seule la manière de mener le calcul est différente.

II.2.2.3. Approche par dualité

Cette approche se base sur l'idée suivante. Au lieu de considérer la fonction f représentée par l'algorithme, fonction des n variables indépendantes $x_1,\ldots,x_n$, on considère maintenant une fonction $\bar{f}$ dépendant des N variables $x_1,\ldots,x_n,\ldots,x_N$, définie par :

$$\bar{f}(x_1,\ldots,x_N) = x_N \tag{II.83}$$

Les variables $x_1,\ldots,x_N$ sont reliées par les relations suivantes :

$$c_i(x_1,\ldots,x_N) \equiv x_i - \varphi_i(x_1,\ldots,x_{i-1}) = 0 \text{ pour } n+1 \le i \le N \tag{II.84}$$

Il faut s'assurer ici que le fait de remplacer les instructions d'affectation $x_i = \varphi_i(x_1,\ldots,x_{i-1})$ par les équations (II.84) est possible, c'est-à-dire que les valeurs des variables indépendantes $x_1,\ldots,x_n$ étant données, le système formé par les équations (II.84) admet une solution unique telle que la valeur de x_N soit identique à celle obtenue par l'algorithme qui représente la fonction. L'existence d'une solution satisfaisant à cette condition est facile à prouver. Il suffit en effet de résoudre les équations (II.84) une à une pour $i = n+1, i = n+2,\ldots, i = N$, comme le ferait l'algorithme. Pour la même raison, il est clair qu'il ne peut y avoir d'autres solutions au système formé par les équations (II.84).

Dans cette approche, il s'agit en fait de calculer le gradient de la fonction $\bar{f}(x_1,\ldots,x_N)$, les variables $x_1,\ldots,x_N$ étant soumises aux contraintes c_i. Cette situation se rencontre fréquement dans les problèmes de contrôle optimal en automatique. En continuant d'utiliser l'analogie entre l'algorithme et un système, les variables indépendantes $x_1,\ldots,x_n$ peuvent être vues comme les

variables de contrôle, et les variables dépendantes $x_{n+1},\ldots,x_N$ peuvent être vues comme les variables d'état. Pour calculer le gradient de $\bar{f}$ par rapport aux variables de contrôle, il est possible d'utiliser la méthode de l'état adjoint [GOC]. On va associer un multiplicateur de Lagrange p_i (pour $i \in \{n+1,\ldots,N\}$) à chaque contrainte c_i et on introduit le lagrangien ℓ défini par :

$$\ell(x_1,\ldots,x_N,p) \equiv \bar{f}(x_1,\ldots,x_N) - \sum_{k=n+1}^{N} c_k(x_1,\ldots,x_N)p_k = x_N - \sum_{k=n+1}^{N}\left(x_k - \varphi_k(x_1,\ldots,x_{k-1})\right)p_k \qquad \text{(II.85)}$$

où le vecteur p est défini comme l'état adjoint et est constitué de la manière suivante :

$$p = \begin{bmatrix} p_{n+1} \\ \vdots \\ p_N \end{bmatrix} \qquad \text{(II.86)}$$

En un point $(x_1,\ldots,x_N)$ donné, on calcule l'état adjoint p au moyen des équations adjointes :

$$\frac{\partial \ell(x_1,\ldots,x_n,p)}{\partial x_i} = 0 \text{ pour } n+1 \le i \le N \qquad \text{(II.87)}$$

Pour tout $i \in \{n+1,\ldots,N\}$, on trouve donc :

$$\sum_{k=n+1}^{N}\left(\delta_{i,k} - \frac{\partial \varphi_k}{\partial x_i}\right)p_k = \delta_{i,N} \qquad \text{(II.88)}$$

Ici, $\delta_{i,k}$ est le symbole de Kronecker. Comme on a :

$$\frac{\partial \varphi_k}{\partial x_i} = 0 \text{ pour } i \ge k \qquad \text{(II.89)}$$

Les équations (II.88) forment donc un système linéaire d'ordre $N-n$ en p dont la matrice est triangulaire supérieure avec une diagonale formée de 1. Ce système est donc inversible (ce qui assure en même temps la validité de cette méthode de l'état adjoint) et peut se résoudre de façon régressive :

$$\begin{cases} p_N = 1 \\ p_i = \displaystyle\sum_{k=i+1}^{N} \frac{\partial \varphi_k}{\partial x_i} p_k \end{cases} \qquad \text{(II.90)}$$

On retrouve donc tous les multiplicateurs de Lagrange p_i pour $i \in \{n+1,\ldots,N\}$. Les dérivées partielles par rapport aux variables indépendantes $x_1,\ldots,x_n$ sont les mêmes pour f et $\bar{f}$ et s'obtiennent de la manière suivante :

$$\frac{\partial f(x_1,\ldots,x_n)}{\partial x_i} = \frac{\partial \ell(x_1,\ldots,x_N)}{\partial x_i} = \sum_{k=n+1}^{N} \frac{\partial \varphi_k}{\partial x_i} p_k \text{ pour } 1 \le i \le n \qquad \text{(II.91)}$$

On a donc bien obtenu les dérivées partielles de la fonction par rapport aux variables d'entrée.

Ici encore, seule la manière de mener le calcul est différente, le graphe de calcul restant identique à celui de la figure II.10.

II.2.2.4. Implémentations algorithmiques

Les trois approches présentées pour la différentiation en mode inverse sont trois justifications théoriques différentes du même principe mathématique. En effet, d'un point de vue théorique, la différentiation en mode inverse revient à calculer les coefficients p_i pour chaque nœud du graphe de calcul. Cependant, elles conduisent à différents algorithmes de calcul des dérivées. Nous allons maintenant voir les différentes implémentations de ces trois approches. Elles seront comparées en terme de performance au paragraphe IV.4.

II.2.2.4.1. Algorithme ReG : approche par le graphe de calcul

Ici, le principe de l'algorithme est d'établir le graphe de calcul du programme à dériver. On va donc stocker en mémoire les différents nœuds du graphe, ainsi que les arcs les reliant. On peut ensuite remonter le graphe de la manière présentée au paragraphe II.2.2.1. L'implémentation se traduit de la manière suivante :

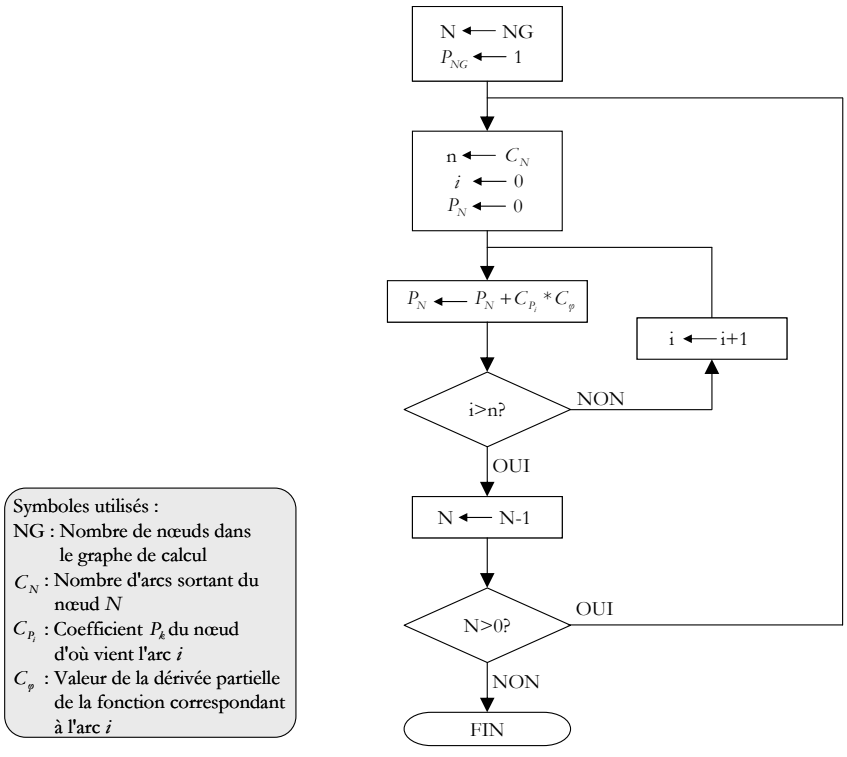


Figure II.12 : Algorithme ReG de calcul des dérivées en mode inverse

II.2.2.4.2. Algorithme ReS : approche par substitutions régressives

Dans cette approche, on ne va pas stocker le graphe de calcul en tant que tel (c'est-à-dire les nœuds et les arcs les reliant) mais plutôt la liste des opérations élémentaires dans l'ordre dans lequel elles ont été effectuées. Il ne reste plus ensuite à l'algorithme qu'à remonter cette liste afin d'obtenir les valeurs des dérivées.

L'algorithme ReS va donc se dérouler de la manière suivante :

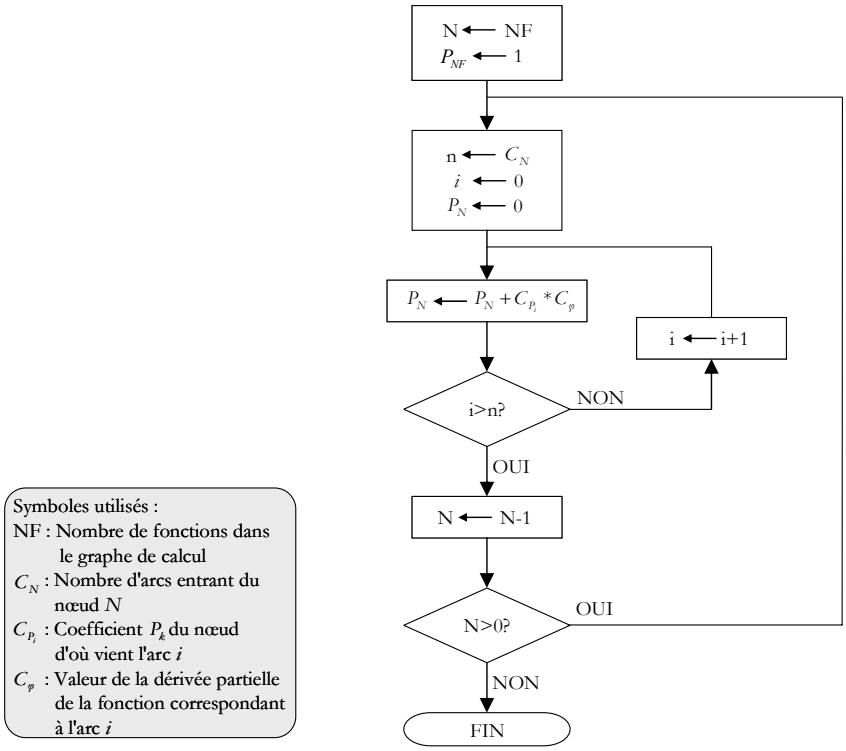


Figure II.13 : Algorithme ReS de calcul des dérivées en mode inverse

II.2.2.4.3. Algorithme ReD : approche par dualité

L'idée de cette approche est non pas de calculer les dérivées directement à partir de l'analyse du graphe de calcul de la fonction, mais plutôt d'établir le système d'équations linéaires (II.90) reliant les p_i entre eux. Ce système admet une solution unique (l'existence de cette solution étant assurée par l'hypothèse de consistance), et peut donc être résolu par des méthodes matricielles classiques [PRE].

CHAPITRE III

III.1. Vers un nouveau standard de composant **73**
III.1.1. Hétérogénéité des composants 73
III.1.1.1. Hétérogénéité de constitution 74
III.1.1.2. Hétérogénéité de fonctionnement 74
III.1.1.3. Conséquences de cette hétérogénéité 75
III.1.2. Besoins établis et émergents des composants 76
III.1.2.1. Mécanisme de chargement 76
III.1.2.2. Mécanisme d'introspection 77
III.I.2.3. Support du schizomorphisme 77
III.1.2.4. Evolutivité du standard 79
III.1.2.5. Portabilité du standard 79
III.1.3. Spécification d'un standard de composants pour la conception 79
III.1.3.1. Choix du langage de base du standard 80
III.1.3.2. Jeu d'interfaces structurant le standard 80
III.1.3.3. Norme des fichiers 83
III.1.3.4. Outils de lecture, d'écriture, et d'introspection directe 84
III.1.4. Spécification des services dans le cadre d'ICAr 86
III.1.4.1. Service de résolution des modèles de dimensionnement 86
III.1.4.2. Service de visualisation et de post-processing 89
III.1.4.3. Service de gestion des algorithmes 90
III.1.4.4. Autres services 90
III.2. Réalisation d'un environnement d'aide à la conception **91**
III.2.1. Spécifications de l'environnement 91
III.2.1.1. Fiabilité, maintenabilité, évolutivité, modularité et portabilité 91
III.2.1.2. Ouverture 92
III.2.1.3. Principe d'utilisation de CoreLab 93
III.2.2. Architecture générale 94
III.2.3. Module de génération des composants de calcul 95
III.2.3.1. Principe de fonctionnement 95
III.2.3.2. Fonctionnement de nos générateurs dédiés 97
III.2.4. Module de composition 100
III.2.4.1. Différentes compositions possibles 100
III.2.4.2. Composition hétérogène 101
III.2.4.3. Architecture du module de composition 103
III.2.5. Module de projection 103

CHAPITRE III

APPROCHE COMPOSANT POUR LE DIMENSIONNEMENT

Dans le chapitre précédent, nous avons posé des bases théoriques et méthodologiques de résolution des modèles de dimensionnement. Mais ces bases ne trouvent leur utilité que grâce à la réalisation informatique, qui va rendre possible leur utilisation pour un ingénieur de bureau d'études. Dans cette partie, nous aborderons donc les différents choix, architectures et réalisations informatiques que nous avons fait au cours de ces travaux.

Dans une première partie, nous commencerons par exposer le standard de composants informatiques que nous avons spécifié : ICAr. Afin de répondre aux différents besoins apparus au cours des recherches menées dans l'équipe CDI ces dernières années, il est composé d'un modèle objet de composants et d'une norme pour ces composants.

Dans une deuxième partie, nous aborderons la réalisation et le fonctionnement de l'environnement d'aide à la conception produit durant ces travaux : CoreLab. Cet environnement a pour but la gestion des composants de modèles pour l'optimisation. Cette gestion comprend différents aspects. Tout d'abord, la génération des composants de calcul doit pouvoir être effectuée, soit à partir d'un fichier contenant un modèle, soit à partir d'un calcul existant que l'on encapsule dans un composant. Ensuite, le concepteur doit pouvoir projeter des composants existants d'une norme vers une autre. Enfin, il faut disposer d'un environnement qui rend possible la composition des composants entre eux afin de pouvoir créer des modèles élaborés et complexes.

III.1. Vers un nouveau standard de composants

III.1.1. Hétérogénéité des composants

Un des premiers problèmes auxquels nous avons du faire face durant ces travaux est la grande hétérogénéité des divers composants que nous avons rencontrés, ainsi que celle de leurs fonctionnalités et celle de leurs comportements. En effet, on comprend bien qu'un composant contenant un algorithme d'optimisation va avoir une constitution et un mode opératoire extrêmement différent d'un composant contenant un modèle de dimensionnement, ou encore un composant de visualisation contenant une vue paramétrée d'un dispositif.

III.1.1.1. Hétérogénéité de constitution

Au niveau de leur constitution tout d'abord, les composants peuvent présenter de fortes disparités. En effet, certains composants peuvent être écrits en Java, d'autres en C / C++, d'autres encore en Fortran ou d'autres langages, et certains même des mélanges de ces différents langages. Mais cette hétérogénéité de langage n'est pas la seule au niveau de la constitution des composants. Ceux-ci peuvent encore différer par leur architecture, ou par la norme d'encapsulation à laquelle ils répondent. Nous devrons donc par la suite prendre en compte cette grande diversité structurelle et constitutionnelle des composants.

III.1.1.2. Hétérogénéité de fonctionnement

III.1.1.2.1. Chargement des composants

Les composants présentent des différences au niveau de leur fonctionnement et de la manière de s'en servir. Un composant contient tout ce dont il a besoin pour fonctionner de manière autonome[1]. Pour autant, cela ne signifie pas que tout soit automatiquement disponible ou prêt à fonctionner dès que l'on utilise un composant. En effet, la première étape de l'utilisation d'un composant est le chargement et l'introspection. Cette étape prépare le composant à effectuer le service pour lequel il a été crée.

L'introspection consiste à aller découvrir ce qu'est le composant, à quel type de composant il appartient, et ainsi comment interagir avec lui. Puis les codes à exécuter sont chargés en mémoire. Par exemple, cela correspond au chargement des librairies dynamiques dans le cas de code C / C++ ou Fortran, ou au chargement des différentes classes dans la machine virtuelle pour du code Java. Certaines parties du composant doivent subir un traitement spécifique avant d'être utilisées, comme la préparations des librairies dynamiques avant leur chargement en mémoire, ou le chargement des sous-composants par exemple. On comprend aisément que cette étape de chargement soit fortement dépendante de la constitution du composant.

III.1.1.2.2. Utilisation des composants

Les composants diffèrent aussi au niveau de leur utilisation par leur interface. Chaque composant implémente en effet une interface, qui définit le service que rend ce composant. On comprend bien que pour chaque type de service qui peut être rendu par les composants correspond une interface qui le définit. Par exemple, l'interface du composant de résolution de modèles, qui

[1] Voir la définition du composant par [SZY] : "unité autonome de déploiement d'un code informatique"

contient la résolution d'un modèle de dimensionnement (méthode `resolve`) ainsi que celui de ses dérivées (méthode `differentiate`) se présente de la manière suivante :

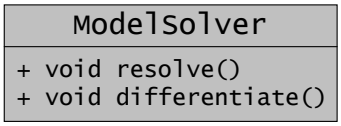


Figure III.1 : Interface du composant de résolution des modèles

L'interface du composant d'optimisation développé durant la thèse de David Magot [MAG] présente, quant à lui, une interface tout à fait différente :

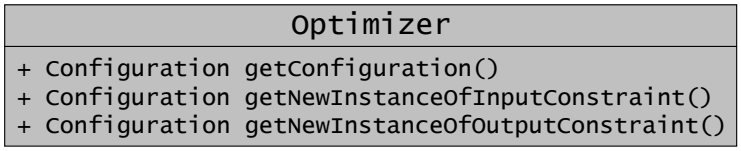


Figure III.2 : Interface du composant d'optimisation

On voit donc bien ici que ces deux interfaces sont complètement différentes. Cela est normal puisque ces deux composants ne rendent absolument pas le même type de service. Il existe donc ici une nouvelle hétérogénéité des composants au niveau de leur utilisation.

III.1.1.3. Conséquences de cette hétérogénéité

Nous venons de voir à quel point les composants peuvent être hétérogènes, tant au niveau de leur constitution que de leur fonctionnement. De plus, cette hétérogénéité est croissante au fur et à mesure que les composants abordent de nouveaux domaines de la conception en rendant de nouveaux services. Ainsi, l'utilisation des composants devient délicate. Jusqu'à présent, aucune tentative d'uniformisation ou de standardisation des différents composants utilisés dans la conception en Génie Electrique n'avait été effectuée. Chacun développait son propre type de composant, répondant à ses besoins particuliers, mais sans aucune vision globale concernant l'intégration et l'utilisation de son composant. En conséquence, pour traiter un nouveau type de composant, il fallait bien évidemment prendre en compte la nouvelle interface définissant le service rendu par le composant. Cependant, il fallait aussi écrire un nouveau mécanisme de chargement adapté pour ce type de composant, et modifier en conséquence les différents logiciels utilisateurs de ce composant.

Nous avons donc proposé et défini un standard de composant afin de disposer d'une base unifiée pour la création et l'utilisation des divers composants utilisés lors de la conception. Certaines spécifications de ce standard sont aussi issues de la confrontation de notre vision avec les besoins ressentis par d'autres développeurs du laboratoire. Plusieurs standards de composants existent déjà mais ne convenaient pas à nos besoins. Le premier standard et le plus répandu en terme d'utilisation est le standard COM développé par Microsoft [COM]. Ce standard est à la base du fonctionnement de Windows et définit le comportement et l'utilisation des composants

Windows. Récemment, l'importance de ce standard s'est encore accrue par l'arrivée de la plateforme ".NET" qui rend plus facile la création de composants au standard COM à partir de plusieurs langages de programmation par l'utilisation d'un système de bytecode[1] unifié. Au début de nos travaux, ".NET" n'existait pas et COM était principalement basé en C / C++, même s'il permet l'utilisation d'autres langages. De plus, COM étant particulièrement adapté à Windows, nous avons donc choisi de ne pas nous appuyer sur ce standard. Un autre standard disponible est celui proposé dans Corba[2], développé par l'OMG[3] [COR]. Corba rend possible l'utilisation de composants fonctionnant sur des machines différentes et avec des systèmes d'exploitation différents. En contrepartie, c'est un système plus lourd à mettre en place, car plus générique que Java. De plus, les environnements de conception que nous proposons sont basés en Java, même s'ils utilisent des pièces logicielles provenant d'autres langages. Nous n'avions donc pas la nécessité de recourir à Corba. Le troisième standard de composant disponible est JavaBean, le standard de composant développé par Sun [BEA]. Ce standard est particulièrement adapté au développement d'applications grâce aux composants, ce qui n'était pas notre optique d'utilisation des composants.

Nous avons donc choisi de développer notre propre standard de composants, adapté à nos besoins en conception, et plus particulièrement en dimensionnement, et permettant d'unifier les différents composants que nous sommes amenés à utiliser, ou tout du moins de leur donner une base commune simplifiant leur utilisation.

III.1.2. Besoins établis et émergents des composants

Afin de développer un standard adapté à nos besoins actuels, mais aussi futurs, en termes d'utilisation et de services rendus par les composants, nous devons analyser les différents besoins apparus au cours des recherches menées dans l'équipe CDI. Certains besoins sont identifiés depuis longtemps et ont été à l'origine des premiers travaux sur les composants. D'autres besoins sont apparus au cours des récents travaux menés dans le laboratoire (applications de dimensionnement, développement d'outils logiciels) et doivent maintenant être pris en compte pour le nouveau standard.

III.1.2.1. Mécanisme de chargement

L'utilisation de tout composant commence par le chargement de ce composant. Il faut donc fournir un mécanisme de chargement facile à utiliser et unifié pour tous les composants. En effet,

[1] Bytecode :Code informatique exécuté par un programme plutôt que par le processeur. Voir glossaire.
[2] Corba :Common Object Request Broker Architecture. Voir glossaire.
[3] OMG : Object Management Group : http://www.omg.org

comme nous l'avons vu précédemment, il n'existe pas à l'heure actuelle de mécanisme de chargement commun aux différents composants utilisés dans nos travaux. Cela tient essentiellement au fait que chaque composant est régi par sa propre norme. Chaque composant possède donc sa propre structure, et nécessite subséquemment un mécanisme de chargement adapté à sa norme et à sa constitution.

Il nous faut donc développer une norme de composant permettant de mettre en place un système de chargement des composants unifié, et facile à utiliser. Le fait de disposer d'une norme commune à tous nos composants nous aidera fortement à satisfaire à cette contrainte.

III.1.2.2. Mécanisme d'introspection

Rappelons tout d'abord que l'introspection est une démarche d'observation intérieure du composant. Elle permet de découvrir les services que le composant peut offrir ainsi que les moyens d'y accéder. Par exemple, lors de l'utilisation d'un composant de résolution de modèles, les services rendus par le composant sont déjà connus à la base : ces services sont le calcul des sorties du modèle de dimensionnement ainsi que leurs différentielles. Par contre, les entrées et les sorties du modèle ne sont pas connues. Il faut donc utiliser un mécanisme d'introspection, afin de pouvoir lister les entrées et les sorties du composant.

Dans l'exemple que nous venons de citer, les services rendus par le composant de résolution de modèles sont déjà connus à la base. Le mécanisme d'introspection peut donc se limiter aux entrées et sorties du modèle, et d'ailleurs le système d'interfaces définissant le composant de résolution de modèles inclut ce mécanisme. Un problème majeur de cette manière de procéder est la non généricité du mécanisme. En effet, ce n'est pas parce qu'on sait introspecter un composant de résolution de modèles que l'on saura introspecter un composant Optimizer.

Un mécanisme d'introspection générique doit donc être mis en place. Il doit permettre de connaître les services rendus par le composant (par exemple le composant est-il un composant de résolution de modèles, ou un composant d'optimisation ?) et pour les services rendus par le composant les moyens d'y accéder (lister les entrées et sorties du composant de résolution de modèles).

III.1.2.3. Support du schizomorphisme

Un des besoins relatifs aux composants et qui a émergé ces dernières années est le schizomorphisme (du grec *schizo*, "séparé", et *morphè*, "forme"). Le schizomorphisme représente le caractère multiple des composants. Comme on l'a vu précédemment, un composant rend des services. Un même composant doit pouvoir rendre plusieurs types de services différents. Une

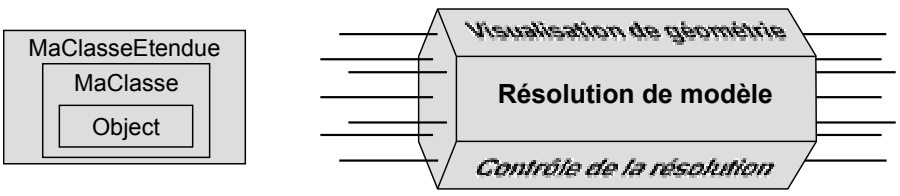
fois chargé, le mécanisme d'introspection doit fournir la liste des différents services rendus par le composant, ainsi qu'un moyen simple d'y accéder.

Actuellement, on utilise un type de composant par application, et on connaît par avance le ou les services rendus par ce composant. Par exemple, si on sait que le composant que l'on va utiliser est un composant de résolution de modèles, on connaît alors le moyen d'accéder aux services rendus par ce composant. Dans l'optique du schizomorphisme, on charge un composant (on parle ici de chargement générique), et ce composant va nous fournir le moyen de savoir quels services il rend (c'est le rôle du mécanisme d'introspection). Si il rend le service de résolution de modèles, alors on pourra utiliser ce composant en tant que composant de résolution de modèles. Mais parallèlement, il peut aussi rendre d'autres services (comme par exemple un service de visualisation du modèle contenu dans le composant), et ceux-ci sont accessibles au même titre que le service de résolution de modèles.

Une notion proche du schizomorphisme existe : il s'agit du polymorphisme (du grec *poly*, "plusieurs", et *morphè*, "forme"), qui est une des caractéristiques principales de la programmation orientée objet, intrinsèquement liée au mécanisme d'héritage. Par exemple, en Java, tous les objets existants héritent de la classe `Object`. Prenons donc l'exemple d'une classe `MaClasse`, héritant d'`Object`. Tous les objets de la classe `MaClasse` peuvent donc être vus aussi comme des objets de la classe `Object`. C'est le polymorphisme.

La différence entre polymorphisme et schizomorphisme est que le polymorphisme récupère toutes les propriétés et fonctionnalités de son héritage, et en ajoute de nouvelles (voir figure III.3). Les objets de la classe `MaClasse` sont des `Object`, mais possèdent des fonctionnalités supplémentaires, apportées par `MaClasse`. Au final, on peut voir un objet de plusieurs manières différentes suivant le niveau auquel on s'intéresse dans le graphe d'héritage de la classe auquel il appartient. Ces différents niveaux du graphe d'héritage sont comme les différentes vues possibles de cette classe.

Le schizomorphisme, quant à lui, est aussi caractérisé par cette notion de vues multiples d'une même entité. Cette différence est illustrée sur la figure III.3 :



Polymorphisme **Schizomorphisme**

Figure III.3 : Différence entre polymorphisme et schizomorphisme

Mais ces vues multiples ne proviennent pas d'un quelconque héritage, et c'est là qu'est la différence avec le polymorphisme. Le service de visualisation des modèles n'hérite en rien du service de calcul de ce modèle. L'un n'a pas besoin de l'autre pour fonctionner, alors que la classe `MaClasse` nécessite la connaissance de la classe `Object` pour être opérationnelle.
Il faudra donc que le standard de composant fournisse un mécanisme de support au schizomorphisme des composants, en donnant un moyen d'accès facile aux différents services que les composants peuvent contenir.

III.1.2.4. Evolutivité du standard

La motivation principale de la création d'un nouveau standard est de pouvoir gérer l'hétérogénéité des divers composants utilisés. Il faut donc que le standard puisse prendre en compte tous les types de composants utilisés actuellement, mais aussi les nouveaux types de composants qui ne manqueront pas de faire leur apparition au cours de futurs travaux, afin de répondre à un nouveau besoin. Il faut donc que le standard permette l'évolutivité des types de composants, tout en donnant une base stable permettant l'utilisation de ces composants.

III.1.2.5. Portabilité du standard

L'essence même d'un standard est de pouvoir s'appliquer partout, donc dans notre cas sur diverses architectures matérielles[1] et sur divers systèmes d'exploitation[2]. Il faut donc que notre standard puisse être utilisé sur les principales architectures matérielles et les principaux systèmes d'exploitations utilisés en conception. L'architecture matérielle prédominante est le PC, et les deux systèmes d'exploitation rencontrés sont Windows et Linux. De plus, nous travaillons essentiellement sous Windows. Notre standard doit donc être accessible au moins dans ces cas là, et plus largement si possible.

III.1.3. Spécification d'un standard de composants pour la conception

Afin de répondre aux différents besoins évoqués, nous avons développé un nouveau standard de composants comprenant un jeu d'interfaces permettant de définir les composants, une norme pour les fichiers contenant les composants, et un jeu de classes utilitaires fournissant divers services relatifs à la gestion des composants. Ce standard est appelé ICAr, pour **I**nterfaces for **C**omponent **Ar**chitecture. Nous allons maintenant spécifier ces différents aspects d'ICAr.

[1] Architecture matérielle : type d'ordinateur utilisé (PC, Macintosh, Sun Workstation, …)
[2] Système d'exploitation : ensemble cohérent de logiciels permettant d'utiliser un ordinateur. Voir glossaire.

III.1.3.1. Choix du langage de base du standard

Nous avons choisi comme langage de base d'ICAr le langage Java. Plusieurs raisons ont conduit à ce choix. D'une part, Java est un langage qui, grâce à son système de bytecode, est portable sur de nombreux systèmes, en particuliers ceux qui ont été évoqués lors des besoins relatifs aux composants. Cela nous assure donc la possibilité du portage du standard vers différentes plateformes. D'autre part, le langage Java, via le mécanisme de la JNI[1], permet d'utiliser du code écrit dans d'autres langages, ce qui permet de gérer une partie de l'hétérogénéité des composants. Cependant, cela se fait au détriment de la portabilité, ce que nous acceptons en raison du fait que nous travaillons majoritairement sous Windows. Troisièmement, Java inclut une interface d'utilisation des objets Corba, ce qui laisse la possibilité d'une future intégration de composants basés en Corba (ce qui sera particulièrement utile dans le cadre de calculs utilisant des systèmes de composants distribués). Enfin les logiciels ainsi que les différents composants développés dans l'équipe CDI du LEG ces dernières années sont basés eux aussi en Java, ce qui nous évite une somme de travail supplémentaire pour intégrer les composants existants au nouveau standard.

III.1.3.2. Jeu d'interfaces structurant le standard

ICAr est basé sur deux interfaces qui permettent de structurer et d'unifier les différents composants. Ces interfaces fournissent aussi le mécanisme d'introspection permettant de lister et d'accéder aux différents services des composants.

III.1.3.2.1. Interface Component

L'interface de base est l'interface définissant un composant pour le standard ICAr. Tous les composants satisfaisant au standard doivent implémenter cette interface `Component`, qui est représentée sur la figure III.4 :

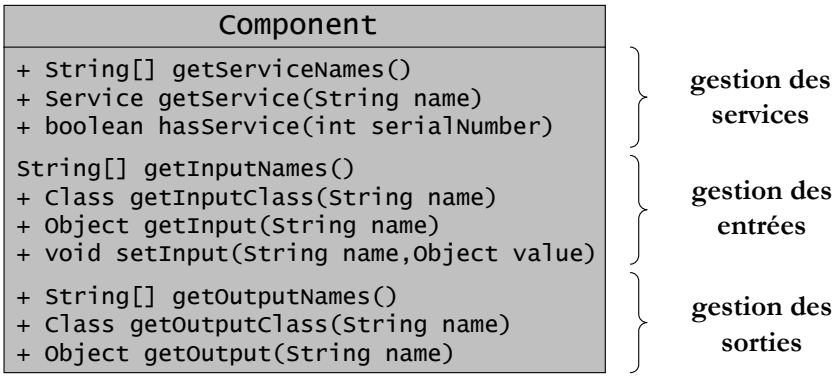


Figure III.4 : Représentation de l'interface `Component`

[1] JNI : Java Native Interface. Voir glossaire.

Les intérêts de cette interface sont multiples. Tout d'abord, ses méthodes constituent une partie du mécanisme d'introspection du composant, puisque la méthode `getServiceNames` donne accès à la liste des services rendus par le composant, la méthode `getService` donne accès à chaque service disponible, et la méthode `hasService` permet de savoir si le composant rend le service spécifié. La première partie du mécanisme d'introspection donnant un accès facile aux différents services du composant est donc assurée par l'interface `Component`.

Une deuxième partie du mécanisme d'introspection est assurée par les méthodes de gestion des entrées et sorties (méthodes `get<Input|Output>Names` et `get<Input|Output>Class`). Elles permettent d'accéder à la liste des entrées et sorties du composant ainsi qu'à la classe de chacune des ces entrées et sorties. Elle comprend aussi des méthodes relatives à la gestions des entrées et sorties, en lecture par la méthode `get<Input|Output>`, et pour la spécification des valeurs des entrées, la méthode `setInput`. On ne peut pas spécifier la valeur des sorties, celles-ci étant le résultat d'un service appliqué sur les entrées.

Mais ce n'est pas le seul intérêt de cette interface. Comme tous les composants doivent l'implémenter, elle constitue le point d'accès du composant au moment du chargement. Elle permet ainsi l'utilisation d'un mécanisme de chargement unifié s'appuyant sur cette interface. En effet, dans le cas du Cob par exemple, on chargeait un Cob en utilisant un mécanisme de chargement spécifique, adapté à la norme Cob [DEL]. Grâce à l'interface `Component`, on va pouvoir créer un chargeur dont la première action lors du chargement d'un composant sera de charger la classe implémentant cette interface. Elle nous donne donc un point d'accès fixe à tous les composants. Elle donne aussi la possibilité de mettre en place une norme pour le fichier contenant le composant.

III.1.3.2.2. Interface Service

L'interface `Component` constitue la base du standard ICAr. Elle donne accès aux différents services du composant, définis par l'interface `Service` représentée sur la figure III.5 :

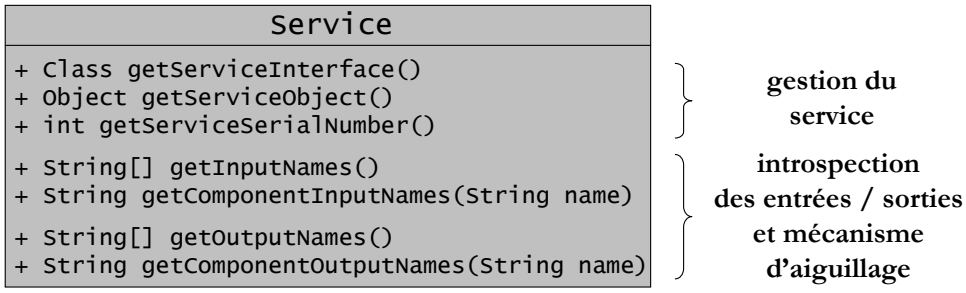


Figure III.5 : Représentation de l'interface `Service`

L'interface `Service` comprend deux séries de méthodes. La première série sert à connaître le type du service, défini par son interface, via la méthode `getServiceInterface`, à accéder à

l'objet rendant le service par la méthode `getServiceObject` et à connaître le numéro de série du service (méthode `getServiceSerialNumber`). Du point de vue du standard, on ne peut pas connaître à l'avance les interfaces des différents services rendus par le composant. Le composant doit donc être capable de fournir ces interfaces. Du point de vue de l'utilisation, on ne peut pas utiliser les services d'un composant sans le connaître : on ne peut pas utiliser un composant de calcul sans savoir ce qu'est un composant de calcul. La distribution des interfaces propres à chaque service (les interfaces métier) reste à la charge des développeurs. On peut ainsi assurer l'évolutivité des services intégrés dans le standard.

Les deux autres méthodes permettent la gestion des entrées et des sorties du composant relatives au service. Elles comprennent la deuxième partie du mécanisme d'introspection du composant, composé des méthodes `get<Input|Output>Names`, ainsi que le mécanisme d'aiguillage. La nécessité du mécanisme d'aiguillage peut être illustré grâce à la figure III.6 :

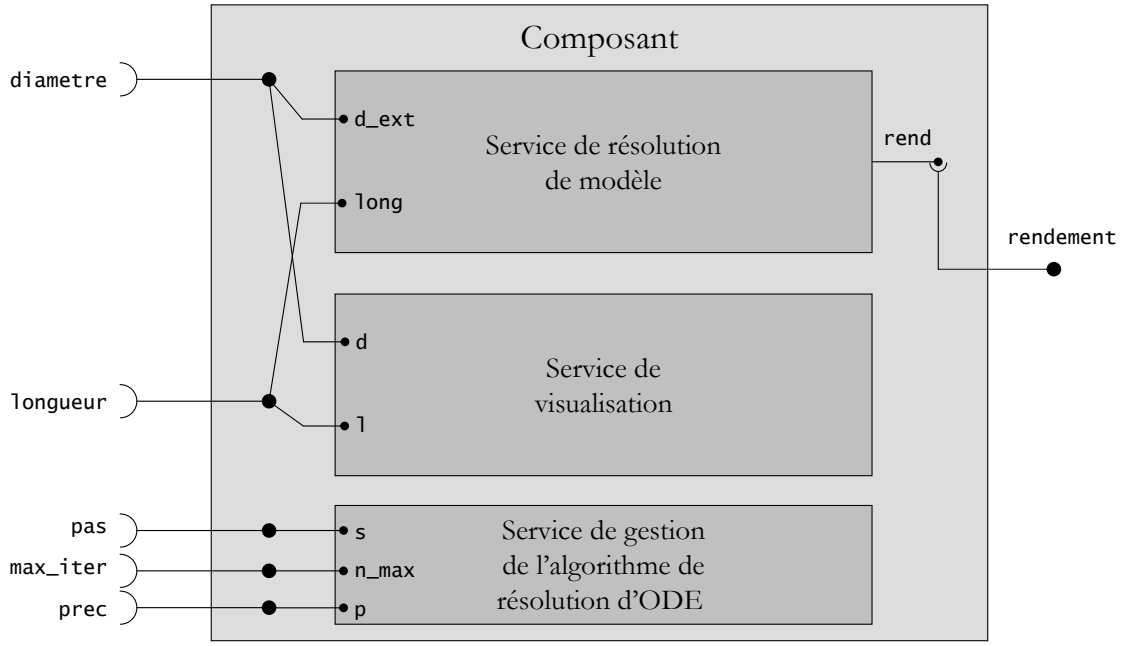


Figure III.6 : Principe du mécanisme d'aiguillage

On peut voir ici que les entrées propres à chaque service sont reliées aux entrées du composant, tout comme pour les sorties. D'ailleurs, on ne peut pas affecter une valeur pour une entrée d'un service, ou récupérer la valeur de la sortie d'un service. Les entrées et sorties sont gérées au niveau du composant. Le service d'aiguillage permet en fait de connaître la liste des entrées du composant utiles à chaque service, ainsi que la provenance de chaque sortie du composant. En effet, au niveau du composant, les entrées ne sont différentiables que par leur nom. Or il faut pouvoir les associer avec les services auxquels elles servent afin de pouvoir utiliser le composant correctement. Un service de gestion d'un algorithme numérique ne possède pas les mêmes entrées qu'un service de résolution de modèles, et celles-ci ne se gèrent donc pas de la même manière.

Au travers de ces deux interfaces, nous avons donc assuré le support d'un certain nombre de besoins dans le standard ICAr, comme un mécanisme d'introspection efficace et facile à utiliser, le support du schizomorphisme, l'évolutivité et la portabilité du standard. Il nous reste maintenant à spécifier une norme pour les fichiers contenant les composants, et le standard sera entièrement défini.

III.1.3.3. Norme des fichiers

Pour l'écriture des composants, nous avons choisi d'utiliser un format de fichier déjà fortement utilisé en Java et parfaitement adapté à nos besoins : le format JAr[1] (pour Java Archive). Ce format, dérivé du zip, est utilisé pour distribuer les applications Java. C'est un format de fichier utilisé pour distribuer du code exécutable, et donc exactement ce qu'il nous faut pour distribuer le code de nos composants. De plus, ce format inclut un système de manifeste[2] (fichier `manifest.mf`) permettant d'inclure les informations nécessaires au chargement du composant. Nous nous sommes donc basés sur le format JAr et nous avons spécifié quelques contraintes supplémentaires pour la structure du fichier :

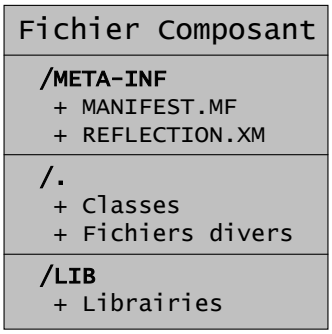


Figure III.7 : Représentation du fichier contenant le composant

Pour ce qui est des classes (donc du code de calcul), elles sont stockées exactement de la même manière que dans les JAr classiques. On peut ensuite trouver des librairies contenant du code natif. Ces librairies doivent être placées dans un répertoire spécial de l'archive, `/lib`. Elles nécessitent en effet un traitement spécifique au moment du chargement du composant. On peut aussi trouver des objets sérialisés[3], qui sont eux directement gérés par les mécanismes d'écriture et de lecture fournis avec ICAr. Enfin, on peut trouver des fichiers divers, contenant par exemple des données, ou encore des sous-composants.

Le manifeste de l'archive désigne pour sa part la classe principale du composant, c'est-à-dire la classe implémentant l'interface `Composant`, ainsi que d'autres informations concernant le

[1] JAr : http://java.sun.com/j2se/1.4.2/docs/guide/jar/
[2] Manifeste : le manifeste répertorie le contenu d'une archive. Voir glossaire.
[3] Sérialisation : stockage de l'état des objets, ainsi que la manière de recréer cet objet pour plus tard. Voir glossaire.

composant. Enfin, le fichier REFLECTION.XM contient des informations utiles pour connaître le composant sans avoir à le charger.

Nous avons donc spécifié totalement la norme ICAr. En tant que telle, elle définit comment est structuré et écrit un composant. Il reste à fournir les outils permettant l'écriture, la lecture et l'introspection directe des composants.

III.1.3.4. Outils d'écriture, de lecture et d'introspection directe

III.1.3.4.1. Outil d'écriture des composants

La création et l'écriture des composants dans le respect de la norme d'ICAr se fait par l'intermédiaire de la classe ComponentWriter. Cette classe se présente de la manière suivante :

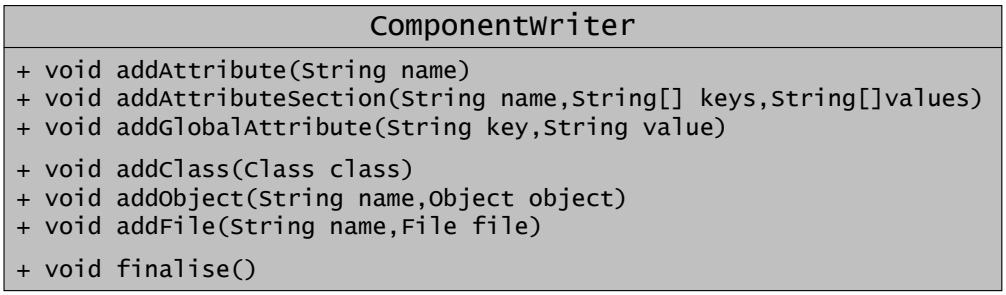


Figure III.8 : L'outil de création des composants

On peut voir que cette classe permet de gérer les informations du manifeste de l'archive, avec les méthodes addAttribute, addAttributeSection et addGlobalAttribute. Elle permet aussi d'incorporer des classes dans le composant (méthode addClass), des objets sérialisés (méthode addObject), et des fichiers divers, dont les librairies (méthode addFile).

III.1.3.4.2. Outil de lecture des composants

Le chargeur de composants se présente lui de la manière suivante :

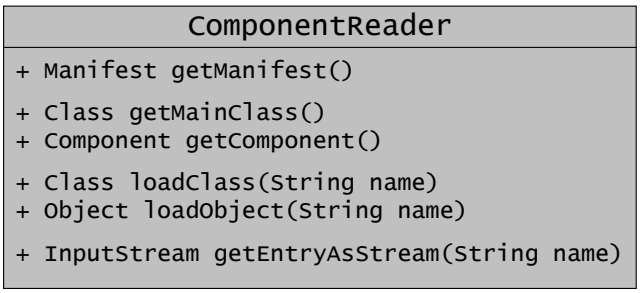


Figure III.9 : Le chargeur des composants

Il permet d'accéder au manifeste du composant (méthode getManifest), afin d'accéder aux informations concernant le composant, comme par exemple la classe principale du composant (celle qui implémente l'interface Component). Le chargeur permet aussi d'accéder directement à la classe principale (méthode getMainClass) ainsi qu'à une instance du composant (méthode getComponent), exécutant donc le chargement de la classe principale ainsi que son instanciation.

Enfin, le chargeur propose aussi des méthodes moins utiles pour le développeur standard, mais qui peuvent servir pour de la programmation plus avancée. La méthode `loadClass` permet ainsi d'accéder à n'importe quelle classe constituant le composant. De même, la méthode `loadObject` permet de charger un objet sérialisé. Enfin, la méthode `getEntryAsStream` permet d'accéder de manière directe à n'importe quel fichier contenu dans le composant sous forme d'un flux de données.

Par ailleurs, certains composants peuvent nécessiter un traitement particulier avant toute utilisation (préparation d'un espace de travail, copie ou chargement de certains fichiers, …). Un service spécial a donc été crée à cette intention :

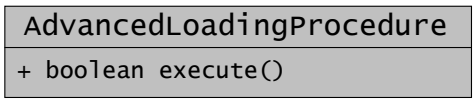


Figure III.10 : Le service de procédure de chargement avancé

Au moment du chargement du composant, le chargeur vérifie si le composant propose ce service, et dans le cas échéant l'exécute.

III.1.3.4.3. Outil d'instrospection directe

Enfin, ICAr propose aussi un outil d'introspection directe des composants, permettant d'accéder à différentes informations à propos d'un composant sans avoir à le charger. En fait, cet outil utilise les informations contenues dans le manifeste ainsi que dans le fichier `REFLEXION.XM`, qui sont accessibles directement sans avoir à charger le composant. Cet outil se présente de la manière suivante :

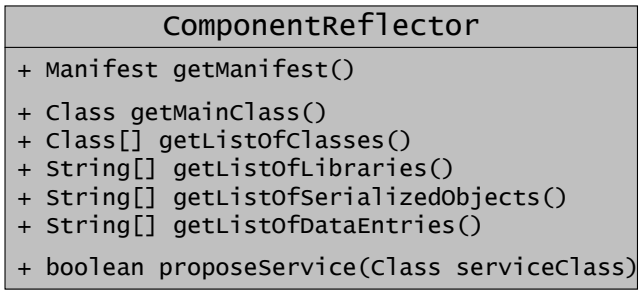


Figure III.11 : L'introspecteur de composants

Comme on peut le voir sur la figure III.11, le `ComponentReflector` permet d'accéder directement au manifeste du composant (méthode `getManifest`). Il propose aussi une série de méthodes permettant de connaître la constitution du composant :

- classe principale (méthode `getMainClass`)
- liste des classes (méthode `getListOfClasses`)
- liste des librairies (méthode `getListOfLibraries`)
- liste des objets sérialisés (méthode `getListOfSerializedObjects`)

- liste des fichiers divers (méthode `getListOfDataEntry`).

Enfin, la méthode `proposeService` permet de savoir si le composant propose un service particulier.

III.1.4. Spécifications des services dans le cadre d'ICAr

Nous venons de voir les spécifications d'un nouveau standard de composants Java. Ce standard n'est en fait qu'un squelette permettant de structurer les différents composants et de fournir une base de manipulation de ces composants. Il reste maintenant à intégrer les différents services rencontrés au cours de ces travaux.

III.1.4.1. Service de résolution des modèles de dimensionnement

Nous allons commencer par nous intéresser à l'intégration des composants qui sont à la base de nos travaux, les composants de résolution des modèles de dimensionnement. Plusieurs types de composants de résolution peuvent être envisagés. Nous allons commencer par voir l'intégration d'un composant de résolution des systèmes d'état, puis nous nous intéresserons à la spécification d'un composant de calcul des modèles de dimensionnement adapté à l'optimisation.

III.1.4.1.1. Service de résolution des systèmes d'état

A l'occasion des travaux sur la résolution des systèmes d'état vu au paragraphe II.1, un composant de résolution des systèmes d'état basé sur une approche symbolique avait été défini de manière autonome, c'est-à-dire hors d'ICAr. Ce composant, baptisé CoRe (pour Composant de Résolution), possède comme entrée les différents paramètres symboliques de la représentation d'état, et le temps. Ce composant est illustré sur la figure III.12 :

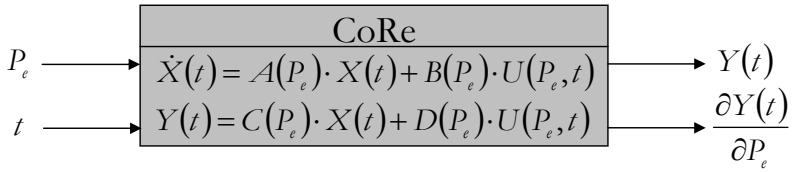


Figure III.12 : Le composant de résolution

L'interface associée à ce composant se présentait de la manière suivante :

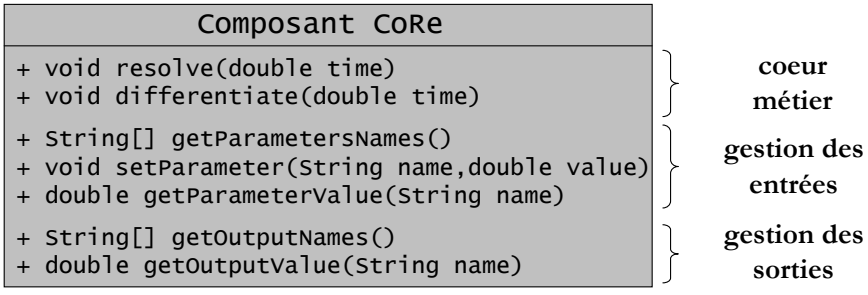


Figure III.13 : Interface associée au composant `CoRe`

Les sorties du composant sont les composantes du vecteur de sortie du système d'état.

On peut tout de suite noter la présence d'un mécanisme d'introspection concernant les entrées et les sorties du composant. En outre, on note la présence de deux méthodes spécifiques au service rendu par le composant, que nous appellerons méthodes métier : les méthodes `resolve` et `differentiate`. Une fois intégré dans ICAr, l'interface correspondante à ce service de résolution de systèmes d'état devient :

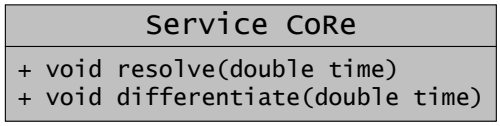


Figure III.14 : Interface associée au service `CoRe` dans le cadre d'ICAr

En effet, comme tout le mécanisme d'introspection et de gestion des entrées et des sorties est déjà inclus dans ICAr, la spécification d'une interface peut se consacrer à l'aspect métier du service rendu. Cette capitalisation du mécanisme d'introspection et de gestion des entrées et sorties constitue un des avantages procurés par l'utilisation d'ICAr.

III.1.4.1.2. Service de résolution de modèles pour l'optimisation

Nous nous proposons maintenant de spécifier les services de résolutions de modèles pour l'optimisation sous contraintes. Plusieurs types de services peuvent être envisagés dans ce cadre, suivant le type d'algorithme d'optimisation utilisé. En effet, certains algorithmes nécessitent la connaissance du gradient du modèle, et d'autres pas. Le service correspondant n'est donc pas forcément le même. Par ailleurs, plusieurs possibilités s'offrent à nous pour le format de transfert des dérivées des sorties du modèle.

Tout d'abord, pour les algorithmes stochastiques, ou certains algorithmes déterministes n'utilisant pas les gradients de la fonction, nous pouvons envisager le service suivant, n'utilisant que les entrées et les sorties du modèle :

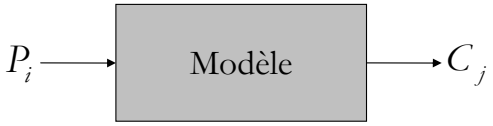


Figure III.15 : Service de calcul des modèles

Dans ce cas là, en effet, les algorithmes ne nécessitant pas la connaissance des dérivées, le service est tout simplement la résolution du modèle, c'est-à-dire le calcul des sorties connaissant les entrées. L'interface définissant ce service peut donc se présenter de la manière suivante :

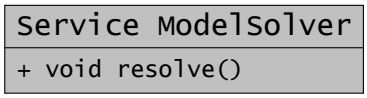


Figure III.16 : Interface associée au service de résolution des modèles

On voit donc que grâce à la capitalisation des aspects purement composant (chargement, introspection, gestions des entrées et des sorties), les interfaces définissant les services sont

réduites à leur plus simple expression, et ne présentent plus que l'aspect purement métier du service qu'elles rendent.

Dans le cadre de l'utilisation d'algorithmes d'optimisation nécessitant la connaissance des dérivées du modèle, il faut donc pouvoir fournir à la fois les critères de dimensionnement et leurs dérivées. Deux possibilités sont envisageables : l'approche jacobienne et l'approche différentielle. Dans l'approche jacobienne, on calcule les dérivées partielles des critères de dimensionnement par rapport aux entrées du modèle (donc le jacobien du modèle) :

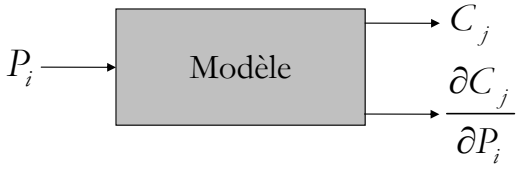


Figure III.17 : Service de calcul du modèle avec calcul du Jacobien

On peut donc associer l'interface suivante à ce service :

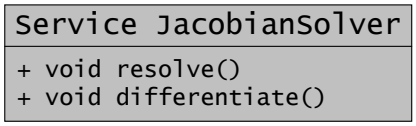


Figure III.18 : Interface associée

Les entrées et les sorties du service sont ici toutes de la classe `Double`, sauf une sortie, qui représente le jacobien du modèle, qui est de la classe `Jacobian`.

Dans l'approche différentielle, on ne calcule pas directement les dérivées partielles du modèle mais ses différentielles. Connaissant les valeurs des entrées du modèle, ainsi que les valeurs de leurs différentielles, on peut alors calculer les valeurs des sorties et de leurs différentielles. Un service de cette nature se présente de la manière suivante :

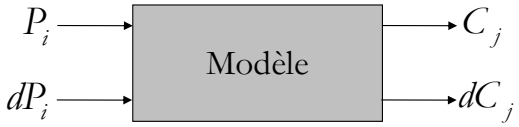


Figure III.19 : Service de calcul du modèle avec approche différentielle

L'interface associée est donc :

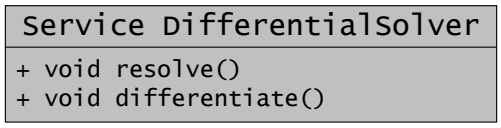


Figure III.20 : Interface associée

On voit tout de suite que cette interface n'est pas différente de celle qui utilise le calcul du jacobien. En effet, le service rendu est le même (c'est le calcul du modèle et de ses dérivées). Il est donc normal que l'interface soit la même. Seules les entrées et les sorties diffèrent d'un service à l'autre. En effet, ici les paramètres du modèle ainsi que les sorties comportent deux champs : leur valeur et leur différentielle.

La différence entre ces deux services se fait au niveau de leur utilisation. Les algorithmes d'optimisation utilisant les dérivées demandent le jacobien du modèle. C'est en effet à partir de la connaissance du jacobien que l'algorithme peut se diriger à travers l'espace des solutions. Si on utilise l'approche jacobienne, le service est donc parfaitement adapté à la procédure d'optimisation. Par contre, si on utilise l'approche différentielle, il faudra alors ajouter une couche de calcul entre l'algorithme et le modèle, puisqu'il faudra transformer les différentielles en dérivées partielles. Il sera donc moins pratique et surtout moins performant d'utiliser l'approche différentielle dans la seule optique de l'optimisation.

Par contre, la composition des modèles sera plus facile avec l'approche différentielle. En effet, dans cette approche, pour composer deux modèles, il suffit de relier les sorties de l'un aux entrées de l'autre et la composition est achevée, puisque les différentielles permettent la propagation directe du calcul des dérivées. Par contre la manière de mener le calcul n'est pas optimal. Dans l'approche jacobienne, il faut ajouter une couche de calcul entre les composants afin de pouvoir propager le calcul des dérivées. En contrepartie, le calcul sera mené de manière plus efficace.

Les deux types de services doivent donc être accessibles dans le cadre d'ICAr. Ces deux types de service peuvent d'ailleurs être disponibles dans le même composant, et on utilise une facette ou une autre du composant suivant le cadre dans lequel on se trouve. Les deux services peuvent d'ailleurs au final utiliser le même moteur de calcul à l'intérieur du composant, et adapter les résultats afin d'obtenir le formalisme adéquat. C'est un autre avantage de l'utilisation d'ICAr.

III.1.4.2. Service de visualisation et de post-processing

Un autre type de service à avoir été intégré dans ICAr est le service de visualisation et post-processing. Le principe de ce service est de fournir une vue paramétrée du dispositif que l'on modélise. Par exemple, dans le cadre du dimensionnement d'une machine, nous disposons du service de résolution du modèle de la machine. Au cours de l'optimisation, ou bien en fin d'optimisation, il peut être intéressant de disposer d'une représentation de la machine paramétrée, qui varie en fonction des paramètres d'entrée du modèle. Un tel service de visualisation des composants a été spécifié dans ICAr :

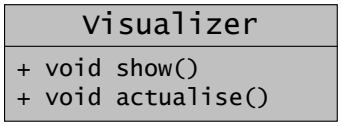


Figure III.21 : Le service de visualisation

Ce service comporte donc deux méthodes, une première pour l'initialisation de la représentation (la méthode `show`) et la deuxième pour l'actualisation de la représentation en fonction des paramètres d'entrée (la méthode `actualise`).

III.1.4.3. Service des gestion des algorithmes

Comme on l'a vu au Chapitre I, différents algorithmes numériques peuvent entrer en jeu lors de la résolution des modèles de dimensionnement, en particulier :

- Des algorithmes d'intégration numérique pour les systèmes différentiels (type Runge Kutta [PRE] par exemple)
- Des algorithmes de résolution de systèmes implicites (comme NS11[HSL] par exemple)

Ces algorithmes nécessitent un paramétrage (par exemple le pas de calcul, la précision, le nombre maximal d'itérations, …) avant leur utilisation. De même, ils renvoient souvent après leur exécution un ou plusieurs indicateurs, appelés *flags*, permettant de connaître leur état, comme par exemple le succès ou l'échec de la résolution.

On peut donc constater qu'un service de gestion des algorithmes peut parfaitement s'intégrer dans la vision des services telle qu'elle a été établie dans ICAr :

Figure III.22 : Représentation du service de gestion des algorithmes

L'interface associée à ce service est la suivante :

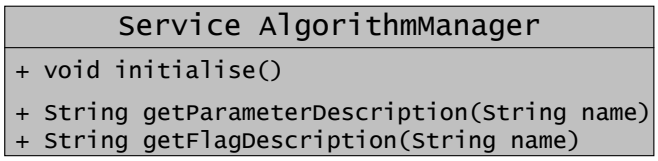


Figure III.23 : Interface du service `AlgorithmManager`

Cette interface permet de récupérer une description associée à chaque paramètre ou flag de l'algorithme (méthodes `get<Parameter|Flag>Description`). Cette description sert à informer l'utilisateur du composant sur la fonction ou la nature de chaque entrée ou sortie de l'algorithme.

Une initialisation de l'algorithme peut être nécessaire avant son exécution (comme par exemple la préparation d'un espace de travail sous forme d'un répertoire temporaire sur le disque dur). Cette initialisation est rendue possible via la méthode `initialise`.

III.1.4.4. Autres services

D'autres services ont été intégrés dans ICAr. Tous ces services ont pu bénéficier de l'architecture imposée par ICAr, sans pour autant être limités par cette architecture. Le composant COB[1], par exemple, existait déjà hors d'ICAr et n'a posé aucun problème lors de son intégration.

[1] COB : Composant de calcul de modèles. Voir glossaire.

III.2. Réalisation d'un environnement de gestion des composants

Afin de pouvoir gérer les composants et les différentes actions que l'on peut mener autour de ces composants (génération, projection, composition), un environnement de gestion des composants a été développé. Cet environnement a été baptisé CoreLab, pour **Co**mponents for **R**esearch and **E**ngineering **Lab**oratory. Nous allons exposer dans un premier temps les spécifications qui ont guidé la réalisation de l'environnement, puis nous nous intéresserons à son architecture et à son fonctionnement. Enfin, nous verrons plus particulièrement le fonctionnement des différents modules de CoreLab pour la gestion des composants.

III.2.1. Spécifications de l'environnement

Ce n'est pas à proprement parler un véritable cahier des charges qui nous a guidé durant la réalisation de CoreLab, mais plutôt quelques grands principes que nous nous sommes efforcés de respecter au maximum. Certains de ces principes sont communs à la réalisation de tout outil informatique, aussi nous ne nous attarderons pas sur ceux-là. Par contre, nous verrons plus en détails les caractéristiques spécifiques à CoreLab.

III.2.1.1. Fiabilité, maintenabilité, évolutivité, modularité et portabilité

Nous allons tout d'abord aborder les critères communs à toute réalisation d'un outil informatique. Ces principes sont généraux et, comme nous allons le voir, ne sont pas requis dans certains cas.

III.2.1.1.1. Fiabilité

Tout d'abord, un logiciel doit être fiable. En effet, si un logiciel présente trop d'erreurs (bugs) ou donne des résultats erronés, il devient très vite inutilisable. La fiabilité d'un logiciel ne s'acquiert vraiment que lors de l'utilisation du logiciel. Cela met en évidence les bugs qu'il présente et donne ainsi l'opportunité de les corriger.

III.2.1.1.2. Maintenabilité

Il faut ensuite qu'un logiciel soit maintenable [CHA], c'est-à-dire qu'il puisse être corrigé en cas de bug. Cette caractéristique est importante car la notion de maintenabilité inclue le fait que l'erreur doit pouvoir être corrigée par un développeur qui n'est pas forcément l'auteur original du logiciel. La maintenabilité impose donc une architecture cohérente et un minimum de documentation du code produit.

III.2.1.1.3. Evolutivité

Il faut aussi qu'un logiciel soit évolutif, c'est-à-dire qu'il faut pouvoir lui incorporer de nouvelles fonctionnalités sans pour autant toucher au code original. Pour cela, il faut que le logiciel soit ouvert un minimum, ce qui revient à laisser des portes d'entrées pour l'ajout des fonctionnalités. Cela est souvent fait au travers de jeux d'interfaces qui sont publiques et qui offrent des points d'accès au logiciel.

III.2.1.1.4. Modularité

Afin d'atteindre les exigences de maintenabilité et d'évolutivité, une bonne manière est de concevoir le logiciel de manière modulaire. Cette modularité permet en effet de découpler les principales fonctionnalités du logiciel, donc de l'architecture, et permet ainsi de trouver plus facilement les sources des bugs et rend la correction plus facile. De plus, les interactions entre les différents modules du logiciel se fait au travers des interfaces de ces modules, ce qui fournit la base de l'ouverture du moment que l'on rend ces interfaces publiques.

III.2.1.1.5. Portabilité

Enfin, une autre exigence qui peut être posée lors de la conception d'un logiciel est la portabilité[1]. C'est particulièrement le cas par exemple dans le domaine de la conception des jeux informatiques, puisque le même jeu est porté sur différentes consoles de jeux, et sur PC sous différents systèmes d'exploitation. Dans le cadre de la conception de CoreLab, la portabilité n'a pas été retenue en tant que critère prioritaire, l'environnement de travail majoritaire dans notre branche de la conception étant l'architecture PC sous Windows. Néanmoins, CoreLab étant programmé en Java (langage conçu pour être portable), cela nous assure ainsi une certaine portabilité . Dans la mesure où certaines parties du logiciel sont programmées en langage natif[2] (C, FORTRAN), une certaine quantité de travail sera nécessaire afin d'effectuer le portage de CoreLab vers d'autres plateformes.

III.2.1.2. Ouverture

Comme CoreLab est un logiciel destiné à la recherche et l'ingénierie, il faut qu'il soit ouvert. En effet, les besoins auxquels il doit répondre ne sont pas forcément connus à l'heure actuelle. Il faut donc prévoir des possibilités pour ajouter des fonctionnalités, et ceci à tous les niveaux architecturaux du logiciel (des grands modules aux petits algorithmes). Nous nous sommes donc efforcés au cours du développement logiciel à choisir l'architecture la plus à même de fournir une

[1] Portabilité : Possibilité d'être utilisé sous différents systèmes d'exploitation. Voir glossaire.
[2] Natif : Dépendance vis à vis de l'architecture matérielle et du système d'exploitation. Voir glossaire.

grande ouverture, et CoreLab est fourni avec un jeu d'interfaces permettant d'accéder au cœur logiciel. De même, CoreLab est basé sur le standard ICAr, qui lui aussi est public. Nous avons donc essayé de garder une ouverture du logiciel aussi grande que possible.

III.2.1.3. Principe d'utilisation de CoreLab

Afin de pouvoir gérer les composants, le concepteur doit être capable de mener plusieurs types d'actions sur les composants. Le premier type d'action est la création ou la génération des composants. Dans CoreLab, nous avons choisi une approche de génération par traitement d'un fichier de modèle. D'autres approches pour la génération des composants sont actuellement étudiées dans l'équipe CDI (notamment plus spécifiques, comme la génération à partir de réseaux de réluctances par exemple), mais notre approche reste plus généraliste. Le deuxième type d'action est la composition et le troisième type est la projection. Toutes ces actions doivent pouvoir être menées dans un même environnement de gestion des composants. Cette organisation est illustrée sur la figure III.24 :

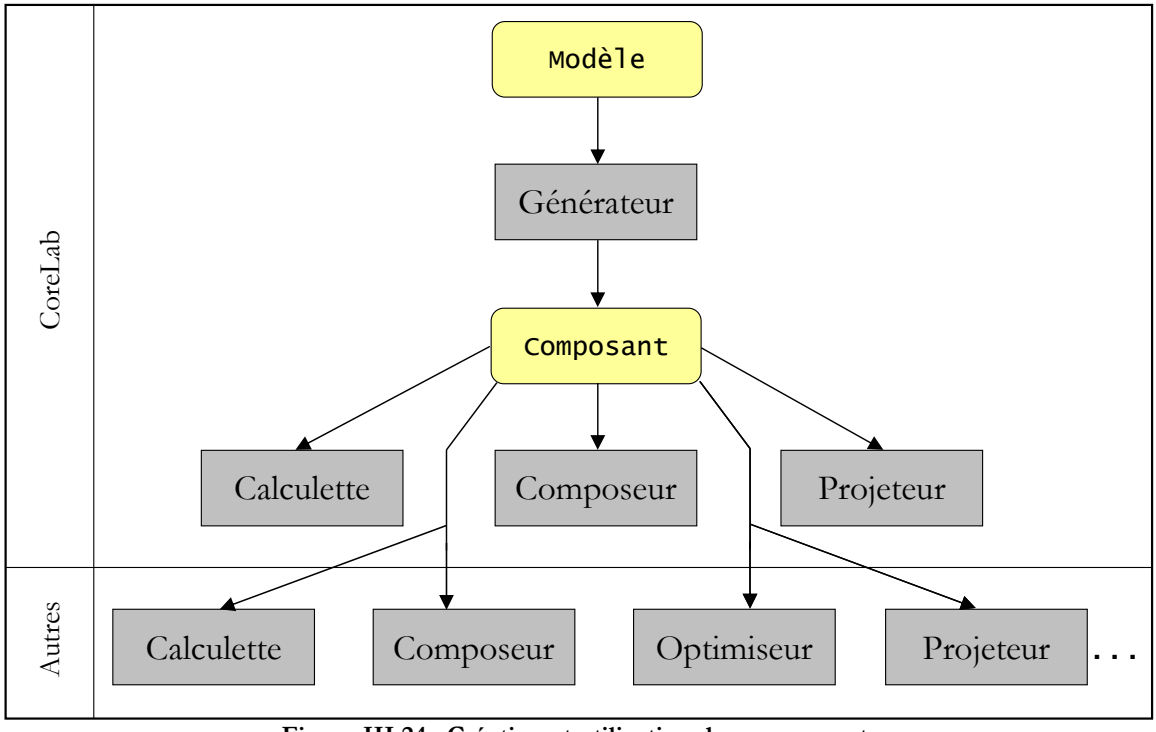


Figure III.24 : Création et utilisation des composants

Dans le cadre d'un composant de résolution de modèle pour le dimensionnement, le concepteur intervient dans un premier temps en décrivant un modèle traduisant sous forme mathématique le comportement d'un dispositif. A partir de ce modèle, le composant de résolution est généré. A ce stade, plusieurs utilisations du composant sont possibles. Une utilisation directe sous forme de calcul du modèle peut être effectuée, soit pour valider le fonctionnement du composant généré, soit pour étudier le comportement du dispositif avec un jeu de paramètres donné. Une procédure

d'optimisation utilisant ce composant peut être lancée. Ce composant peut aussi être utilisé en interaction avec d'autres composant pour construire un modèle plus complexe.

Bien sûr, les composants ne sont pas forcément générés par CoreLab. Tous les composants satisfaisant au standard ICAr sont utilisables dans CoreLab. De même, les utilisateurs (c'est-à-dire ici les outils informatiques) des composants sont pour la plupart externes à CoreLab, en particulier l'optimiseur, qui a été développé durant la thèse de David Magot [MAG], ou d'autres composeurs, comme par exemple celui développé par Benoît Delinchant [DEL].

III.2.2. Architecture générale

Afin d'offrir une évolutivité et une maintenabilité maximale, CoreLab a été conçu de manière modulaire. L'architecture générale de CoreLab peut se présenter de la manière suivante :

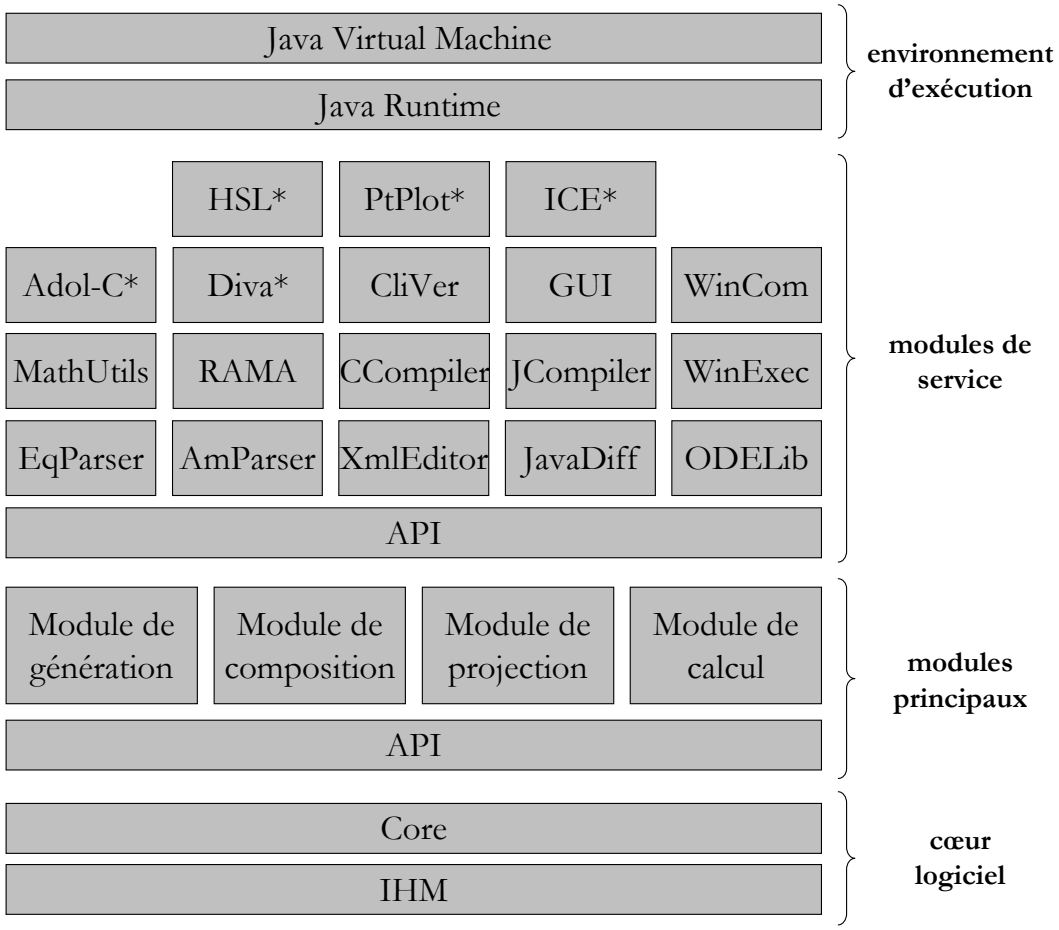


Figure III.25 : Architecture générale de CoreLab

Les modules de service marqués d'une étoile (*) n'ont pas été développés par nos soins mais ont été fournis par d'autres développeurs :

- Adol-C [GRI] est un package de différentiation automatique en C / C++ et a été fourni par Andreas Griewank et Olaf Vogel, qui travaillent sur la différentiation automatique à l'Université Technique de Dresde.

- HSL [HSL] (Harwell Subroutine Library) est une bibliothèques de fonctions numériques développée en Fortran et fournie par le Numerical Analysis Group, CSE, CCLRC.
- Diva et PtPlot sont deux packages faisant partie de Ptolemy [PTO], une suite logicielle pour la modélisation hétérogène développée par l'EECS de l'Université de Berkeley.
- ICE est un package open source contenant divers utilitaires pour la programmation.

Comme on peut le voir, deux types de modules existent :

- Les modules principaux, directement chargés des actions sur les composants, et directement pilotés par l'utilisateur au travers de l'IHM[1] de CoreLab :
  - Génération
  - Composition
  - Projection
  - Calcul
- Les modules de service, pilotées par les modules composants qui les utilisent lors des différentes actions menées sur les composants :
  - EqParser (parser d'expressions mathématiques)
  - RAMA (applicateur de règles mathématiques), cf Annexe I.
  - JavaDiff (dérivateur de code en Java)
  - ODELib (module de résolution des systèmes d'état)
  - XmlEditor (module d'édition de fichiers XML)
  - C Compiler (module de compilation de code C, qui utilise le compilateur Open Watcom [WAT])
  - …

Les modules sont définis par une interface qu'ils implémentent, et chaque module peut utiliser les fonctionnalités offertes par d'autres modules. Cette architecture modulaire est celle qui permet d'ajouter de nouvelles fonctionnalités le plus facilement et naturellement (ajout de nouveaux modules). De plus, la modularité offre une plus grande maintenabilité.

III.2.3. Module de génération des composants de calcul

III.2.3.1. Principe de fonctionnement

Le principe de fonctionnement du générateur de composants de calcul est de générer du code de calcul à partir d'un fichier contenant une description d'un modèle.

[1] IHM : Interface Homme-Machine. Voir glossaire.

Ce fonctionnement est illustré sur la figure III.26 :

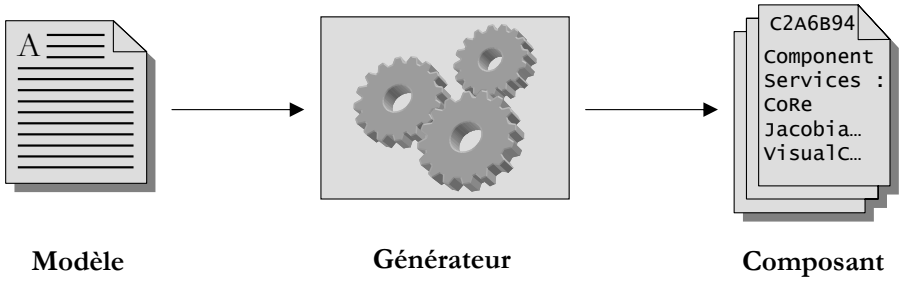


Figure III.26 : Principe de fonctionnement du module de génération

A priori, la description du modèle peut être de différents types :

- une description analytique sous forme d'équations
- un système d'état
- …

Au niveau du module de génération, nous ne pouvons pas prévoir ni d'ailleurs imposer un formalisme pour la description des modèles. Les formalismes seront imposés par chaque générateur, qui spécifie le type de description qu'il requiert. Il faut donc que le module de génération puisse prendre en compte différentes modélisations, et puisse générer différents types de composants (`CoRe`, `ModelSolver`, `JacobianSolver`, `DifferentialSolver`). Pour cela, le module de génération doit pouvoir proposer l'accès à différents générateurs.

L'architecture du module de génération est donc décomposée en deux parties principales, l'interface permettant d'éditer le modèle, et une bibliothèque de générateurs, comme le montre la figure III.27 :

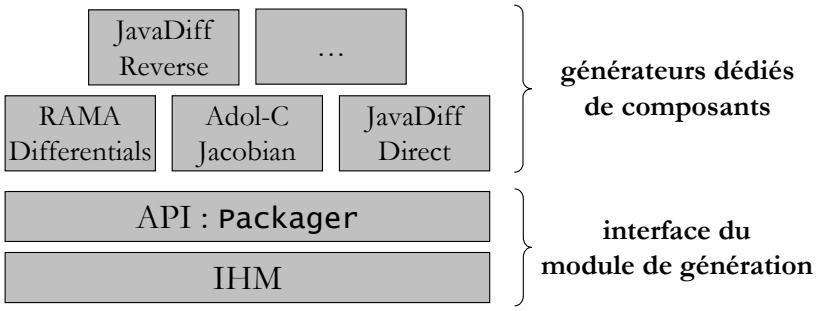


Figure III.27 : Architecture du module de génération

Il est bien sûr possible d'écrire son propre générateur de composants, ceci afin de conférer au module de génération un maximum d'ouverture. En effet, de nouvelles manières de générer des composants de calcul apparaissent tous les jours [CHE] [RAK]. Il faut donc pouvoir ajouter facilement de nouveaux générateurs à CoreLab. Ainsi, Edouard Dezille, au cours de son DEA, a développé plusieurs générateurs de composants pour CoreLab [DEZ].

III.2.3.2. Fonctionnement de nos générateurs dédiés

III.2.3.2.1. Organisation générale

Plusieurs générateurs ont été développés pour CoreLab. Le fonctionnement des générateurs que nous avons développés s'appuie sur le processus suivant :

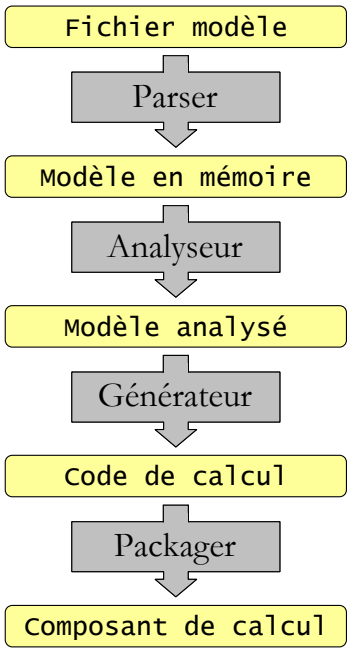


Figure III.28 : Processus de génération des générateurs de CoreLab

La première étape (le parsing) consiste à parcourir le fichier contenant le modèle afin d'en construire une image en mémoire. La structure grammaticale du fichier est validée (si il existe des erreurs dans le fichier la génération s'arrête). Une fois le modèle en mémoire, il est analysé en vue de la génération du code de calcul de sa résolution. Enfin, le code de calcul correspondant est généré puis compilé. Enfin, le composant proprement dit est créé.

III.2.3.2.2. Analyse du modèle

Comme nous l'avons vu, plusieurs types de modélisations peuvent être utilisés avec nos différents générateurs. Cependant, l'analyse du modèle s'organise de manière quasiment identique pour les différents générateurs.

L'analyse peut généralement se décomposer en quatre sous étapes principales, que nous allons maintenant détailler.

III.2.3.2.2.1. L'analyse et le tri des variables

Parmi les variables que le modèle comporte, certaines vont être des paramètres d'entrée du modèle, d'autres des sorties calculées lors de la résolution du modèle, et d'autres encore des

variables internes au modèle et n'apparaissant ni comme entrée ni comme sortie. L'étape d'analyse et de tri des variables permet de faire le distingo entre ces différents types de variables.

III.2.3.2.2.2. Le réordonnancement du modèle

Cette étape n'a d'intérêt que dans le cadre d'une modélisation analytique. Elle dépend de la grammaire imposée au concepteur lors de l'écriture du modèle. En effet, deux approches ont été proposée pour l'écriture des modèles au cours des travaux de l'équipe CDI : les modèles ordonnés ou non. Dans le cas d'un modèle ordonné, les expressions permettant de calculer les valeurs des variables sont écrites par le concepteur dans l'ordre exact dans lequel elles seront évaluées. L'écriture d'un modèle s'apparente alors fortement à l'écriture d'un code de calcul, avec les aspects purement informatiques (déclaration des variables, réservation de l'espace mémoire, …) en moins. Dans le cas d'un modèle non ordonné, les expressions des variables sont écrites dans un ordre quelconque. Potentiellement, une variable peut donc être utilisée avant d'avoir été calculée si le code de calcul est généré dans le même ordre que le modèle. Il faut alors réordonnancer les expressions afin de générer un code de calcul cohérent. L'utilisation d'une matrice d'occurrence permet de calculer l'ordre correct d'évaluation des expressions du modèle.

III.2.3.2.2.3. La vérification de la validité des fonctions utilisées dans le modèle

Deux types de fonctions peuvent être utilisés dans un modèle, les fonctions mathématiques de base (*cos*, *exp*, *arcsin*, …) et les fonctions définies par l'utilisateur dans le modèle. Il faut vérifier que toutes les fonctions utilisées dans le modèle sont soit des fonctions mathématiques de base, soit ont été définies dans le modèle.

III.2.3.2.2.4. La vérification de la calculabilité du modèle

Il faut en effet que le modèle soit calculable, c'est-à-dire par exemple éviter la présence de boucles dans le modèle.

III.2.3.2.3. Génération du composant de calcul

Une fois le modèle analysé, il reste à partir de ces informations à générer le code de calcul correspondant, le compiler, générer les codes ayant trait aux aspects purement composant (implémentation des interfaces, intégration au sein d'ICAr), et finalement packager les différents fichiers et ressources nécessaires au fonctionnement du composant et donc à la résolution du modèle. Pour illustrer ceci, considérons le cas d'un composant de résolution des modèles de dimensionnement avec calcul du Jacobien du modèle (service `JacobianSolver`). Nous avons par exemple utilisé Adol-C afin de calculer les dérivées partielles. Le composant va donc utiliser

un code de calcul écrit en C pour résoudre le modèle (repère 1), une partie en C++ pour calculer les dérivées (repère 2) et enfin une partie en Java pour toute la partie composant (repère 3). Le composant généré aura alors la structure suivante :

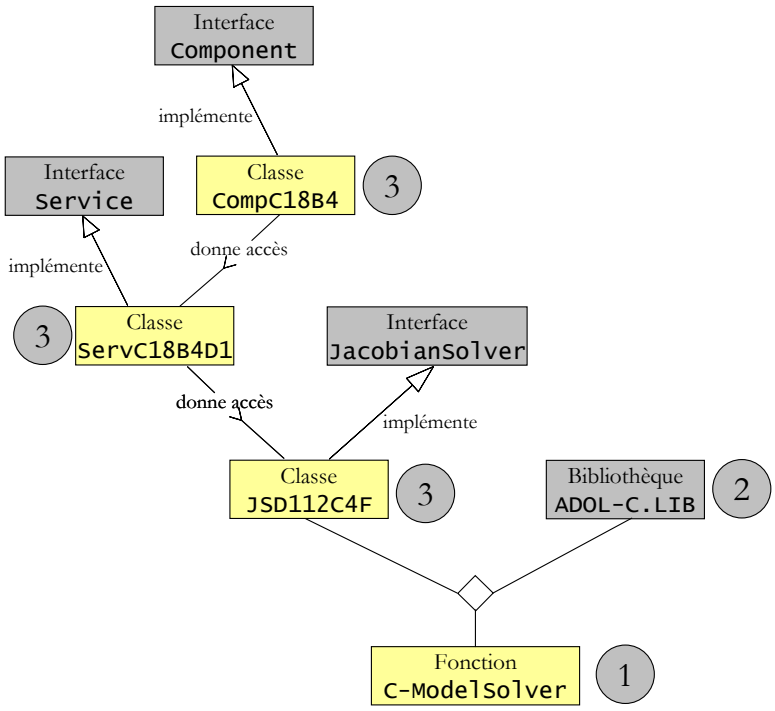


Figure III.29 : Structure du composant généré

On voit bien ici comment l'utilisation des interfaces `Component` et `Service` facilité l'accès aux différents services proposés par le composant.

Au final, l'utilisation de notre module de génération peut être synthétisée sur la figure III.30 :

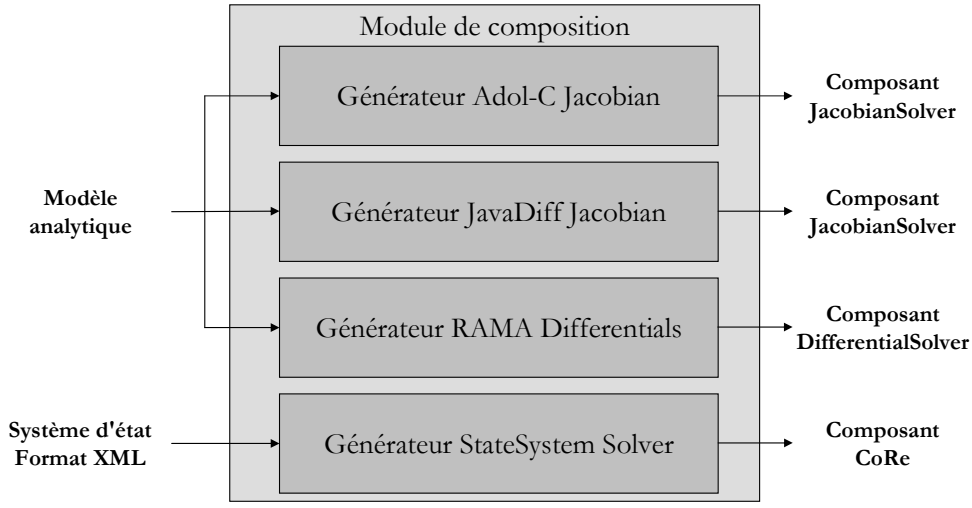


Figure III.30 : Synthèse de l'utilisation du module de génération

Les différents types de modèles sont traités par un jeu de générateurs dédiés pour tel ou tel type de composant de calcul.

III.2.4. Module de composition

III.2.4.1. Différentes compositions possibles

Une des principales caractéristiques des composants est le fait de pouvoir les relier entre eux. Il faut évidement fournir un support pour pouvoir effectuer cette composition. Le module de composition de CoreLab a été créé dans ce but. Tout comme pour la génération, différents types de compositions peuvent être envisagés. Par exemple, dans le cas de la composition des composants de résolution de modèles (afin par exemple d'obtenir des modèles développés par morceaux par des experts issus de différents domaines de la physique), plusieurs types de compositions doivent être pris en compte suivant le type de composant de résolution de modèle. Dans le cas des composants de type `ModelSolver`, la composition est simple, comme montré sur la figure III.31 :

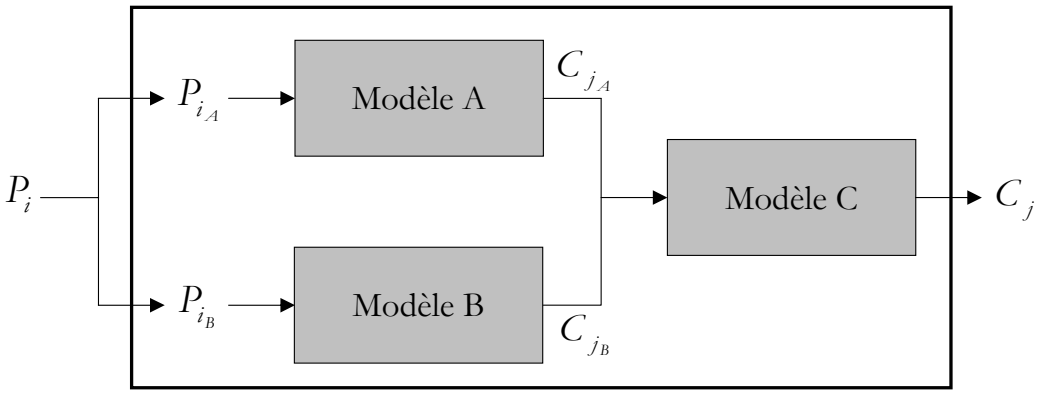


Figure III.31 : Composition de modèles simples au format `ModelSolver`

Dans le cas de composants de type `DifferentialSolver`, la composition reste simple, mais il faut propager le calcul des différentielles dans le modèle composé :

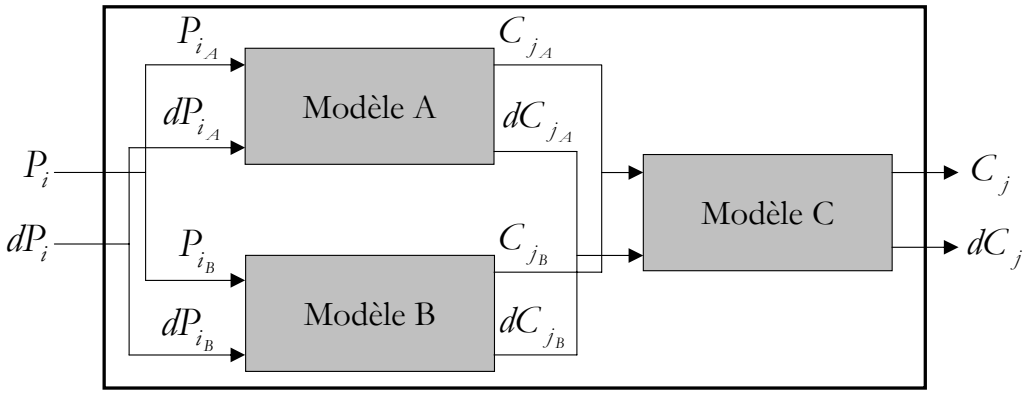


Figure III.32 : Composition de modèles au format `DifferentialSolver`

On voit bien, sur ces deux exemples, qu'on ne peut pas utiliser le même schéma de composition suivant les types de modèles utilisés. Il faut donc pouvoir changer de système de composition. Un système de composeurs modulaires identique au système de générateurs modulaires a été mis en place, afin de garder le maximum d'ouverture possible. La possibilité de rajouter dans CoreLab

un nouveau composeur a évidement été conservée. L'avantage de ce système, qui est encore plus parlant dans le cas de la composition que dans le cas de la génération, est que le développeur qui va écrire le composeur bénéficie de toute l'IHM du composeur, et n'a en fait qu'à se préoccuper de la composition proprement dite, à partir d'un modèle qui lui est fourni par CoreLab. Le but est de faire que les développeurs de modules n'aient à se focaliser que de l'aspect métier de leur développement et non pas des aspects d'interfaçage ou des à-côtés informatiques.

III.2.4.2. Composition hétérogène

III.2.4.2.1. Exemple de composition hétérogène

Afin de pouvoir gérer génériquement différents types de composants, le module de composition s'appuie sur l'architecture de composants ICAr. Cela présente l'avantage de pouvoir composer différents types de composants entre eux, et pas seulement des composants du même type :

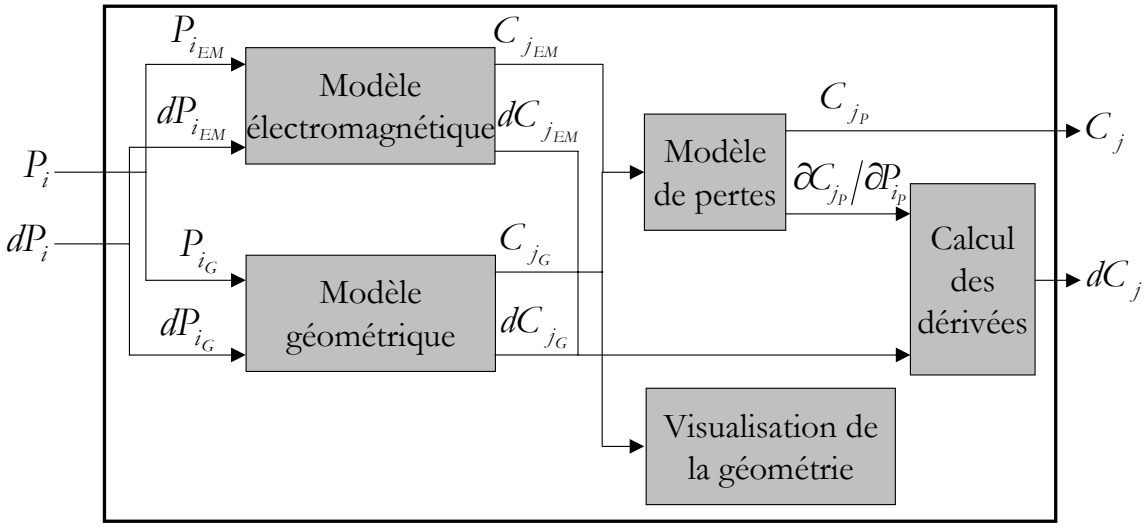


Figure III.33 : Composition hétérogène

La figure III.31 montre l'exemple de la création d'un modèle composé d'un transformateur en vue de son dimensionnement, avec différents types de composants.

Tout d'abord, le modèle électromagnétique ainsi que le modèle géométrique sont au format `DifferentialSolver`, c'est-à-dire basés sur les différentielles totales pour le calcul des dérivées. Le modèle de pertes, lui, est au format `JacobianSolver`, basé sur le jacobien. Pour passer d'un mode de propagation des dérivées à l'autre, il peut être nécessaire d'ajouter des filtres de transformation (comme ici le calcul des dérivées). Enfin, le composant de visualisation de géométrie permet en cours de calcul de visualiser le transformateur grâce aux valeurs des dimensions calculées par le modèle géométrique.

III.2.4.2.2. Intérêts de la composition hétérogène

Les intérêts de pouvoir composer des composants de type différent sont nombreux.

Tout d'abord, cela permet dans le cas des modèles de dimensionnement de créer des modèles complexes ayant trait à différents domaines de la physique à partir de modèles simples, explorant un aspect particulier du dispositif, et créé par un expert du domaine. Par exemple, dans le cas d'un transformateur, on pourrait trouver un modèle électromagnétique, un modèle géométrique, un modèle thermique, un modèle de vieillissement, un modèle de pertes et finalement un modèle économique, chacun de ces modèles étant écrit par un expert du domaine auquel il se rattache.

Tous ces différents modèles peuvent ensuite être assemblés afin d'être utilisés conjointement lors de l'optimisation du modèle global du transformateur. On voit ici l'intérêt en terme de facilité de modélisation (il est toujours plus facile de développer un modèle du dispositif par morceaux que globalement), ainsi qu'en terme de travaux collaboratifs. Le composant devient alors un bon support du travail de co-conception [RIB].

Enfin, on peut composer des composants contenant des services autres que des services de calcul pur, comme par exemple des composants de visualisation et post-processing. Cela permet de créer des composants évolués, proposant des services récupérés à partir des sous-composants.

Un composant ainsi généré à partir de la composition de plusieurs autres composants inclut ces sous-composants, afin d'être parfaitement autonome et de ne pas dépendre d'autres composants pour pouvoir fonctionner. Dans le cas du transformateur, le composant résultant de la composition aura la structure suivante :

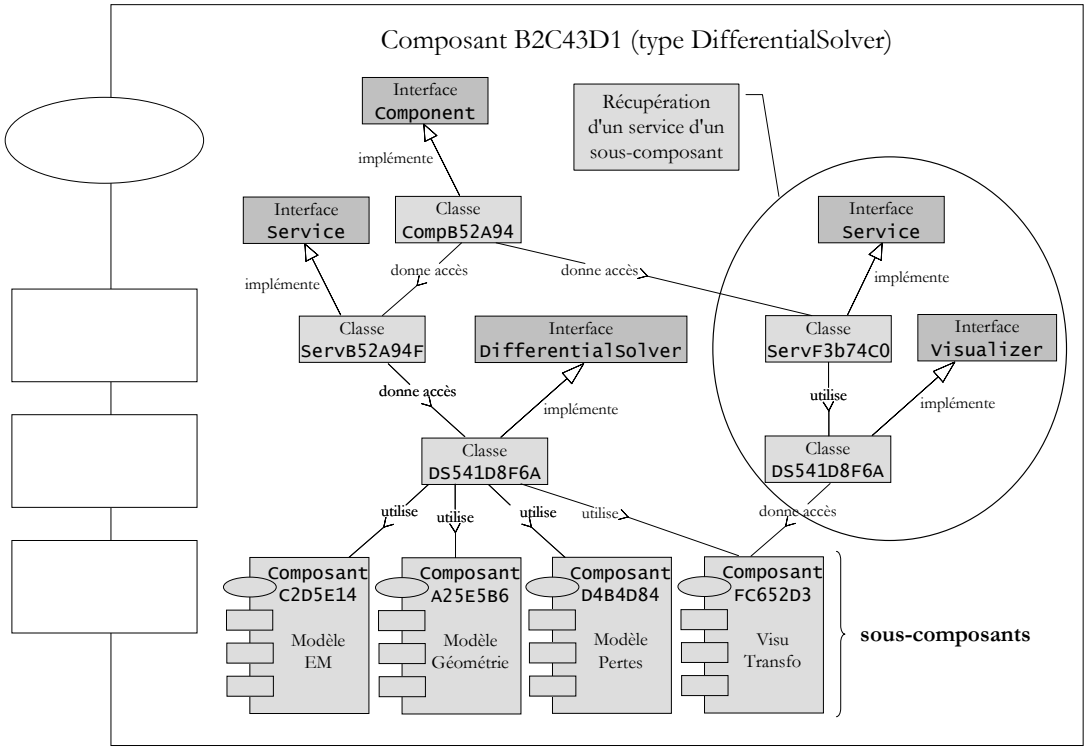


Figure III.34 : Structure d'un composant résultant d'une composition

On voit ici comment le composant généré inclut et utilise les sous-composants à partir desquels il est créé. On peut aussi voir comment le composant proposant le service de visualisation est intégré à la structure globale, tout en veillant à ce que le service de visualisation soit accessible directement à partir du composant généré. On peut ainsi récupérer des services contenus dans des sous-composants et les déclarer en tant que service du composant généré. On peut ainsi rendre accessible dans cet exemple les services de résolution des sous-modèles individuellement.

III.2.4.3. Architecture du module de composition

L'architecture du module de composition est donc la suivante :

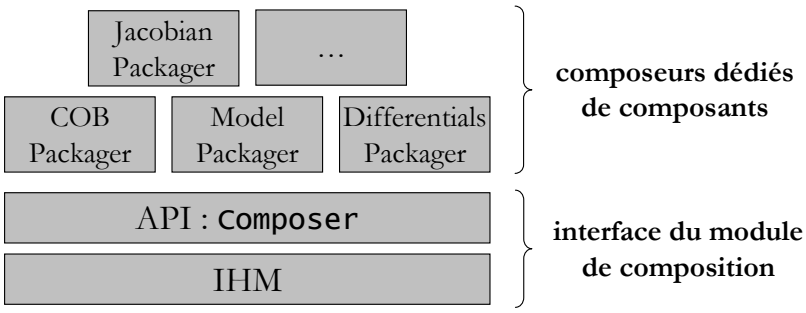


Figure III.35 : Architecture du module de composition

On voit ici la similarité d'architecture avec le module de génération des composants. En effet, cette architecture permet au développeur de composeurs (ou de générateurs) de se consacrer un maximum à l'aspect métier de son développement, en s'appuyant sur l'interface intégrée dans CoreLab, qui fournit les informations suffisantes au composeur pour générer la composition.

III.2.5. Module de projection

Une autre opération que l'on peut effectuer est la projection, c'est-à-dire un changement de type de service. Par exemple, on peut passer facilement d'un service du type `JacobianSolver` à un `DifferentialSolver`, comme le montre la figure III.36 :

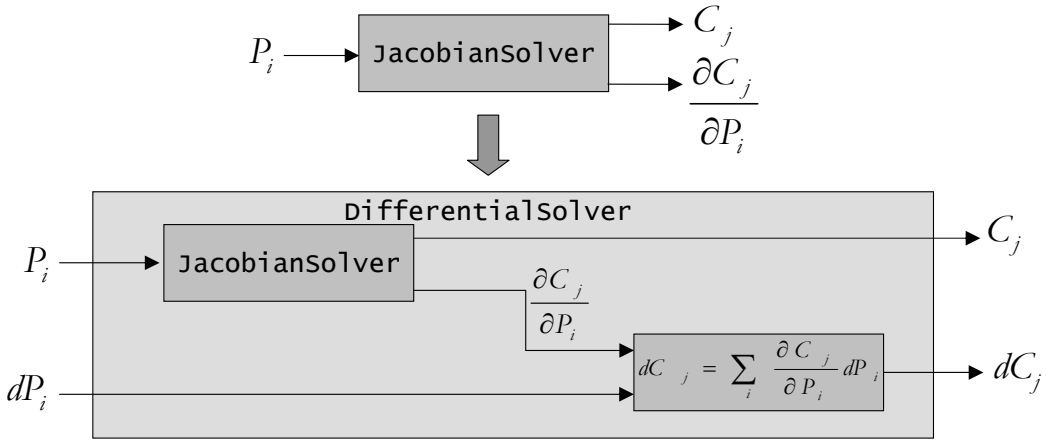


Figure III.36 : Projection de `JacobianSolver` vers `DifferentialSolver`

Ce changement de type de service est appelé projection d'un type vers un autre. Le problème spécifique est que le mécanisme de projection est étroitement lié avec les types de service de départ et d'arrivée. En effet, projeter un calcul de dérivées partielles vers un calcul de différentiel ne se fait pas de la même manière que la projection inverse par exemple.

Constatant la dépendance de la projection vis-à-vis des types de services de départ et d'arrivée, nous n'avons pas développé de module de projection à visée généraliste, comme nous l'avons fait pour les modules de génération et de composition. Nous avons donc développé différents modules de projection :

- Un module permettant de passer du type `JacobianSolver` au type `DifferentialSolver`.
- Un module permettant de passer du type `DifferentialSolver` au type `JacobianSolver`.
- Un module permettant d'intégrer des composants au format COB[1] au standard ICAr.

[1] COB : Computational Object. Voir glossaire.

CHAPITRE IV

IV.1. Présentation de CoreLab **107**
IV.2. Génération des composants de résolution **108**
IV.2.1. Génération d'un composant de résolution des équations différentielles 108
IV.2.1.1. Présentation de l'exemple 108
IV.2.1.2. Génération du composant 110
IV.2.1.3. Résultats obtenus 112
IV.2.2. Génération d'un composant de résolution de modèles analytiques 114
IV.2.2.1. Présentation de l'application 114
IV.2.2.2. Génération du composant 116
IV.3. Utilisation de la composition hétérogène **119**
IV.3.1. Présentation de l'application 119
IV.3.2. Composition du modèle 120
IV.4. Comparaison des systèmes de différentiation automatique **121**
IV.4.1. Comparaison des performances des différentiateurs 123
IV.4.2. Limites d'utilisation des différentiateurs 125
IV.4.3. Conséquences des limites d'utilisation de la dérivation de code 126
IV.5. Avantages et limites de la démarche proposée **126**

CHAPITRE IV

EXEMPLES D'UTILISATION

La démarche que nous proposons dans ce mémoire a été appliquée sur plusieurs cas tirés soit de publications, soit des travaux réalisés au sein du laboratoire. Dans ce chapitre, nous allons présenter et illustrer l'utilisation de CoreLab. Nous détaillerons ensuite l'utilisation de chaque module au travers de quatre exemples :

- un convertisseur statique connecté à un filtre RSIL (modélisation CEM) [BLA]
- un alternateur à griffes destiné à des applications automobiles [ALB]
- un transformateur triphasé contenant un modèle économique qui permet d'optimiser le transformateur sur son coût global capitalisé sur 30 ans [PO2] [FAN]
- un actionneur linéaire [CHI]

Nous profiterons de chaque exemple pour évaluer les avantages et les limites de l'utilisation de la démarche et des outils proposés. Toutes les applications ont été traitées sur un PC avec un processeur Intel Pentium IV 1.8 GHz, avec 512 MO de RAM.

IV.1. Présentation de CoreLab

D'une manière générale, CoreLab se présente de manière suivante :

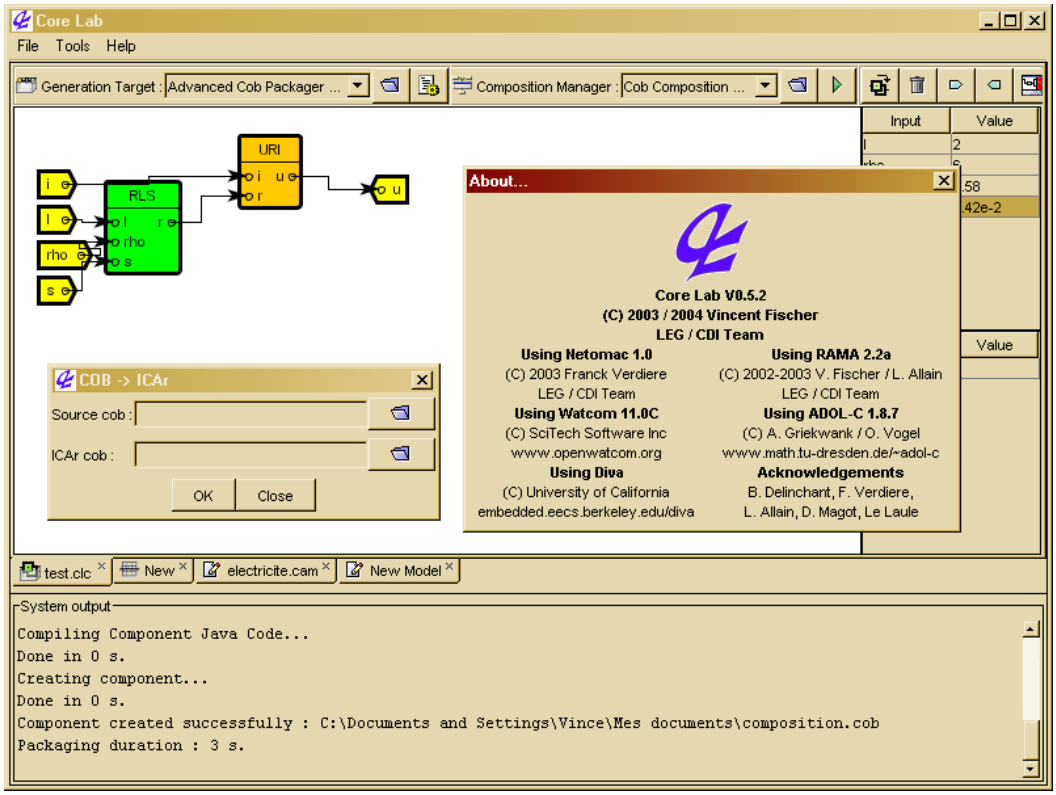


Figure IV.1 : Interface de CoreLab

On note la présence d'un espace de travail et d'une zone affichant les sorties de CoreLab et de ses modules. L'espace de travail se divise en onglets, chacun pouvant fonctionner indépendamment des autres, et étant l'interface d'un des modules principaux de CoreLab :

- Le module de génération des composants, qui permet de générer des composants à partir d'une description textuelle. Il donne accès aux différents générateurs, chacun étant spécifique à un type de description (modèle analytique, système d'état, …) et à une cible de génération (composant de calcul de type `JacobianSolver` ou `DifferentialSolver`, composant de résolution d'équations différentielles de type `CoRe`, …).
- Le module de composition, qui permet de créer des composants à partir d'un schéma de composition reliant d'autres composants. Comme le module de génération, il donne accès aux différents composeurs.
- Le module de projection, qui n'est pas encore terminé à l'heure actuelle. Il devra permettre de donner accès aux différents projeteurs permettant de projeter des composants d'une norme spécifique vers une autre.

IV.2. Génération des composants de résolution de modèles

Nous allons maintenant voir en détail le fonctionnement du module de génération des composants à partir d'une description textuelle. Nous allons illustrer ce fonctionnement au travers de deux exemples :

- La génération d'un composant de résolution du système d'équations différentielles modélisant le comportement d'un convertisseur statique connecté à son filtre RSIL
- La génération d'un composant de résolution du modèle analytique d'un alternateur à griffes

IV.2.1. Génération d'un composant de résolution d'équations différentielles

IV.2.1.1. Présentation de l'exemple

Afin d'illustrer la génération d'un composant de résolution d'équations différentielles, ou `CoRe`, nous allons nous baser sur l'exemple d'un convertisseur statique, entièrement modélisé par des éléments passifs, sauf pour ses semi-conducteurs qui sont modélisés par des sources de tension [BLA]. Le but de cette modélisation était d'obtenir les caractéristiques CEM de ce convertisseur. Les semi-conducteurs sont donc modélisés par des sources de tensions représentant les perturbations qu'ils injectent dans le circuit au moment de leur commutation.

Ce convertisseur est présenté sur la figure IV.2 :

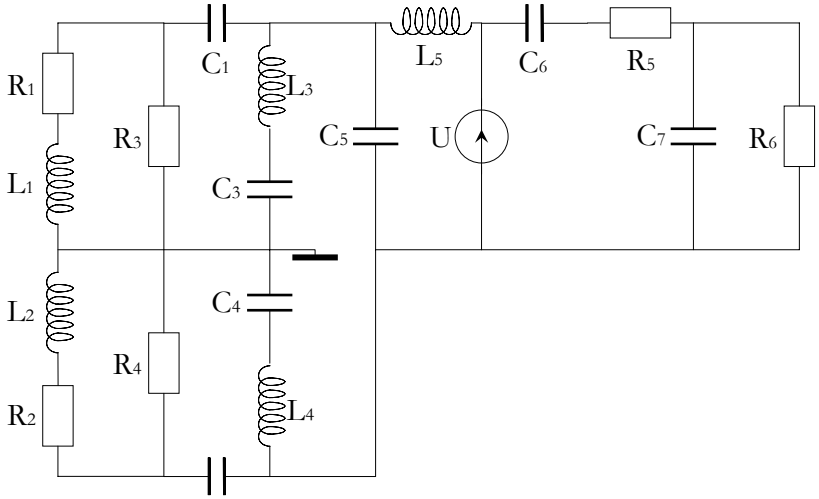


Figure IV.2 : Le convertisseur statique modélisé et son filtre RSIL

Les équations régissant ce circuit sont les suivantes :

$$\frac{\partial I_{L1}}{\partial t} = -\frac{R_3V_{C5} + R_3V_{C2} - R_3V_{C1} + R_1(R_3 + R_4)I_{L1} + R_3R_4(I_{L1} - I_{L2} + I_{L3} - I_{L4})}{L_1(R_3 + R_4)} \quad \text{(IV.1)}$$

$$\frac{\partial I_{L2}}{\partial t} = -\frac{R_4V_{C5} + R_4V_{C2} - R_4V_{C1} + (R_2(R_3 + R_4) + R_3R_4)I_{L2} + R_3R_4(-I_{L1} - I_{L3} + I_{L4})}{L_2(R_3 + R_4)} \quad \text{(IV.2)}$$

$$\frac{\partial I_{L3}}{\partial t} = -\frac{R_3V_{C5} + R_3V_{C2} - (R_3 + R_4)V_{C3} + R_4V_{C1} + R_3R_4(I_{L1} - I_{L2} + I_{L3} - I_{L4})}{L_3(R_3 + R_4)} \quad \text{(IV.3)}$$

$$\frac{\partial I_{L4}}{\partial t} = -\frac{R_4V_{C5} - R_3V_{C2} - (R_3 + R_4)V_{C4} - R_4V_{C1} - R_3R_4(I_{L1} - I_{L2} + I_{L3} - I_{L4})}{L_4(R_3 + R_4)} \quad \text{(IV.4)}$$

$$\frac{\partial I_{L5}}{\partial t} = \frac{U - V_{C5}}{L_5} \quad \text{(IV.5)}$$

$$\frac{\partial V_{C1}}{\partial t} = \frac{V_{C5} + V_{C2} - V_{C1} + R_4(I_{L2} - I_{L3} + I_{L4}) - R_3I_{L1}}{C_1(R_3 + R_4)} \quad \text{(IV.6)}$$

$$\frac{\partial V_{C2}}{\partial t} = -\frac{V_{C5} + V_{C2} - V_{C1} + R_4(I_{L2} + I_{L3} - I_{L4}) - R_3I_{L1}}{C_2(R_3 + R_4)} \quad \text{(IV.7)}$$

$$\frac{\partial V_{C3}}{\partial t} = -\frac{I_{L3}}{C_3} \quad \text{(IV.8)}$$

$$\frac{\partial V_{C4}}{\partial t} = -\frac{I_{L4}}{C_4} \quad \text{(IV.9)}$$

$$\frac{\partial V_{C5}}{\partial t} = -\frac{V_{C5} + V_{C2} - V_{C1} - R_4(I_{L2} + I_{L4}) - (R_3 + R_4)I_{L5} - R_4(I_{L1} + I_{L3})}{C_5(R_3 + R_4)} \quad \text{(IV.10)}$$

$$\frac{\partial V_{C6}}{\partial t} = -\frac{V_{C6} - V_{C7} + U}{R_5C_6} \quad \text{(IV.11)}$$

$$\frac{\partial V_{C7}}{\partial t} = \frac{-V_{C7}(R_5 + R_6) + R_6V_{C6} + R_6U}{R_5R_6C_7} \quad \text{(IV.12)}$$

A partir de ces équations, nous pouvons donc en déduire le système d'état :

$$A=\begin{bmatrix}
-\frac{R_1(R_3+R_4)+R_3R_4}{L_1(R_3+R_4)} & \frac{R_3R_4}{L_1(R_3+R_4)} & -\frac{R_3R_4}{L_1(R_3+R_4)} & \frac{R_3R_4}{L_1(R_3+R_4)} & 0 & \frac{R_3}{L_1(R_3+R_4)} & \frac{-R_3}{L_1(R_3+R_4)} & 0 & 0 & \frac{-R_3}{L_1(R_3+R_4)} & 0 & 0\\
\frac{R_3R_4}{L_2(R_3+R_4)} & -\frac{R_2(R_3+R_4)+R_3R_4}{L_2(R_3+R_4)} & \frac{R_3R_4}{L_2(R_3+R_4)} & \frac{-R_3R_4}{L_2(R_3+R_4)} & 0 & \frac{R_4}{L_2(R_3+R_4)} & \frac{-R_4}{L_2(R_3+R_4)} & 0 & 0 & \frac{-R_4}{L_2(R_3+R_4)} & 0 & 0\\
\frac{-R_3R_4}{L_3(R_3+R_4)} & \frac{R_3R_4}{L_3(R_3+R_4)} & \frac{-R_3R_4}{L_3(R_3+R_4)} & \frac{R_3R_4}{L_3(R_3+R_4)} & 0 & \frac{-R_4}{L_3(R_3+R_4)} & \frac{-R_3}{L_3(R_3+R_4)} & \frac{-R_3-R_4}{L_3(R_3+R_4)} & 0 & \frac{-R_3}{L_3(R_3+R_4)} & 0 & 0\\
\frac{R_3R_4}{L_4(R_3+R_4)} & \frac{-R_3R_4}{L_4(R_3+R_4)} & \frac{R_3R_4}{L_4(R_3+R_4)} & \frac{-R_3R_4}{L_4(R_3+R_4)} & 0 & \frac{R_4}{L_4(R_3+R_4)} & \frac{R_3}{L_4(R_3+R_4)} & 0 & \frac{-R_3-R_4}{L_4(R_3+R_4)} & \frac{-R_4}{L_4(R_3+R_4)} & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{L_5} & 0 & 0\\
\frac{-R_3}{C_1(R_3+R_4)} & \frac{-R_4}{C_1(R_3+R_4)} & \frac{R_4}{C_1(R_3+R_4)} & \frac{-R_4}{C_1(R_3+R_4)} & 0 & \frac{-1}{C_1(R_3+R_4)} & \frac{1}{C_1(R_3+R_4)} & 0 & 0 & \frac{1}{C_1(R_3+R_4)} & 0 & 0\\
\frac{R_3}{C_2(R_3+R_4)} & \frac{R_4}{C_2(R_3+R_4)} & \frac{R_3}{C_2(R_3+R_4)} & \frac{-R_3}{C_2(R_3+R_4)} & 0 & \frac{1}{C_2(R_3+R_4)} & \frac{-1}{C_2(R_3+R_4)} & 0 & 0 & \frac{-1}{C_2(R_3+R_4)} & 0 & 0\\
0 & 0 & \frac{-1}{C_3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & \frac{-1}{C_4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
\frac{R_3}{C_5(R_3+R_4)} & \frac{R_4}{C_5(R_3+R_4)} & \frac{R_3}{C_5(R_3+R_4)} & \frac{R_4}{C_5(R_3+R_4)} & \frac{R_3+R_4}{C_5(R_3+R_4)} & \frac{1}{C_5(R_3+R_4)} & \frac{-1}{C_5(R_3+R_4)} & 0 & 0 & \frac{-1}{C_5(R_3+R_4)} & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{C_6R_5} & \frac{1}{C_6R_5}\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{C_7R_5} & -\frac{R_5+R_6}{C_7R_5R_6}
\end{bmatrix} \qquad \text{(IV.13)}$$

$$B=\begin{bmatrix}
0\\0\\0\\0\\\frac{1}{L_5}\\0\\0\\0\\0\\0\\\frac{-1}{R_5C_6}\\\frac{1}{R_5C_7}
\end{bmatrix} \qquad \text{(IV.14)}$$

IV.2.1.2. Génération du composant

Dans CoreLab, cette représentation d'état peut être obtenue automatiquement à partir de la description du circuit au format NetList de PSpice [PSP]. On peut donc dessiner son circuit sous PSpice, puis générer cette représentation automatiquement. Le fichier XML contenant cette description du système d'état est construit de la manière suivante :

```
<CoreInfo>
  <CorIO>
      <CorIn name="r_1" default="5" />
      <CorIn name="r_2" default="5" />
      <CorIn name="r_3" default="50" />
      <CorIn name="r_4" default="50" />
      <CorIn name="r_5" default="1" />
      <CorIn name="r_6" default="400" />
      … … …
      <CorOut index="0" name="I_L1" />
      <CorOut index="1" name="I_L2" />
      <CorOut index="2" name="I_L3" />
      … … …
  </CorIO>
  <StateSystem states="12" sources="1" outputs="12">
      <Sources>
       <Source index="0" name="u_v1">
          <Pol d0="uc" />
          <Sin u="us" omega="om" phi="ph" />
       </Source>
```

```
</Sources>
… … …
<Matrix name="A">
 <Line index="0">
     <Element index="0" value="-(r_1*(r_4+r_3)+r_3*r_4)/l_1/(r_4+r_3…
     <Element index="1" value="r_3*r_4/l_1/(r_4+r_3)" />
     <Element index="2" value="-r_3*r_4/l_1/(r_4+r_3)" />
     <Element index="3" value="r_3*r_4/l_1/(r_4+r_3)" />
     <Element index="4" value="0" />
     <Element index="5" value="r_3/l_1/(r_4+r_3)" />
     <Element index="6" value="-r_3/l_1/(r_4+r_3)" />
     <Element index="7" value="0" />
     <Element index="8" value="0" />
     <Element index="9" value="-r_3/l_1/(r_4+r_3)" />
     <Element index="10" value="0" />
     <Element index="11" value="0" />
 </Line>
 … … …
```

Le format XML de représentation des systèmes d'état ainsi que le modèle complet de cette application sont présentés en Annexe III. Cette représentation peut paraître complexe à première vue, mais elle est obtenue automatiquement. De plus, une interface graphique spéciale a été développée pour permettre une édition du fichier XML plus facile qu'en mode texte. A partir de cette représentation, le composant est généré dans CoreLab :

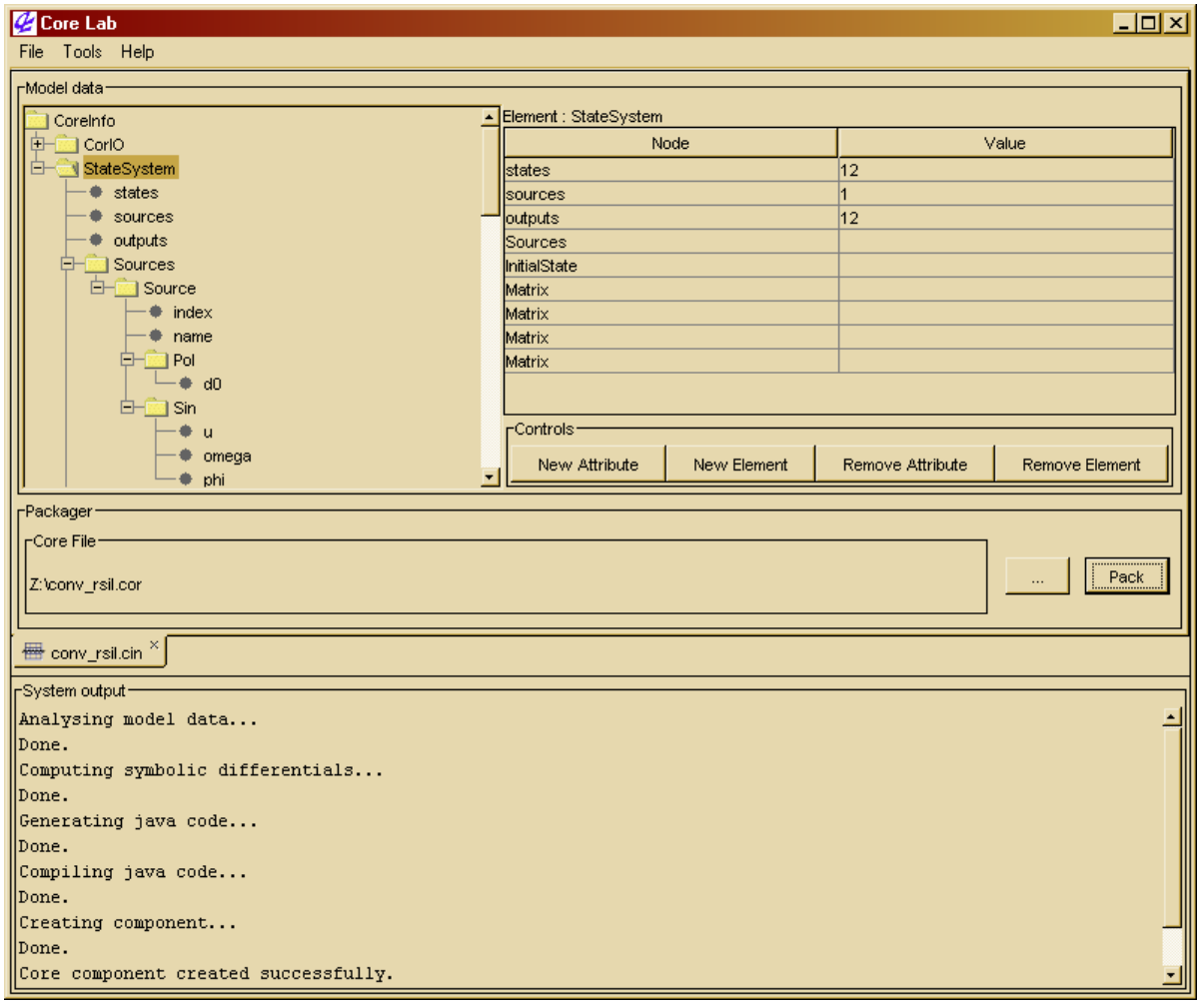


Figure IV.3 : Génération d'un composant de résolution d'équations différentielles

Une fois le composant créé, il ne reste plus qu'à l'utiliser. On peut par exemple le composer avec d'autres composants afin d'obtenir un composant de résolution de modèles en vue de l'optimisation.

On peut aussi vouloir exploiter directement la résolution du système différentiel. Pour cela, une calculette a été développée dans CoreLab, permettant d'exploiter la simulation. Cette calculette se présente de la manière suivante :

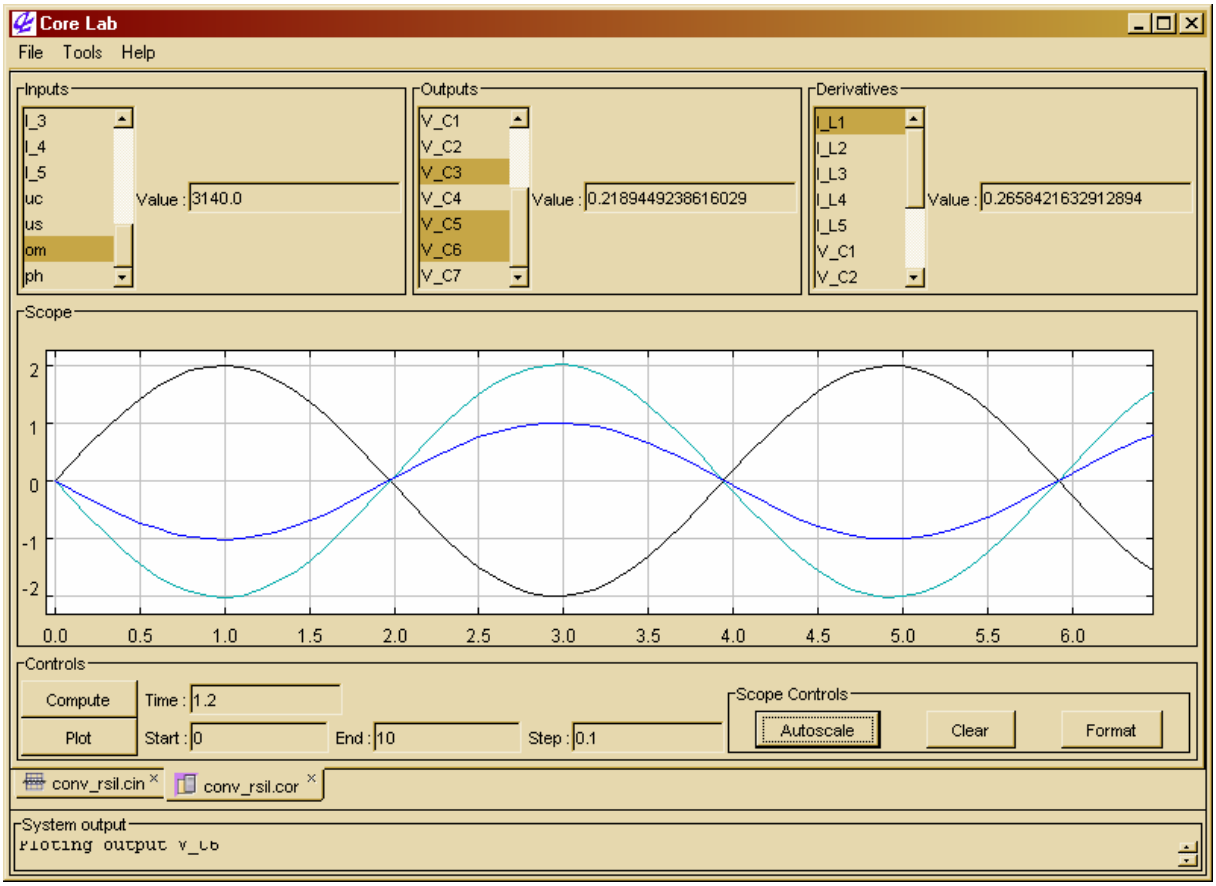


Figure IV.4 : Interface de la calculette pour composants de simulation de CoreLab

Cette calculette permet d'affecter des valeurs aux différents paramètres d'entrée du modèle (boite de dialogue `Inputs`), de lancer le calcul à l'instant désiré (bouton `Compute`), et d'obtenir la valeur des sorties à cet instant (boite de dialogue `Outputs`), ainsi que la valeur de leurs différentielles (boite de dialogue `Derivatives`). Elle permet également de lancer une simulation entre deux instants donnés pour un pas fixé (bouton `Plot`), et de tracer les évolutions des valeurs sélectionnées dans la boite de dialogue Outputs. On peut donc exploiter directement la résolution du système d'équations différentielles contenue dans le composant.

IV.2.1.3. Résultats obtenus

Le circuit présenté a été modélisé sous Simplorer [SIM], un logiciel de simulation numérique des circuits électriques.

Les résultats en simulation obtenus avec cette approche ont étés comparés avec la simulation sous Simplorer :

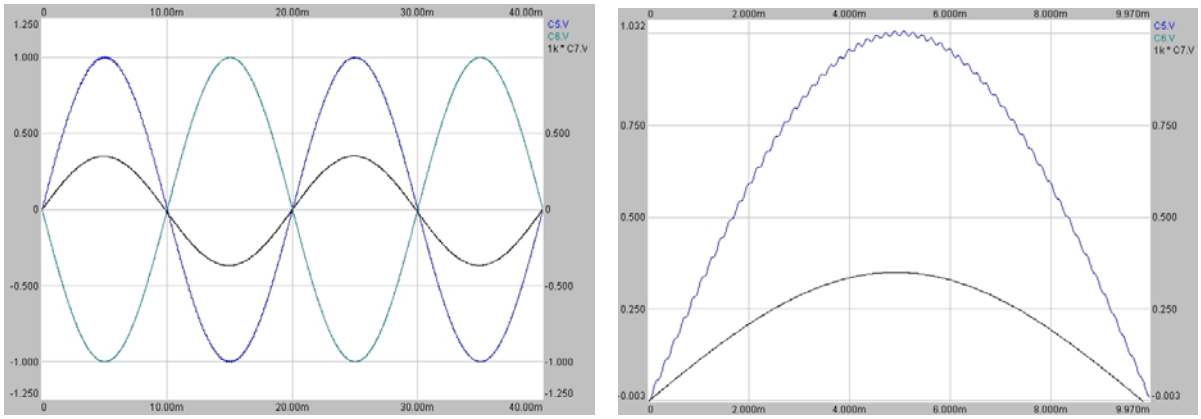


Figure IV.5 : Résultats de la simulation sous Simplorer

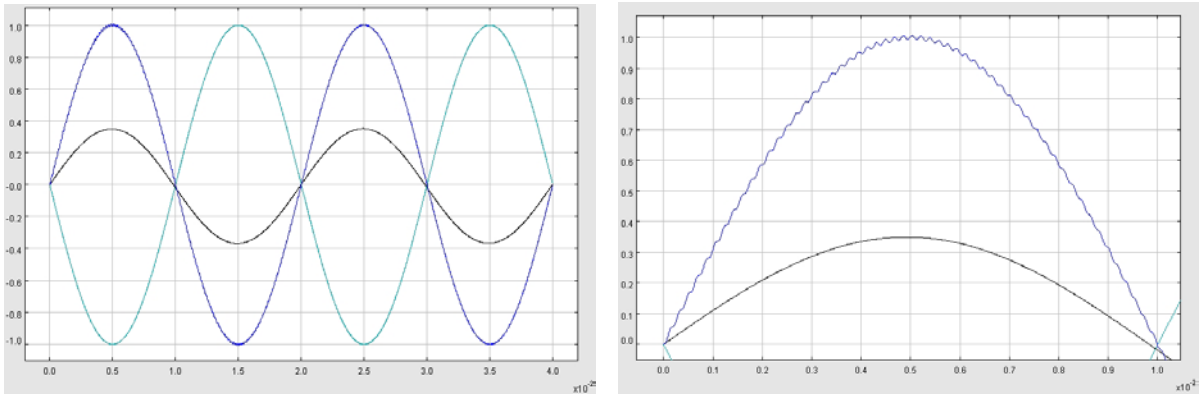


Figure IV.6 : Résultats de la simulation avec CoreLab

On voit donc ici que les résultats des deux simulations sont exactement identiques aux erreurs d'arrondi près (résultats identiques à 10^{-12} près). Les temps et la mémoire occupée pour obtenir la simulation totale (c'est-à-dire les courbes de gauche, contenant 10^6 points calculés) ainsi qu'uniquement le dernier point de la simulation (c'est-à-dire le point pour $t = 40ms$) sont présentés dans la table IV.1 :

	Simplorer Simulation Complète	CoreLab Simulation Complète	Simplorer Simulation Dernier Point	CoreLab Simulation Dernier Point
Temps (s)	3.7	4.1	3.7	0.002
Mémoire (MO)	25.5	12.3	25.5	1.1

Table IV.1 : Comparaison des performances de résolution entre Simplorer et CoreLab

On voit que pour obtenir la simulation totale, l'approche par exponentielle de matrice est un peu plus longue, mais est moins consommatrice de mémoire. La différence essentielle vient quand on s'intéresse uniquement au dernier point de la simulation (ce qui est souvent le cas lors du dimensionnement). C'est ici que tout l'intérêt de l'exponentielle de matrice est visible

Simplorer intègre numériquement les équations. Cela signifie que, connaissant un point de départ, il intègre toutes les équations pas à pas jusqu'à arriver au point désiré. Il est donc obligé de calculer tous les points de la simulation avant d'arriver à la valeur du dernier point. Il met forcément le même temps pour obtenir la simulation complète ou le dernier point.
En passant par l'approche exponentielle de matrice, on obtient en fait la solution symbolique de l'équation d'état, qu'il nous reste alors à évaluer à l'instant qui nous intéresse. On peut donc calculer directement le dernier point de la simulation, sans se préoccuper de tous les états intermédiaires amenant à ce point. C'est pourquoi l'approche par exponentielle de matrice est beaucoup plus rapide et demande beaucoup moins de mémoire que Simplorer dans ce cas là. A cela s'ajoute, dans notre approche, l'obtention du jacobien des valeurs calculées en fonction des paramètres du circuit.

IV.2.2. Génération d'un composant de résolution de modèles analytiques

IV.2.2.1. Présentation de l'application

La génération de composants de résolution de modèles de dimensionnement à partir d'une description analytique du modèle est un maillon essentiel de notre approche pour le dimensionnement. Nous allons maintenant illustrer la génération de ces composants en nous basant sur l'exemple d'un alternateur à griffes destiné à des applications automobiles, dont le modèle a été développé durant les travaux de Laurent Albert [ALB]. Ces travaux ont également porté sur l'optimisation de cet alternateur, représenté sur la figure IV.7 :

Figure IV.7 : Vue de l'alternateur à griffes

Le cahier des charges de l'optimisation impose des contraintes sur les dimensions totales de l'alternateur (encombrement maximal lié à l'intégration de l'alternateur dans le moteur d'une automobile), sur sa masse (masse maximale à ne pas dépasser), ainsi que sur la puissance que doit fournir l'alternateur (puissance minimale consommée par les différents dispositifs électriques de l'automobile).

Deux objectifs ont été recherchés durant l'optimisation :

- Obtenir le meilleur rendement possible dans le cadre des contraintes exprimées par le cahier des charges, ceci afin de minimiser les pertes et donc la consommation de l'automobile.
- Obtenir la puissance massique maximale tout en conservant un rendement équivalent à la version actuellement utilisée de l'alternateur, ceci afin de prévoir l'évolution future des automobiles, dont la consommation électrique est appelée à augmenter fortement durant les prochaines années.

Les paramètres sur lesquels va jouer l'optimisation sont les dimensions géométriques de l'alternateur ainsi que les caractéristiques des conducteurs, comme leur nombre ou leur taille par exemple. Plusieurs modèles de cet alternateur ont été utilisés. Celui que nous utilisons pour cet exemple comporte 54 paramètres d'entrée, et 76 critères de dimensionnement en sortie. Ce modèle se compose principalement d'équations simples du type :

```
M_cu_stator=M_v_cu*N_phases*fils_en_parallele*L_fil_stator*S_fil_stator;
```

En plus de ces équations simples, le modèle peut également comporter des déclarations de fonctions, qui peuvent être utilisées dans les équations :

```
fem(flux)=N_spires_par_phase*abs(flux)*omega/sqrt(2);
```

Le formalisme de modélisation analytique ainsi que les différents générateurs fournis dans CoreLab sont présentés en Annexe VI.

Le modèle de l'alternateur comporte approximativement 700 lignes, ce qui représente environ 12300 octets de texte (donc 12300 caractères). Les modèles que nous pouvons traiter sont donc de taille conséquente.

Une autre caractéristique intéressante de ce modèle est qu'il comporte un système implicite d'une douzaine d'équations. C'est la procédure d'optimisation qui va résoudre ce système. Pour cela, on ajoute pour chaque équation du système implicite une contrainte supplémentaire qui devra être annulée en fin d'optimisation. Cette technique permet d'intégrer des systèmes implicites à un modèle analytique, mais requiert que l'évaluation des gradients du modèle soit extrêmement précise. En effet, une évaluation peu précise de ces derniers va amener l'algorithme d'optimisation dans une mauvaise direction, et en fin de procédure, les contraintes liées au système implicite ne seront pas annulées. Le système implicite sera donc mal résolu.

IV.2.2.2. Génération du composant

La génération du composant de calcul se déroule dans le module de génération analytique de CoreLab :

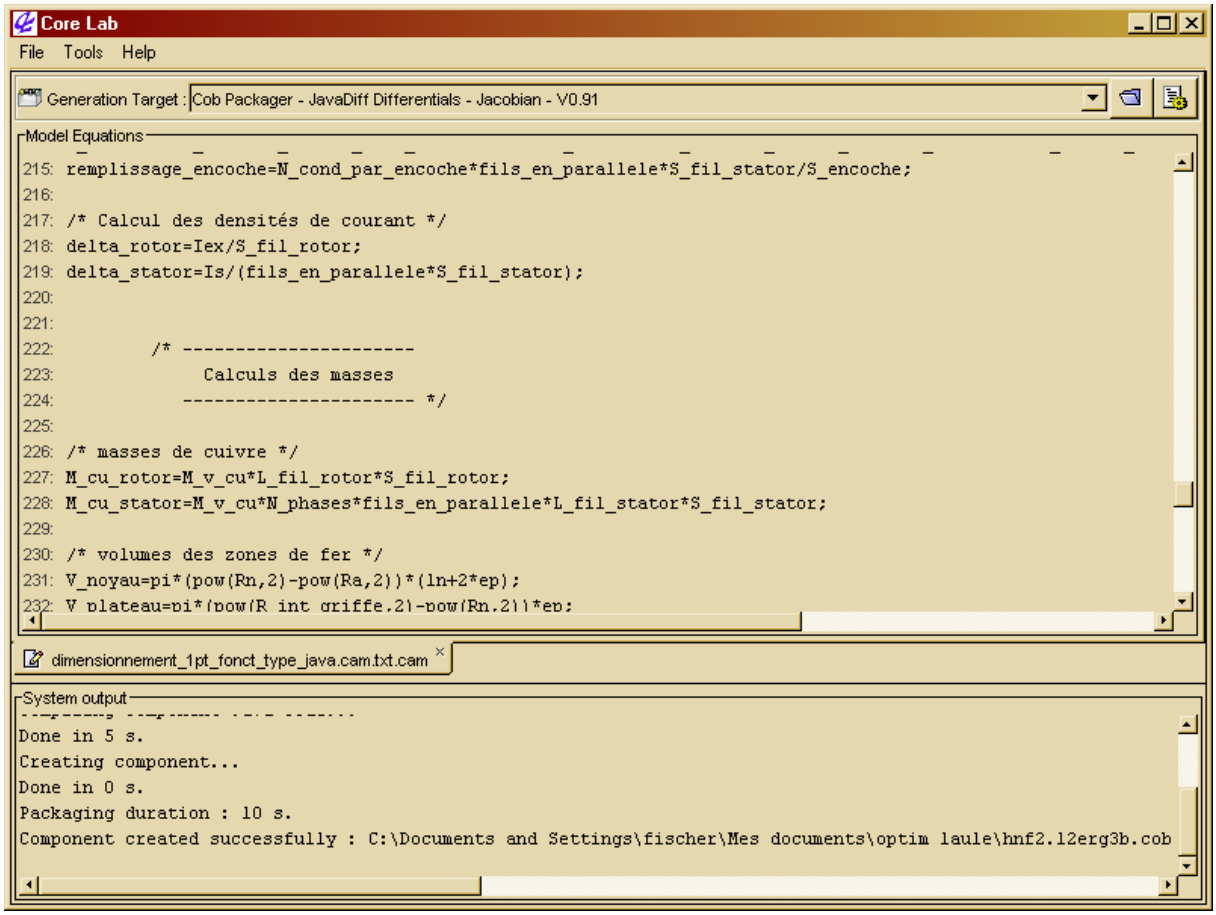


Figure IV.8 : Génération du composant de calcul dans CoreLab

Le choix parmi les différents générateurs possibles est effectué via la boite de dialogue `Generation Targets`. La génération du composant prend moins d'une trentaine de secondes. Une fois généré, le composant est directement utilisable, soit dans le cadre d'une calculette comme nous l'avons vu précédemment, soit dans le cadre d'une composition, soit pour une procédure d'optimisation.

IV.2.2.3. Optimisation de l'alternateur

L'environnement d'optimisation utilisé, CDIOptimizer, a été développé durant les travaux de David Magot [MAG]. L'utilisation de cet outil se déroule en trois phases.

Durant la première phase, on prépare la procédure d'optimisation. On commence par sélectionner le composant contenant le modèle du dispositif que l'on veut utiliser, et le composant contenant l'optimiseur. On peut ici choisir des optimiseurs utilisant ou non les gradients. On spécifie ensuite toutes les contraintes afin de définir le cahier des charges de

l'optimisation. Pour chaque paramètre d'entrée du modèle, on peut choisir parmi différentes contraintes :

- La variable peut être libre, elle peut donc prendre n'importe quelle valeur.
- La variable peut être fixée, elle a donc une valeur assignée.
- La variable peut être contrainte dans un intervalle, elle peut donc varier entre une valeur maximale et minimale.

La spécification du cahier des charges dans CDIOptimizer se fait de la manière suivante :

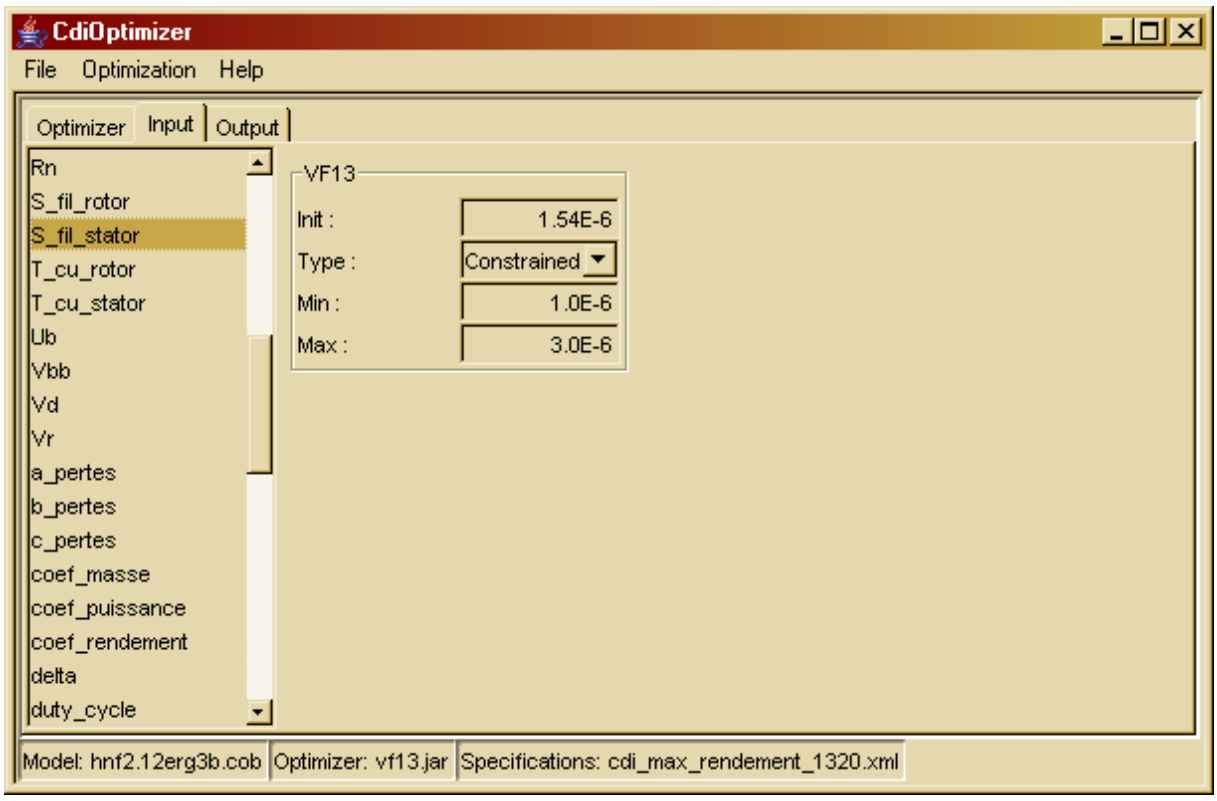


Figure IV.9 : Spécification du cahier des charges dans CDIOptimizer

Arrivé à ce stade, la procédure d'optimisation est prête à être lancée.

La deuxième phase consiste donc à lancer cette procédure. Toute l'optimisation se déroule automatiquement. L'algorithme d'optimisation spécifie des jeux de valeurs pour les paramètres d'entrée, le modèle est évalué, jusqu'à arriver à la fin de l'optimisation.

La troisième et dernière phase consiste à exploiter les résultats de l'optimisation. L'algorithme d'optimisation peut aboutir à différents résultats. L'optimisation peut se terminer avant son terme pour plusieurs raisons (rencontre d'un point non calculable, dérivées évaluées de manière imprécise, …). De plus, certains cahiers des charges peuvent comporter des contraintes incompatibles ou contradictoires. Quand l'optimisation se déroule correctement, l'algorithme détermine donc le minimum (ou le maximum) de la fonction objectif de l'optimisation.

Il suffit donc de récupérer les résultats de l'optimisation :

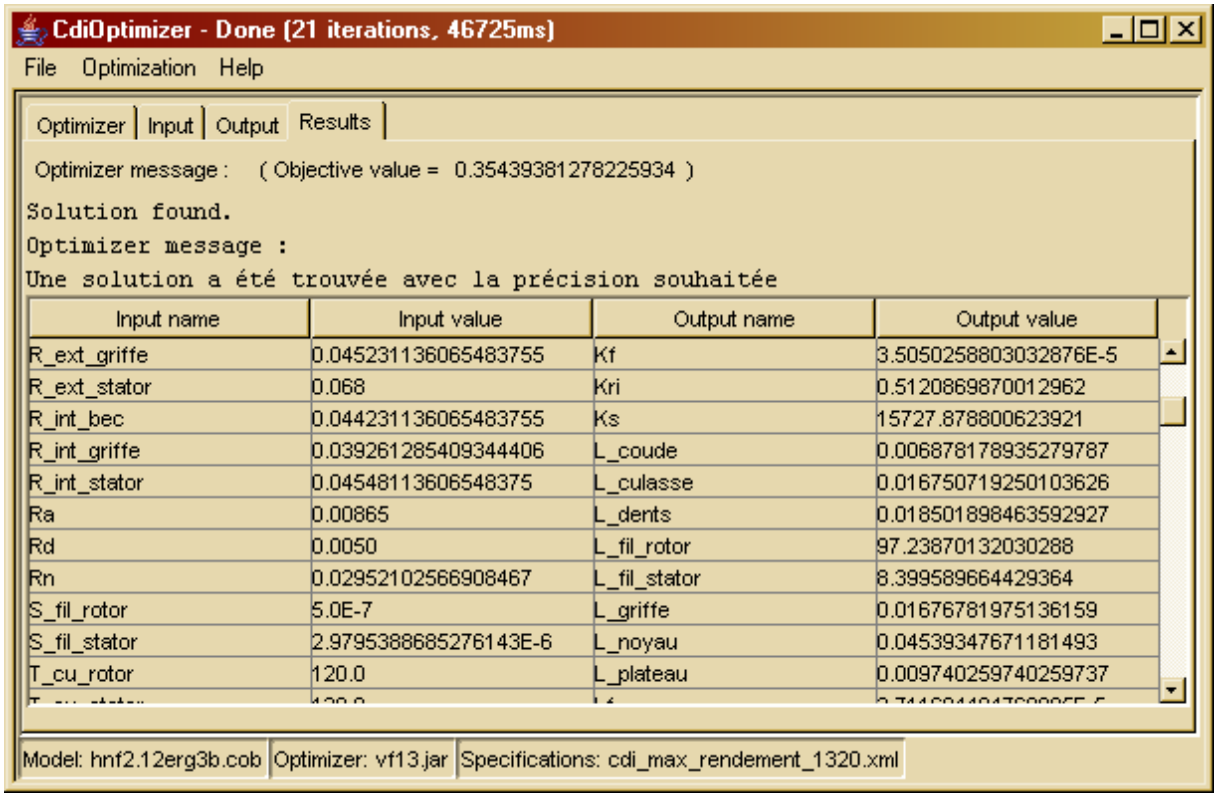


Figure IV.10 : Exploitation des résultats de l'optimisation

Nous avons ici utilisé l'algorithme d'optimisation VF13 [HSL]. Cet algorithme utilise les gradients du modèle afin de se diriger à travers l'espace des solutions.

Dans le cas de l'alternateur, les résultats de l'optimisation apportent de nettes améliorations au rendement de l'alternateur, ou permettent d'accroître la puissance massique, suivant le cahier des charges utilisé [ALB], comme le montre la figure IV.11 :

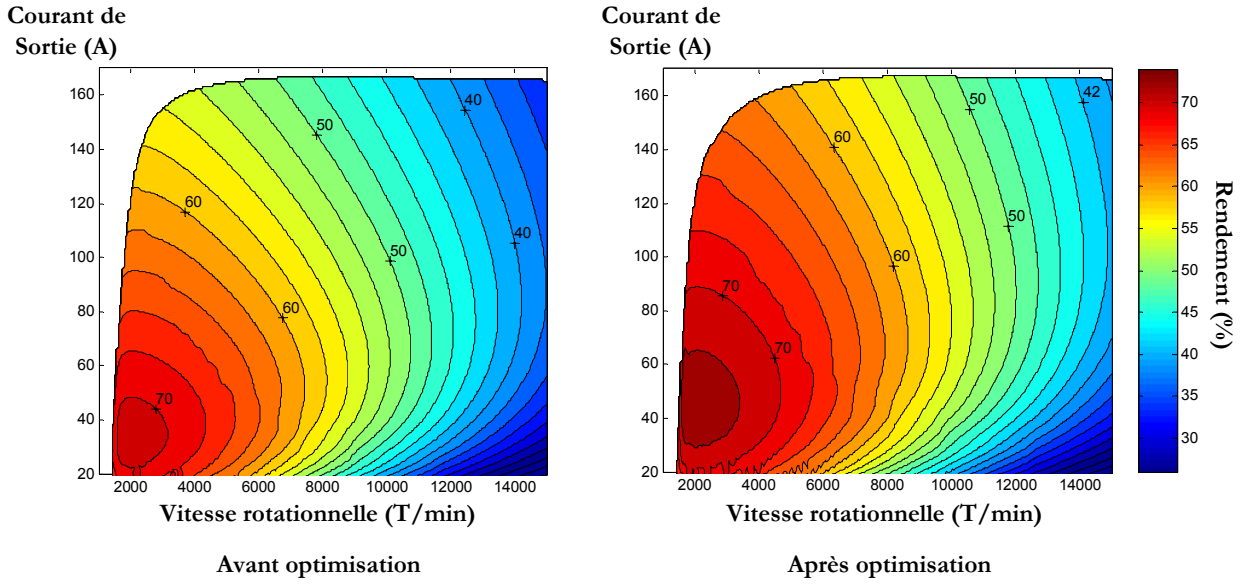


Figure IV.11 : Résultats de l'optimisation : cartographie du rendement de l'alternateur

IV.3. Utilisation de la composition hétérogène

Comme nous l'avons vu, la composition, et plus particulièrement la composition hétérogène, nous offre la possibilité d'établir un modèle de dimensionnement en s'appuyant sur différents sous-modèles, plus ou moins dépendant les uns des autres, et traitant chacun d'un aspect particulier du dispositif à dimensionner.

De plus, la composition hétérogène permet l'intégration, dans le composant de résolution du modèle, d'autres services utiles à l'optimisation, comme par exemple un service de visualisation de géométrie, ou des services de post-processing. Ces différents services permettent au concepteur d'analyser le déroulement de l'optimisation et ainsi d'affiner sa conception.

IV.3.1. Présentation de l'application

L'objectif est ici de dimensionner un transformateur triphasé, présenté sur la figure IV.12 :

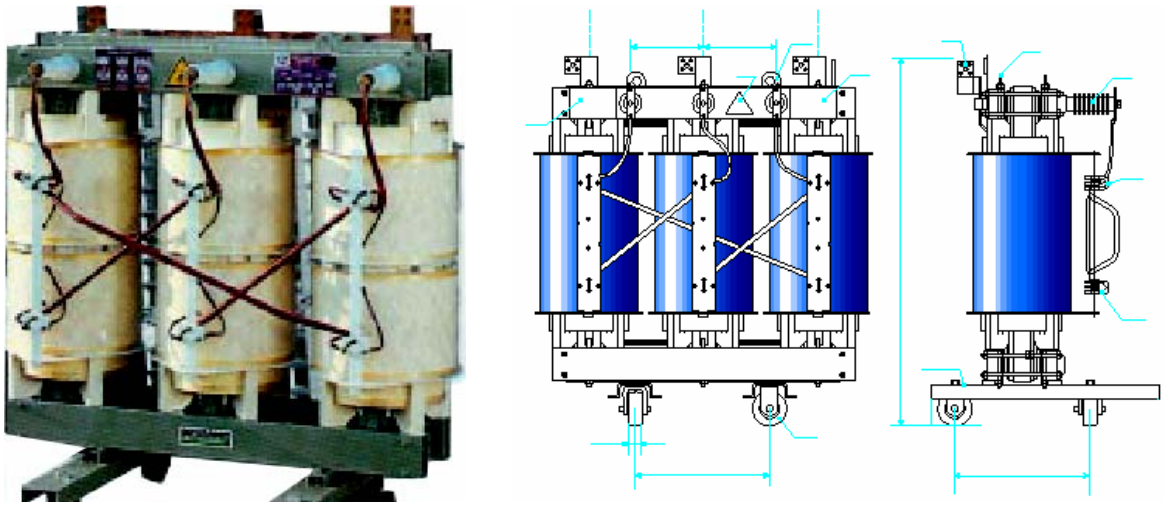


Figure IV.12 : Présentation du transformateur

Un modèle analytique de ce transformateur a été établi [PO2], puis modifié pour les besoins spécifiques du dimensionnement [FAN]. L'objectif de l'optimisation sera ici de dimensionner les différents paramètres géométriques du transformateur afin de minimiser son coût total capitalisé sur 30 ans, en respectant les contraintes imposées par le cahier des charges. Pour cela, le modèle analytique calcule les paramètres électromagnétiques (induction, densité de courant, pertes, …), les dimensions du transformateur, et le coût total, incluant la fabrication ainsi que l'utilisation du transformateur. Le modèle du transformateur a été défini à partir de quatre sous-modèles :

- Un modèle calculant les caractéristiques électromagnétiques du transformateur, comme l'induction, les pertes cuivre et fer, les réactances de fuite, …
- Un modèle calculant les paramètres géométriques du transformateur, comme la hauteur totale, le coefficient de remplissage des bobines, le volume total de cuivre, …

- Un modèle calculant les pertes en utilisation du transformateur (pertes cuivre et fer).
- Un modèle calculant le coût total du transformateur (coût des matières premières, coût des pertes capitalisées sur 30 ans).

A partir de ces différents modèles, le modèle complet du transformateur peut être établi :

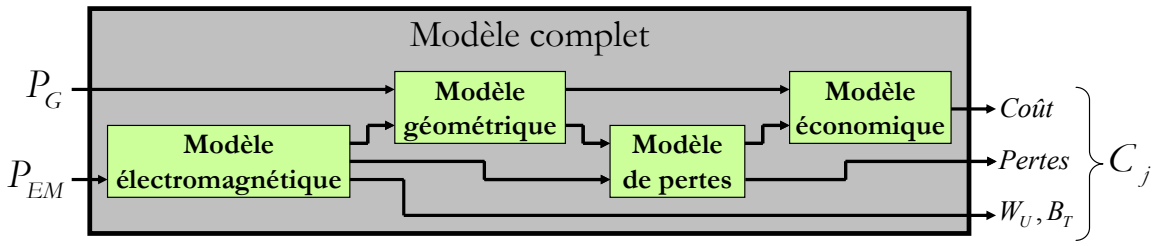


Figure IV.13 : Composition des sous-modèles définissant le modèle complet

IV.3.2. Composition du modèle

Les quatre composants nécessaires à l'établissement du modèle complet sont générés à partir des descriptions analytiques des sous-modèles comme on l'a vu au paragraphe IV.2.2. Les descriptions analytiques sont présentées en Annexe IV. En plus des composants de résolution de modèles, un composant de visualisation de la géométrie du transformateur va être ajouté dans la composition, afin de pouvoir suivre l'évolution de la géométrie en cours d'optimisation. Actuellement, ce composant doit être créé manuellement (codage effectif du dessin de la géométrie). Un générateur de composants de visualisation est actuellement à l'étude.

La composition du modèle global se déroule dans le module de composition de CoreLab :

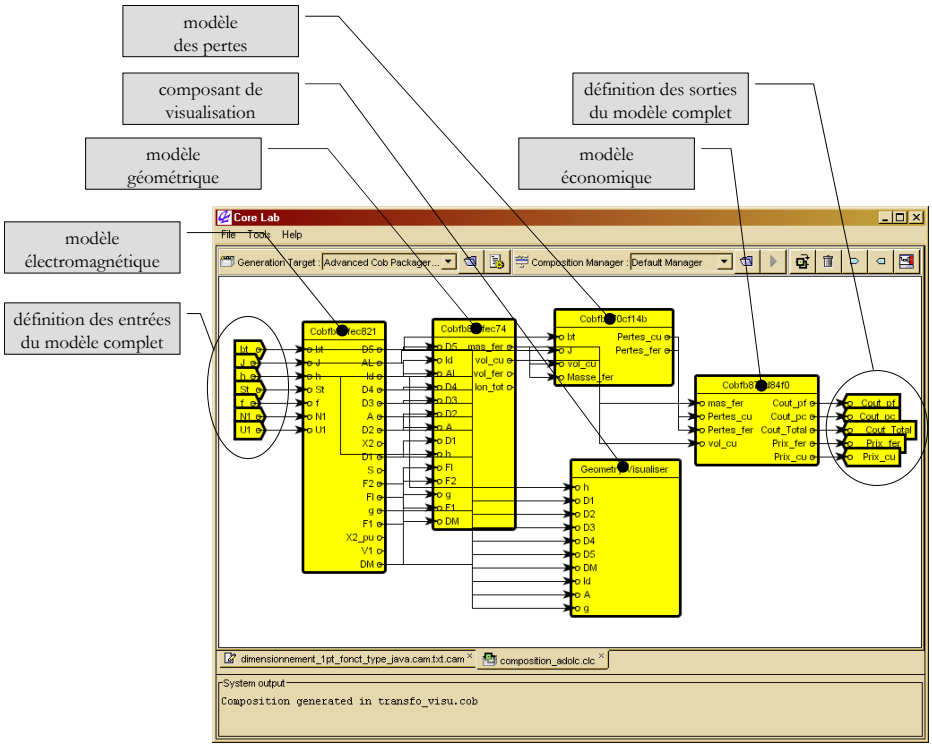


Figure IV.14 : Composition du modèle complet du transformateur sous CoreLab

Le choix parmi les différents composeurs est possible via la boite de dialogue `Generation Targets`, par un mécanisme similaire à celui utilisé dans le module de génération. En supplément, le module de composition permet de tester la composition que l'on est en train de créer sans pour autant avoir à générer le composant résultant. Cela permet de s'assurer du comportement du composant résultant. Cette fonctionnalité est accessible via la boite de dialogue `Composition Manager`.

La génération de composant à partir du schéma de composition prend moins d'une trentaine de secondes. Une fois généré, le composant est utilisable au même titre que les autres composants. Il peut bien sûr être utilisé dans une calculette ou dans une procédure d'optimisation. Il peut aussi être utilisé au sein d'une nouvelle composition. Cela offre donc une possibilité de composition récursive.

L'utilisation de ce composant dans CDIOptimizer se déroule de manière standard, à l'exception de la fenêtre de visualisation de géométrie qui apparaît au moment du chargement du composant :

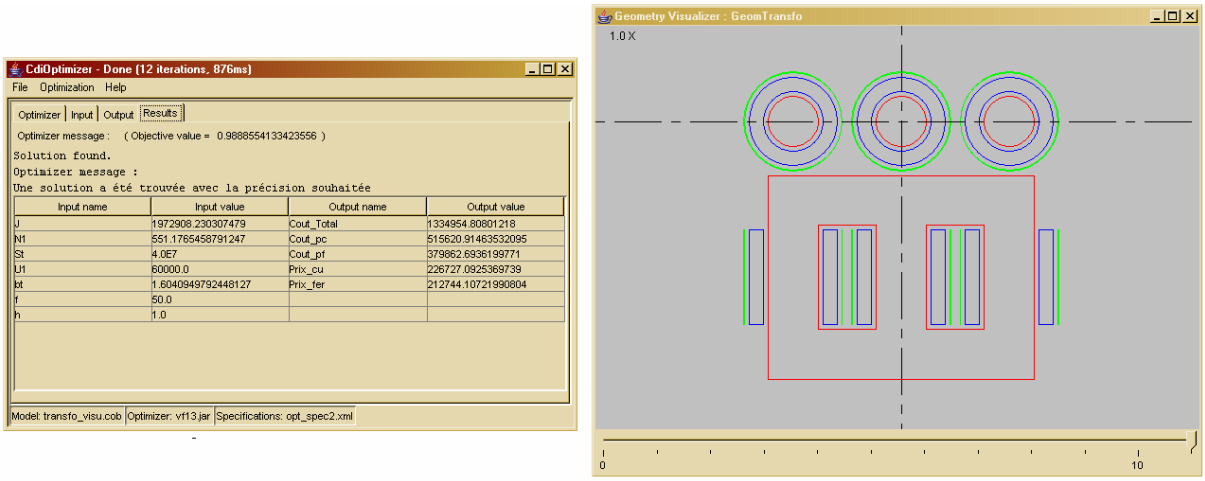


Figure IV.15 : Utilisation d'un composant de résolution contenant une visualisation de géométrie

On peut voir ici l'intérêt pour le concepteur de pouvoir disposer d'une visualisation immédiate de la géométrie du dispositif étudié.

IV.4. Comparaison des systèmes de différentiation automatique

Afin d'étudier les performances des différents systèmes de différentiation automatique, nous avons comparé leurs performances sur différents modèles de taille variable (de quelques équations à plusieurs centaines). Nous avons commencé par évaluer sur l'optimisation d'un modèle les différentes possibilités des deux différentiateurs utilisés, ADOL-C [GRI] et JavaDiff.

L'exemple sur lequel nous nous basons est l'optimisation d'un actionneur linéaire :

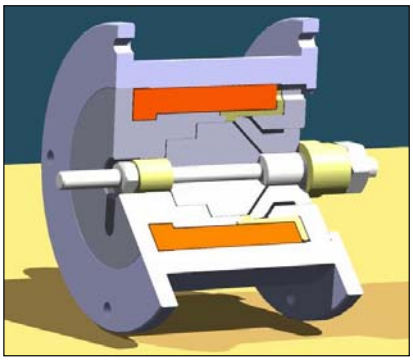
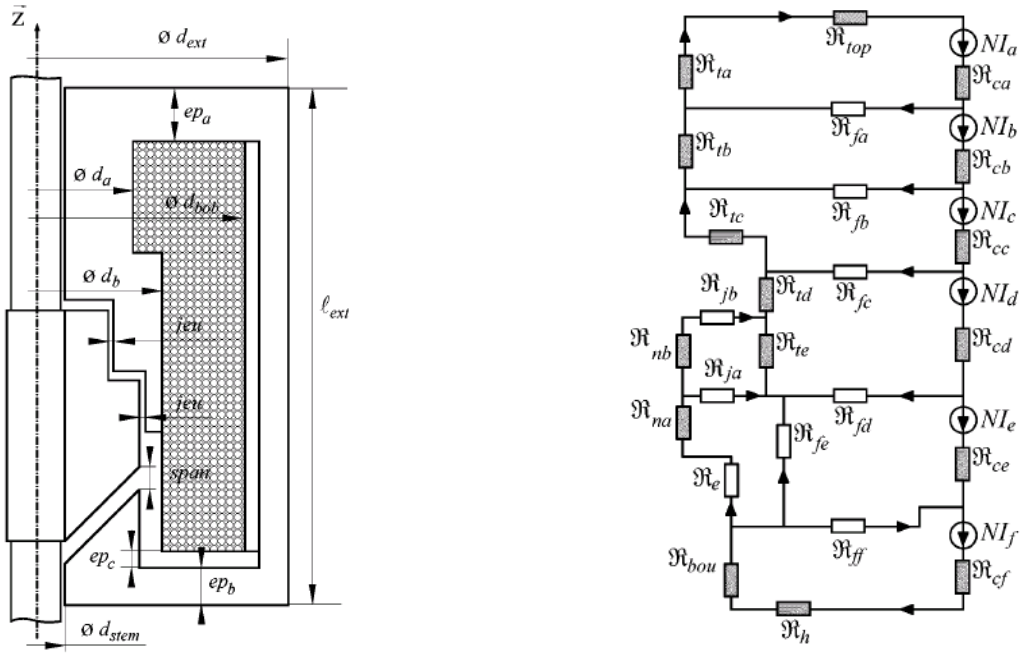


Figure IV.16 : Représentation de l'actionneur linéaire

La modélisation de cet actionneur repose sur sa représentation sous la forme d'un réseau de reluctances [CHI]. A partir de ce réseau, on peut tirer les équations régissant le comportement de l'actionneur. La modélisation de l'actionneur est présenté sur le figure IV.17 :



IV.17.a : Schéma du circuit magnétique **IV.17.b : Réseau de reluctances associé**

Figure IV.17 : Modélisation de l'actionneur linéaire

A partir de cette modélisation, on obtient donc un modèle analytique de l'actionneur linéaire. Cette modélisation analytique comporte environ 430 lignes de code (9000 octets en mode texte.) et conduit à un graphe de calcul possédant 1597 variables intermédiaires. Afin d'appréhender les différentes possibilités des systèmes de différentiation de code étudiés, nous nous sommes appuyés dans un premier temps sur une librairie déjà existante, Adol-C, qui permet la dérivation de code écrit en C / C++ par la technique dite de surcharge d'opérateurs. Il offre les deux modes de différentiation, direct et inverse. En mode inverse, il est basé sur l'approche par le graphe de calcul. Nous avons par la suite développé notre propre système de différentiation de code en Java. En effet, la majeure partie de notre environnement informatique étant en Java, il est bien

plus pratique pour nous de créer et d'exécuter du code en Java que du code en C. De plus, avoir notre propre système de différentiation nous permet de tester les différentes approches pour la différentiation évoquées au chapitre II. Enfin, cela nous permet aussi d'intercepter les erreurs dues à la dérivation bien plus facilement et plus finement que nous ne pouvons le faire dans du code C. Nous avons donc développé JavaDiff, un package de différentiation automatique en Java, qui permet la différentiation en mode direct, en mode inverse avec approche par le graphe de calcul, par substitutions régressives, et par dualité.

IV.4.1. Comparaison des performances des différentiateurs

Nous avons donc généré le modèle de l'actionneur de sept manières différentes, chacune comportant une évaluation du jacobien différente :

- Dans un premier composant, noté RAMA, les dérivées du modèle sont calculées symboliquement grâce à un outil de calcul formel, RAMA (voir Annexe I), puis le code de calcul correspondant est généré. Cette approche nous servira comme élément de référence de la comparaison.
- Le deuxième composant, noté AD-Dir, comporte un calcul de jacobien effectué avec Adol-C en mode direct.
- Dans le troisième composant, noté AD-Rev, le calcul du jacobien est fait par Adol-C en mode inverse.
- Le quatrième composant, noté JD-Dir, est basé sur JavaDiff en mode direct.
- Le cinquième composant, noté JD-ReG, utilise JavaDiff en mode inverse avec approche par le graphe de calcul.
- Le sixième composant, noté JD-ReS, utilise JavaDiff en mode inverse avec approche par substitutions régressives.
- Le septième composant, noté JD-ReD, utilise JavaDiff en mode inverse avec approche par dualité.

Nous avons comparé le nombre d'itérations et les temps d'optimisation de chacun de ces composants pour un même cahier des charges. Les résultats obtenus sont présentés sur la table IV.2 :

Modèle	**RAMA**	**AD-Dir**	**AD-Rev**	**JD-Dir**	**JD-ReG**	**JD-ReS**	**JD-ReD**
Nbr. Iter.	72	72	72	72	72	72	72
Tps. Opt. (ms)	11284	21513	14063	18348	125664	124678	118904

Table IV.2 : Temps d'optimisation et nombre d'itérations pour les différents calculs de dérivées

Si, aux erreurs numériques près (les résultats sont égaux à 10^{-12} près), les résultats sont équivalents pour les deux différentiateurs en terme de précision, les temps de calcul sont en revanche très différents. Le calcul le plus rapide est donc celui pour lequel les dérivées ont été calculées de manière symbolique, puis le code correspondant généré. C'est forcément la manière la plus efficace (en terme de rapidité du calcul) de mener le calcul des dérivées. Les systèmes basés sur la différentiation de code sont donc plus lents. En revanche, ils permettent de prendre en compte des modèles plus complexes (contenant des affectations conditionnelles par exemple).
La comparaison entre les deux modes de différentiation d'Adol-C nous montre que le mode inverse est plus rapide que le mode direct. En fait, cela est d'autant plus vrai qu'il y a beaucoup d'entrées. En effet, le mode inverse calcule la sensibilité d'une sortie par rapport aux entrées en un calcul, tandis que le mode direct a besoin d'autant de calculs qu'il y a d'entrées, car il détermine en fait l'influence d'une entrée sur les sorties.
La comparaison entre le mode direct d'Adol-C et de JavaDiff montre que ce dernier est un peu plus rapide. Cela est dû au fait que la base de nos calculs est en Java. Or pour pouvoir utiliser Adol-C, nous sommes obligés de créer du code en C, puis d'appeler ce code à partir de Java. Cette étape supplémentaire explique la légère supériorité de JavaDiff en mode direct par rapport à Adol-C. En mode inverse, les temps de JavaDiff sont sans commune mesure avec les autres mode de calcul. Cela vient du fait que JavaDiff est encore en développement et que certains points peuvent être améliorés. Par exemple, la gestion du graphe de calcul n'est pas menée de manière optimale. JavaDiff nous a permis de démontrer la viabilité des différentes méthodes de différentiation en mode inverse, mais il reste encore à améliorer la gestion des calculs, qui reste extrêmement pénalisante dans sa version actuelle.
Trois générateurs sont donc parfaitement opérationnels dans CoreLab :

- Un générateur utilisant RAMA pour calculer symboliquement les dérivées du modèle
- Un générateur utilisant Adol-C en mode inverse
- Un générateur utilisant JavaDiff en mode direct.

Nous avons ensuite comparé les performances de ces générateurs sur différents modèles :

- Le modèle simple d'un moteur, comportant une dizaine d'équations, une dizaine d'entrée et une vingtaine de sorties.
- Le modèle du transformateur présenté au paragraphe IV.3, dont chaque sous-modèle va être généré avec les différentes approches pour le calcul des dérivées, puis composé.
- Le modèle d'un actionneur linéaire, présenté précédemment, comportant une quarantaine d'entrées et une centaine de sorties
- Le modèle de l'alternateur à griffes présenté au paragraphe IV.2

Nous allons comparer les optimisations de chaque application pour les différentes manières de dériver les modèles. Quelque soit la manière envisagée pour calculer les dérivées, les résultats de l'optimisation sont identiques, en termes de résultat et de nombre d'itérations. Cela valide le fait que les performances en terme de précision numérique des différentes approches sont équivalentes.

Seuls les temps de calcul varient entre une approche et une autre :

Modèle	Nombre d'itérations			Temps d'optimisation (ms)		
	RAMA	Adol-C	JavaDiff	RAMA	Adol-C	JavaDiff
Transformateur	12	12	12	130	150	11500
Moteur	94	94	94	450	860	13000
Actionneur linéaire	72	72	72	3000	13000	18300
Alternateur à griffes	21	21	21	3280	15500	68800

Table IV.3 : Comparaison des performances des différentes manières de calculer les dérivées

La similarité des résultats et du nombre d'itérations pour les différentes approches confirme le fait que les dérivées sont calculées de manière identique quelque soit l'approche utilisée. En effet, cela signifie que l'algorithme d'optimisation emprunte le même chemin au travers de l'espace des solutions.

En termes de rapidité de calcul, si les approches par RAMA et par Adol-C varient linéairement en fonction de la taille du modèle, on voit bien que les temps de calcul de JavaDiff varient exponentiellement quand le modèle devient trop gros. Cela est dû à la mauvaise gestion du calcul dans JavaDiff.

IV.4.2. Limites d'utilisation des différentiateurs

L'étude et l'utilisation des méthodes de différentiation de code nous a permis de déterminer leurs limites d'utilisation. Ces limites sont toutes liées à la notion de persistance du graphe de calcul. Il faut en effet, pour que la différentiation se passe correctement, que le graphe de calcul ne soit pas directement dépendant des paramètres par rapport auxquels on différentie. On peut illustrer cette notion grâce au code de calcul suivant :

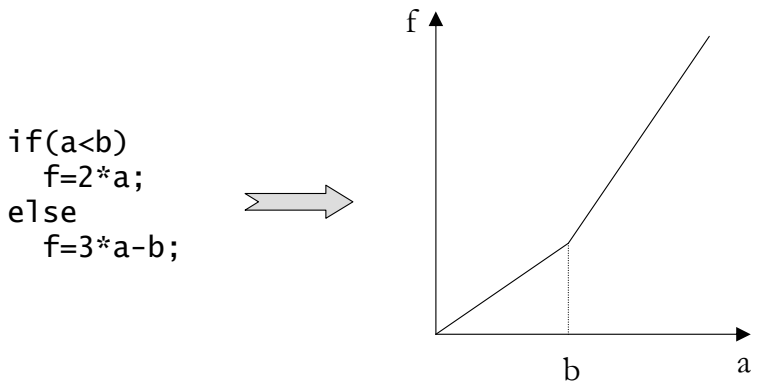


Figure IV.18 : Code de calcul et fonction correspondante

Dans ce cas, le point bloquant est la différentiation de ce code par rapport à a au point $a = b$. En effet, tant que $a < b$ ou $a > b$, le graphe de calcul est fixé, même s'il est différent d'un côté ou de l'autre. Par contre, au point $a = b$, le graphe de calcul est directement dépendant de a. Ce code n'est donc pas différentiable en ce point. D'ailleurs, la fonction mathématique correspondante à ce code de calcul n'est pas non plus dérivable en ce point. En tout point non dérivable mathématiquement, le graphe de calcul sera directement dépendant des variables par rapport auxquelles on veut différentier.

Mais la non-dérivabilité mathématique n'est pas le seul cas où la persistance du graphe de calcul n'est pas assurée. Dans le cas des algorithmes d'intégration numérique d'équations différentielles, par exemple, le graphe de calcul est directement dépendant des bornes d'intégration. En effet, ces algorithmes fonctionnant pas à pas entre les points de départ et d'arrivée, on comprend aisément que le nombre de pas de calcul, et donc le graphe de calcul, sont directement fonctions des bornes d'intégration. On ne pourra donc pas différentier les solutions obtenues par intégration numérique par rapport aux bornes de cette intégration (donc par rapport au temps final de la simulation).

IV.4.3. Conséquences des limites d'utilisation de la dérivation de code

Lors de l'intégration de méthodes de résolution d'équation différentielles numériques (du type Runge-Kutta 44 par exemple) à la résolution des modèles de dimensionnement, Edouard Dezille, au cours de son DEA, a rencontré de sérieuses difficultés liées à la persistance du graphe de calcul. Si la dérivation de code fonctionne parfaitement pour la majorité des paramètres, la dérivation de l'intégration d'équations différentielles par rapport au pas de calcul de l'algorithme d'intégration ou aux bornes ne fonctionne pas. Cela peut être un point bloquant, notamment dans le cas d'optimisations de temps de réponses par exemple. Afin de pallier à ce manque, d'autres approches sont actuellement envisagées, comme l'utilisation de la dérivation de code pour connaître toutes les dérivées qu'elle est capable de fournir, associée à d'autres méthodes basées sur des calculs symboliques pour obtenir les dérivées par rapport aux bornes :

$$\frac{\partial}{\partial a}\left(\int_a^b f(X,t)dt\right) = -f(X,a) \qquad \text{(IV.15)}$$

$$\frac{\partial}{\partial b}\left(\int_a^b f(X,t)dt\right) = f(X,b) \qquad \text{(IV.16)}$$

IV.5. Avantages et limites de la démarche proposée

Nous allons maintenant faire un bilan des avantages procurés par l'utilisation de cette démarche, ainsi que de ses limites.

Au niveau de l'utilisation de CoreLab, un ensemble de générateurs, de composeurs, et déjà plusieurs projeteurs sont disponibles. C'est un logiciel ouvert satisfaisant aux besoins de la recherche et de l'industrie. Le concepteur peut donc s'appuyer sur un logiciel qui permet d'obtenir des résultats rapidement. D'un autre côté, le développeur peut enrichir CoreLab en lui ajoutant de nouveaux modules.

Notre approche s'appuie sur le standard de composants ICAr. Ce standard ouvert, adapté aux besoins de la conception, et plus particulièrement du dimensionnement, laisse la possibilité aux développeurs d'enrichir les composants existants en intégrant de nouveaux services, ou de nouvelles manières de rendre des services existants.

Notre approche de la résolution des systèmes différentiels linéaires permet d'obtenir directement la résolution du système pour n'importe quel instant, sans avoir à simuler tous les transitoires ayant conduit à cet état. Cette approche est donc nettement plus rapide que les résolutions numériques existantes. De plus, elle permet d'obtenir les gradients des solutions par rapport aux paramètres d'entrée du modèle sans passer par une méthode numérique de type différences finies. Là encore, cela procure un gain par rapport aux méthodes numériques en terme de fiabilité (difficultés de réglages et instabilités numériques des différences finies).

Evidemment, comme toute approche, notre méthodologie possède aussi ses limites.

Tout d'abord, l'utilisation d'un standard comme ICAr impose au développeur de l'accepter et de s'y intégrer. De même, comme tout standard, son utilisation est limitée aux outils qui l'adoptent. Actuellement, en dehors de CoreLab, deux outils s'appuient sur la norme ICAr :

- L'outil de génération de composants de résolution à partir de réseaux de reluctances développé par Bertrand du Peloux au cours de ses travaux de thèse.
- Les composants de visualisation de géométrie, développés par Franck Verdière au cours de ses travaux de thèse.

La portabilité des composants que nous créons, particulièrement ceux qui s'appuient sur Adol-C ou sur l'approche par exponentielle de matrices, n'est pas assurée. En effet, ceux-ci contiennent du code natif (en C ou en Fortran), et ne sont donc pas portables en l'état. Afin d'assurer leur portabilité, il faudrait générer les code natifs pour plusieurs environnements (Linux, Windows) et les inclure dans les composants.

Notre approche pour la résolution des systèmes d'états ne traite que le cas de systèmes linéaires. Des méthodes s'appuyant sur des décompositions en séries sont actuellement étudiées pour résoudre de manière symbolique les systèmes d'état non linéaires.

Enfin, l'utilisation de la différentiation de code dans le cadre de modèles de dimensionnement est limitée par la persistance du graphe de calcul, comme nous l'avons vu au paragraphe IV.4.

CONCLUSION

CONCLUSION

Au cours de ces travaux de thèse, nous avons proposé une approche basée sur les composants logiciels pour le dimensionnement. Pour cela, nous avons défini un nouveau standard unifié de composant : ICAr. Nous avons également développé un environnement de gestion de ces composants : CoreLab. Cet environnement ouvert est adapté à la fois pour les besoins de l'ingénierie et de la recherche. L'ingénieur de bureau d'étude peut ainsi gérer les composants utilisés pour le dimensionnement, et plus particulièrement les trois aspects fondamentaux de l'utilisation des composants : la génération, la composition, et la projection. D'autre part, le développeur de nouveaux types de services pour les composants, ou de nouvelles manières de générer des services peut intégrer ses travaux au sein de CoreLab par le biais de son architecture modulaire. C'est ainsi qu'Edouard Dezille a défini trois générateurs de composants de type ICAr.

De plus, nous avons défini plusieurs générateurs de composants de résolution de modèles de dimensionnement, à partir de plusieurs types de description de modèles, ainsi que plusieurs façons de traiter la différentiation de ces modèles. Dans ce contexte, des composants permettant de résoudre des systèmes d'équations différentielles, ou encore des composants générés à partir d'une modélisation analytique (équations simples, fonctions) peuvent être générés rapidement et facilement. Certains générateurs permettent également de générer des composants satisfaisant à une autre norme qu'ICAr.

Pour la composition, nous avons défini deux outils permettant de composer les modèles de dimensionnement et les composants de visualisation de géométries. Tout comme pour les générateurs, de nouveaux composeurs peuvent également être intégrés dans CoreLab.

En complément, nous avons mis en œuvre trois projeteurs entre différentes normes de composants.

Pour le dimensionnement, nous utilisons l'environnement d'optimisation CDIOptimizer, développé durant les travaux de thèse de David Magot, en respectant la norme des composants qu'il requiert.

En ce qui concerne la résolution des équations différentielles, lorsqu'elles sont linéaires, nous avons proposé des méthodes permettant non seulement de les résoudre de manière symbolique, mais aussi de les différentier par rapport aux paramètres du modèle, sans passer par une méthode de type différences finies. Pour les équations différentielles non linéaires, des solutions ont été proposées par le passé pour encapsuler des simulations temporelles de manière "manuelle", notamment par Loig Allain. Ces encapsulations pourront s'appuyer sur la norme ICAr et ainsi bénéficier de l'ensemble des services qu'elle apporte. D'autres solutions ont également été

envisagées au cours des travaux d'Edouard Dezille, comme par exemple l'utilisation combinée d'algorithmes d'intégration numérique de type Runge-Kutta et d'algorithmes de dérivation de code, avec les limitations que nous avons évoquées dans le rapport.

Différentes perspectives s'offrent à nous suite à ces travaux.

Au niveau de la résolution des équations différentielles, des méthodes permettant de traiter numériquement des systèmes non linéaires sans simuler tous les transitoires semblent prometteuses et méritent d'être étudiées malgré leurs limites. Des réflexions dans ce sens ont été commencées, via des développements en série, en collaboration avec le Laboratoire de Modélisation et de Calcul de l'INPG.

Aider à l'encapsulation des simulations temporelles ou par éléments finis dans des composants ICAr serait un plus considérable afin de traiter des comportements dynamiques complexes, à l'aide par exemple de modélisations hybrides ou mixtes.

Il serait aussi intéressant pour le concepteur de pouvoir décrire les équations différentielles de ses systèmes au niveau même de la description complète de ses modèles de dimensionnement. De plus en plus, la génération des composants de résolution de modèles de dimensionnement devra prendre en compte la diversité de la description de ces modèles. Ceci est un élément important pour résoudre par une approche globale certains problèmes de causalité liés à une description modulaire des modèles (par composition des composants) développés dans les travaux de Benoit Delinchant.

L'extension des générateurs de composants devra bien sûr satisfaire aux différents services compris dans la norme ICAr tout en intégrant de nouvelles manières de traiter des modèles de natures différentes : par exemple, l'appel à des codes programmés, ou le traitement des tableaux de valeurs par interpolation.

Cette extension devra se faire en parallèle avec le développement de nouveaux services pour la norme ICAr. En l'occurrence, il faudra faire en sorte que CDIOptimizer, qui traite actuellement des composants de résolution des modèles basés sur les différentielles, puisse utiliser des composants ICAr fournissant des jacobiens, ce qui est plus naturel et judicieux dans l'optique des optimiseurs à base de gradients qu'il utilise.

Une extension des possibilités de composition de CoreLab devra aussi avoir lieu, notamment pour une meilleure gestion de l'hétérogénéité.

ANNEXE I

I.1. Fonctionnement général de RAMA **135**
I.2. Le modèle MOM **137**
I.3. Ecriture des règles **138**
I.3.1. Attributs du ruleset 138
I.3.2. Format des règles 138
I.3.3. Description du contexte 139
I.3.4. Description du résultat 139
I.3.5. Formalisme MOMPath 140
I.3.6. Exemple de règle 140
I.4. Fonctionnement détaillé de RAMA **141**
I.4.1. Structure et fonctionnement du parser d'expressions 141
I.4.2. Structure du ROM 144
I.4.3. Les actions élémentaires 145
I.4.4. La création de la liste d'actions à partir du résultat 145
I.4.5. Fonctionnement du moteur d'application de règles 146
I.5. Conclusion **146**
I.6. Exemple d'un fichier de règles **147**

ANNEXE I

OUTIL DE TRAITEMENT FORMEL

RAMA : RULE APPLICATOR FOR MATHEMATICAL ANALYSIS

A différentes reprises au cours des travaux, le besoin d'un outil léger[1] de traitement formel des expressions mathématiques s'est fait sentir. L'utilisation de moteurs de calcul formel externes (comme celui fourni par Maple® par exemple) était trop lourde au vu des traitements envisagés. Nous avons donc développé un outil, RAMA (Rule Applicator for Mathematical Analysis), avec les contraintes suivantes :

- Il doit permettre d'effectuer des traitements formels simples (dérivation, séparation des parties réelles et imaginaires, …) sur des expressions mathématiques classiques
- Il doit être rapide, étant donné le grand nombre d'appels à l'outil, et donc également peu gourmand en mémoire
- Il doit être indépendant de tout autre pièce logicielle
- Il doit être facilement extensible : l'ensemble des besoins en traitement n'étant pas connu *a priori*, il faut donc pouvoir programmer de nouveaux traitements facilement et rapidement
- Il doit être pilotable depuis un autre logiciel

RAMA a été développé en collaboration avec Loig Allain, doctorant de l'équipe CDI du LEG qui travaille sur la capitalisation et le traitement des modèles pour le dimensionnement.

I.1. Fonctionnement général de RAMA

Afin que l'outil soit facilement extensible, il faut découpler la programmation des traitements proprement dits de la structure logicielle qui applique ces traitements. La solution pour mettre en œuvre ce découplage est de créer un outil à base de règles. L'application d'un traitement (par exemple la dérivation) à une expression se passe donc en deux phases :

- Le chargement en mémoire des règles définissant le traitement à partir du fichier dans lequel elles sont décrites
- L'application de ces règles sur l'expression

[1] Outil léger : outil rapide, de faible occupation mémoire, et rapide à utiliser (quelques lignes d'instructions suffisent)

Le fonctionnement général de RAMA se présente donc de la manière suivante :

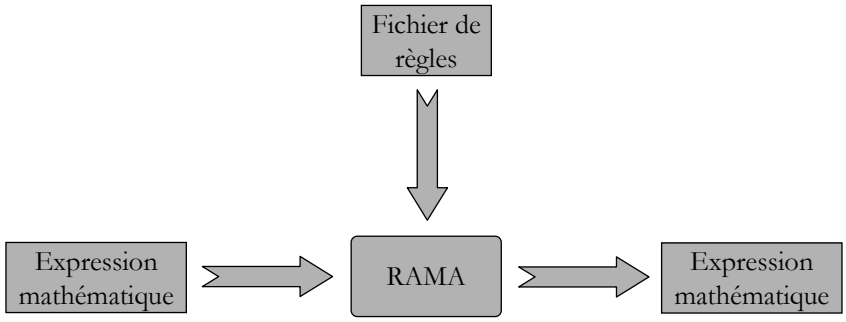


Figure I.1 : Fonctionnement général de RAMA

L'utilisateur (que ce soit le concepteur ou un logiciel utilisant RAMA) fournit une expression mathématique et un fichier de règles à appliquer. RAMA commence par transformer l'expression en une représentation sous forme d'arbre. Puis un traitement est appliqué sur les nœuds de l'arbre. Enfin, l'arbre représentant le résultat est transformé en une nouvelle expression mathématique, résultat de l'application des règles sur l'expression de départ. Ce fonctionnement est illustré sur la figure 2 :

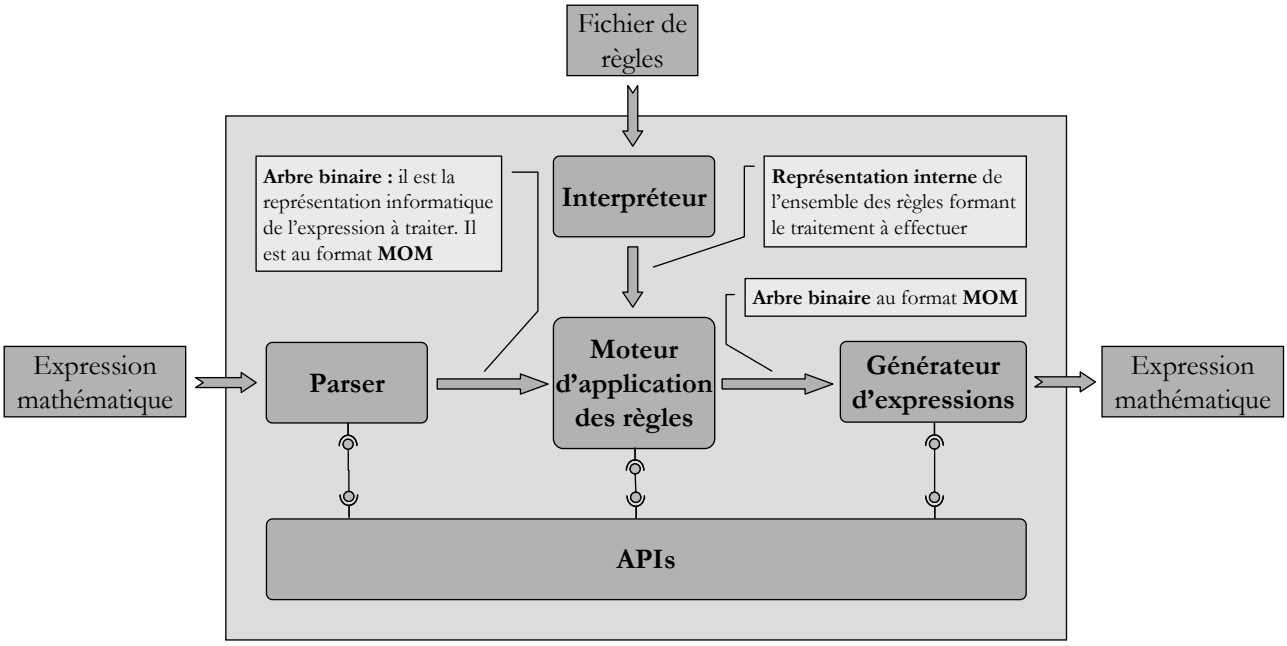


Figure I.2 : Architecture interne de RAMA

Le traitement à appliquer est décrit par ensemble de règles, chacune constituée par deux éléments :

- Un contexte d'application, qui est la description de la sous-structure de l'arbre sur laquelle la règle s'applique
- Un résultat, qui reproduit la structure de l'arbre résultant de l'application de la règle sur le contexte

I.3.a : représentation du contexte **I.3.b : représentation du résultat**

Figure I.3 : Exemple de description d'une règle : la dérivation d'une somme

I.2 Le modèle MOM

Afin de pouvoir travailler sur les expressions mathématiques, celles-ci doivent être traduites dans un modèle approprié aux traitements formels. Ce modèle est appelé MOM (Mathematical Object Model). Le format MOM est basé sur une représentation sous forme d'arbre. Les nœuds de l'arbre peuvent être de quatre types suivant ce qu'ils représentent :

- Les constantes identifient tous les opérandes constants de l'expression (entiers, réels, paramètres formels, …). Ces nœuds sont des feuilles[1].
- Les variables identifient les grandeurs variables pour le traitement formel en cours (par exemple, lors d'une dérivation partielle par rapport à **x**, seul **x** est variable, tous les autres paramètres sont constants). Ces nœuds sont des feuilles.
- Les opérateurs identifient tous les opérateurs mathématiques classiques : +, -, *, / et les deux opérateurs de signe + et –. Ces nœuds sont binaires[2] sauf les opérateurs de signes qui sont unaires[3].
- Les fonctions représentent toutes les fonctions mathématiques, c'est-à-dire les fonctions mathématiques classiques (sin, cos, log, …) et les fonctions quelconques (f, g, …). Ces nœuds sont planaires[4].

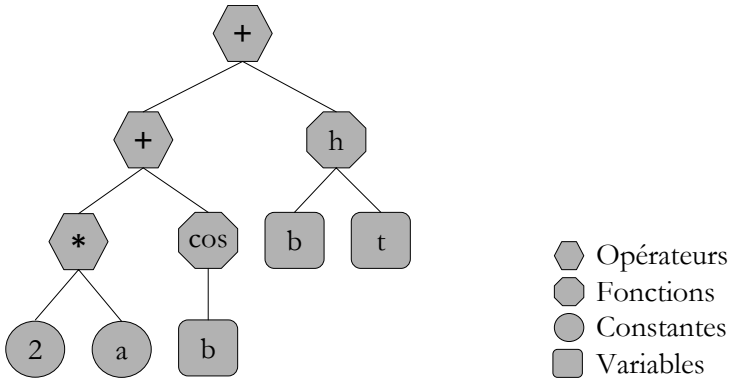


Figure I.4 : Exemple de MOM : représentation de 2a+cos(b)+h(b,t) où a est constant et b et t variables

[1] Feuilles : nœuds qui ne peuvent pas avoir d'enfant
[2] Nœuds binaires : nœuds qui ont forcément deux enfants
[3] Nœuds unaires : nœuds qui ont forcément un enfant
[4] Nœuds planaires : nœuds qui peuvent avoir n'importe quel nombre d'enfants

L'arbre est ordonné, c'est-à-dire que les enfants d'un nœud sont ordonnés, et possèdent un indice. Outre son type, chaque nœud possède aussi un nom qui lui est propre et qui représente la valeur du nœud.
Le modèle MOM a été développé par analogie avec le modèle DOM de représentation des fichiers XML.

I.3 Ecriture des règles

Comme on l'a vu, un traitement est défini par un ensemble de règles. Chaque traitement appliqué par RAMA doit être codé dans un fichier de règle, appelé *ruleset*. Ces fichiers sont au format XML, afin d'être facilement compréhensibles et éditables, dans le but de favoriser l'extensibilité de RAMA.

I.3.1. Attributs du ruleset

Un ruleset possède la structure suivante :

```
<RAMA:RULESET name="differentiate" cyclic="false">

description des règles

</RAMA:RULESET>
```

L'attribut **name** définit le nom du ruleset.
L'attribut **cyclic** définit le caractère cyclique du ruleset. Sa valeur peut être *true* ou *false*. Quand cet attribut a pour valeur *true*, le ruleset est appliqué de manière itérative sur l'expression de départ jusqu'à ce que le résultat ne varie plus d'une itération à la suivante. L'application cyclique est en particulier utilisée pour les opérations de simplification ou de factorisation. Cet attribut est optionnel.

I.3.2. Format des règles

Une règle est composée d'un contexte et d'un résultat :

```
<RAMA:RULE>
<RAMA:CONTEXT priority="1">

        description du contexte

</RAMA:CONTEXT>
<RAMA:RESULT>

        description du résultat

</RAMA:RESULT>
</RAMA:RULE>
```

Le contexte possède un attribut **priority**, qui est utilisé lorsque RAMA rencontre un nœud dans l'expression à traiter qui correspond à plusieurs contextes. Le contexte avec la priorité la plus élevée est choisi. Les priorités ne peuvent pas être négatives.

I.3.3. Description du contexte

Pour décrire le contexte, on décrit l'arbre le représentant :

```
<RAMA:MOM name="+" type="operator">
<RAMA:MOM name="0" index="2"/>
</RAMA :MOM>
```

Chaque nœud de l'arbre est décrit par la balise RAMA :MOM. On peut ensuite ajouter des tests sur le nom, le type ou l'indice du nœud :

- L'attribut **name** ajoute le test sur le nom du nœud.
- L'attribut **type** ajoute le test sur le type du nœud. Le type du nœud doit être parmi *operator*, *function*, *variable*, ou *constant*
- L'attribut **index** ajoute le test sur l'indice du nœud.

I.3.4. Description du résultat

Pour décrire le résultat, on décrit l'arbre le représentant. Pour créer un nouveau nœud, il faut donner son nom et son type :

```
<RAMA:MOM name="'f'+self@name" type="operator" …
```

Le nom peut être fixe ou des éléments concaténés par l'opérateur +. Ces éléments peuvent être des chaînes de caractère fixes ou des valeurs d'attributs récupérés par un chemin au formalisme MOMPath[1]. Pour créer l'arbre résultant du traitement du contexte, différentes actions existent :

- l'action **RAMA:apply-ruleset** permet d'appliquer un traitement (donc son ruleset correspondant) à une partie du contexte. Il faut donner le nom du ruleset à appliquer grâce à l'attribut **name** et le nœud sur lequel l'appliquer grâce à l'attribut **select**. Ce nœud est sélectionné par un chemin relatif donné au formalisme MOMPath.

  ```
  <RAMA:apply-ruleset name="differentiate" select="self"/>
  ```

- l'action **RAMA:copy** permet de copier un nœud (ainsi que tous ses enfants). Il faut donner le chemin du nœud grâce à l'attribut **select**.

  ```
  <RAMA:copy select="self"/>
  ```

- l'action **RAMA:for-each** permet de répéter une action pour tous les nœuds sélectionnés grâce à l'attribut **select**.

  ```
  <RAMA:for-each select="self/child::*"> action </RAMA:for-each>
  ```

[1] Voir le paragraphe Formalisme MOMPath

I.3.5. Formalisme MOMPath

Le formalisme MOMPath permet de donner de chemins relatifs à travers un arbre MOM. Il permet d'aller récupérer des nœuds aussi bien que des attributs déterminés.

```
self/child::[name='+']/child::[1]@name
```

Chaque noeud du chemin est séparé par **/**. Chaque nœud est donné relativement au précédent, à l'exception de **self**. Les nœuds peuvent être des types suivants :

- **self** : il désigne le nœud courant. On retrouve ce nœud obligatoirement à chaque début de MOMPath (caractère relatif de MOMPath).
- **child::** : il désigne les fils du nœud précédent. Comme il y a potentiellement plusieurs résultats possibles, il faut donner un opérateur de sélection, qui peut être du type :
  - ***** : tous les nœuds sont alors sélectionnés
  - **[entier n]** : sélectionne le $n^{ième}$ nœud
  - **[name='…']** : sélectionne le nœud avec le nom correspondant
  - **[type='…']** : sélectionne le nœud avec le type correspondant
- **brother::** : il désigne les frères du nœud précédent. Comme pour **child::**, il faut lui ajouter un opérateur de sélection.
- **father** : il désigne le nœud parent du nœud précédent.

Si on veut sélectionner un attribut d'un nœud particulier, une fois le nœud désigné par son chemin, il suffit d'ajouter à la fin du chemin **@** et le type de l'attribut :

- **name** pour récupérer le nom
- **type** pour récupérer le type

I.3.6. Exemple de règle

Voici la règle pour la derivation de la fonction **sin** :

```
<RAMA:RULE>
<RAMA:CONTEXT priority="2">
        <RAMA:MOM name="sin"/>
</RAMA:CONTEXT>
<RAMA:RESULT>
        <RAMA:MOM name="*" type="operator">
                        <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
                <RAMA:MOM name="cos" type="function">
                        <RAMA:copy select="self/child::*"/>
                </RAMA:MOM>
        </RAMA:MOM>
</RAMA:RESULT>
</RAMA:RULE>
```

I.4. Fonctionnement détaillé de RAMA

I.4.1. Structure et fonctionnement du parser d'expressions

Le parser d'expressions est chargé d'analyser les expressions données en entrée à RAMA , et de transformer ces expressions dans le format MOM. Il est constitué de deux unités principales, une unité d'analyse syntaxique et une unité de construction d'arbre.

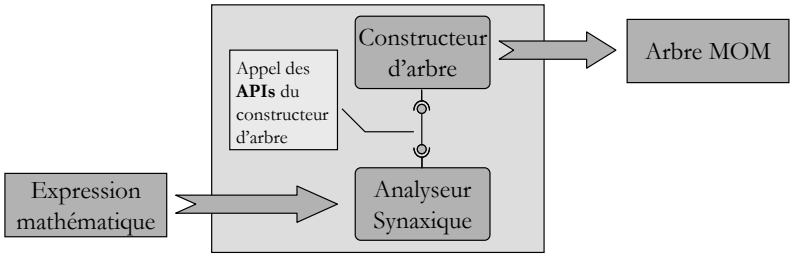


Figure I.5 : Structure interne du parser d'expressions

L'analyseur syntaxique parcours l'expression et vérifie que la syntaxe est correcte. A chaque fois qu'il tombe sur un élément remarquable (comme un opérateur ou une fonction par exemple), il le signale au constructeur d'arbre par le biais de ses APIs[1]. Les différentes informations que peut passer l'analyseur au constructeur sont les suivantes :

- Un **identifiant** a été trouvé. Un identifiant représente toutes les variables ou constantes formelles.
- Un **début de fonction** a été trouvé.
- Une **fin de fonction** a été trouvée.
- Un **réel** a été trouvé.
- Une **liste d'éléments** a été trouvée.
- Un **opérateur** a été trouvé.
- Une **parenthèse ouvrante** a été trouvée.
- Une **parenthèse fermante** e été trouvée.
- La **fin de l'expression** a été trouvée.

Le constructeur crée l'arbre en fonction des informations fournies par l'analyseur syntaxique.

Deux modes de construction de l'arbre sont possibles :

- Le mode linéaire.
- Le mode branche à branche.

[1] API : Application Programming Interface. Voir glossaire.

Par exemple, lorsqu'un nœud "+" est rencontré, la première branche du nœud est connue (c'est la branche représentant la première partie de l'expression, qui a donc déjà été créée). Il faut donc ajouter le nœud "+" en haut de cette branche. On rencontre ensuite un nouvel élément. Deux stratégies sont alors possibles. Dans le mode de construction linéaire de l'arbre, le nœud représentant le nouvel élément rencontré est directement ajouté au nœud "+" en tant que deuxième enfant. Le problème posé par cette méthode est que le nœud qui vient d'être ajouté en tant que deuxième enfant n'est pas forcément le nœud définitif à ajouter. Dans le cas de l'expression suivante :

$$a + b * c$$

La construction de l'arbre en mode linéaire se déroule de la façon suivante :

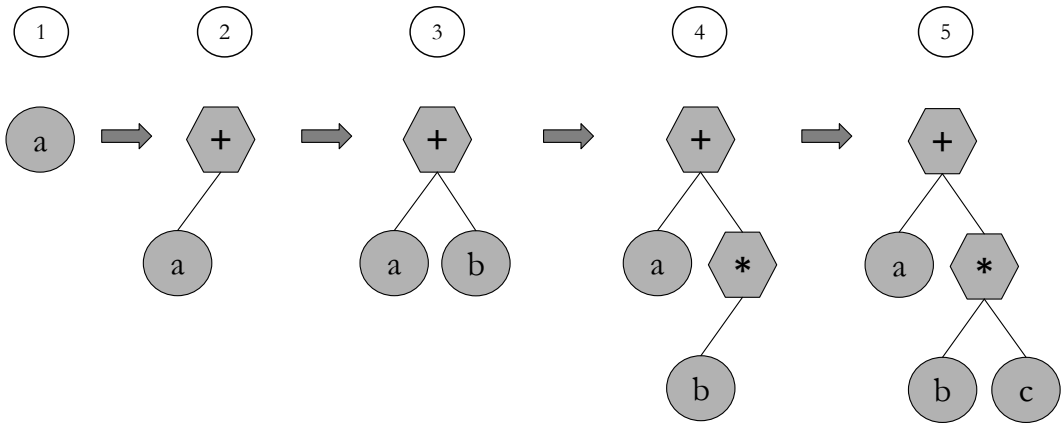


Figure I.6 : Construction de l'arbre en mode linéaire

Lors de l'étape 1, l'élément rencontré est le premier, et donc forme le début de l'arbre. Puis, dans l'étape 2, un nœud "+" doit être ajouté en tant que parent du nœud précédent. Puis le nœud "b" est rencontré dans l'étape 3, et donc ajouté en tant que deuxième enfant du nœud "+", puisque nous sommes ici dans le mode linéaire de construction. Dans l'étape 4, un nœud "*" est rencontré. Comme on vient d'ajouter le nœud "b" comme deuxième enfant de l'arbre, les règles de priorité de calcul obligent à remplacer ce nœud par le nœud "*", et d'ajouter à ce nœud le nœud "b" qui devient son enfant. On voit tous les problèmes posés par ce mode de construction (ajout, soustraction, substitution de nœud directement sur le résultat final). De plus, tous les cas que l'on peut rencontrer au niveau des enchaînements de priorités de calcul ne peuvent pas être prévus puisqu'il y en a une infinité.

La construction de l'arbre en mode branche à branche suit une stratégie différente. Le principe est de créer des constructeurs de branche qui vont s'appeler les uns les autres afin de construire la totalité de l'arbre.

Dans le cas de l'expression traitée précédemment, la construction en mode branche à branche se fait de la façon suivante :

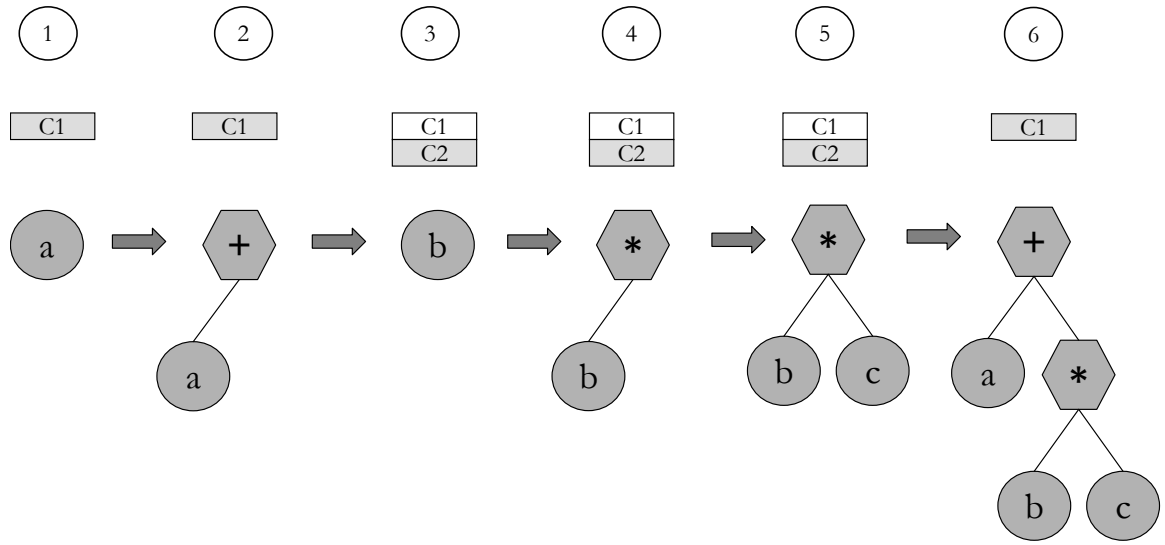


Figure I.7 : Construction de l'arbre en mode branche à branche

Lors de la première étape, le constructeur C1 est actif. Il crée donc un arbre commençant par le nœud "a". Puis le nœud "+" est rencontré et ajouté en tête de l'arbre. Il faut donc maintenant créer la deuxième branche du nœud "+". Comme on ne peut pas savoir à l'avance quel va être le nœud racine de cette branche, on crée un deuxième constructeur C2, chargé de construire la deuxième branche du nœud "+". Le constructeur C1 est donc désactivé et attend le résultat que va lui fournir C2. Dans les étapes 3, 4 et 5, le constructeur C2 construit le deuxième branche du nœud "+". Finalement, dans l'étape 6, quand la fin de l'expression est rencontrée, le constructeur C2 passe la branche qu'il vient de construire à C1 afin qu'il l'ajoute en tant que deuxième enfant du nœud "+".

Ce mode de construction est donc mieux adapté à la construction des arbres représentant des expressions mathématiques. On peut facilement donner des règles d'arrêt à chaque constructeur. Par exemple, quand un nœud "+" est rencontré, le constructeur chargé de construire son deuxième enfant ne s'arrêtera que lorsque la fin de l'expression aura été trouvée. Par contre, lorsqu'un nœud "*" est rencontré, le constructeur chargé de construire la deuxième branche s'arrêtera dès qu'un nœud "+" ou la fin de l'expression est rencontrée. On gère ainsi facilement les priorités de calcul.

De même, la création de nœuds particuliers (comme les nœuds de fonction qui peuvent avoir un nombre indéfini d'enfants) sont facilement gérés par des types de constructeurs particuliers.

Dans le constructeur d'arbre de RAMA, on dispose ainsi de trois types de constructeurs différents.

Le constructeur général, chargé de la construction de tous les nœuds (opérateurs, variables, constantes) sauf des nœuds de fonctions. Les règles d'arrêt de ce constructeur peuvent varier suivant le cadre dans lequel ils sont créés, comme on vient de le voir.
Le constructeur des nœuds de fonction, dont la règle d'arrêt est que la fin de la fonction que l'on est en train de construire a été trouvée.
Le constructeur des expressions entre parenthèses, qui ne s'arrête que si la parenthèse fermante a été trouvée.
Grâce à ces trois constructeurs, la construction de l'arbre est gérée facilement et de manière beaucoup plus robuste que dans le mode de construction linéaire.

I.4.2. Structure du ROM

Le ROM (Rule Object Model) est la représentation interne à RAMA des règles qui lui ont été fournies par les fichiers de règles. RAMA possède un catalogue de tous les différents rulesets disponibles. Ce catalogue est stocké dans le **RuleSetFactory**, qui est le module permettant de gérer les différents rulesets. Chaque ruleset possède un certain nombre de règles. Ces règles possèdent chacune un contexte dans lequel elles s'appliquent, et une liste d'actions élémentaires à effectuer pour construire le résultat donné dans le fichier de règles. Un contexte est constitué d'une liste de tests à effectuer sur un MOM afin de déterminer si ce MOM est compatible avec ce contexte.
Globalement, le ROM peut donc se présenter de la manière suivante :

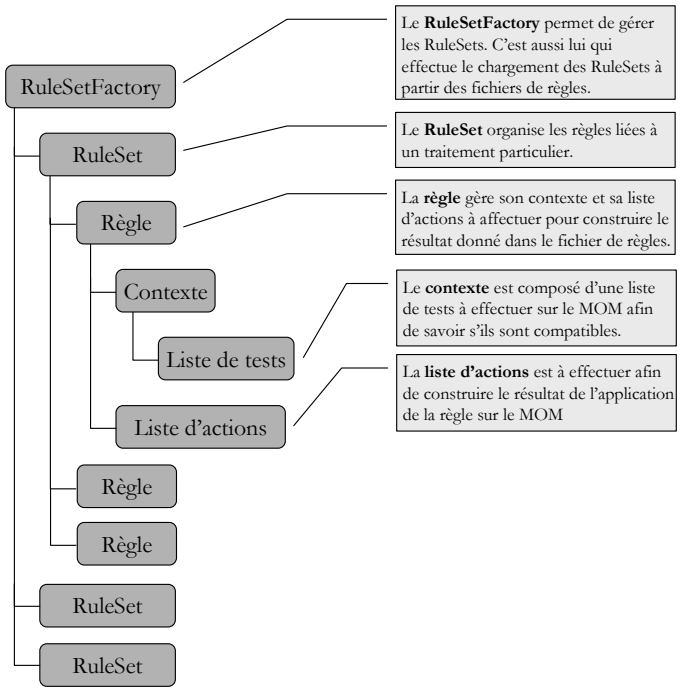


Figure I.8 : Structure du Rule Object Model

I.4.3. Les actions élémentaires

Comme on l'a vu dans l'écriture des règles, le résultat de l'application de la règle est donné sous la forme d'un arbre. A partir de cet arbre, le RuleSetFactory doit inférer la liste des actions élémentaires qui, effectuées dans le bon ordre, vont donner le résultat escompté. Les actions élémentaires mise à la disposition du RuleSetFactory sont les suivantes :

- **InstanciateAction** : cette action permet d'instancier (de créer) un nouveau nœud de l'arbre. Elle possède comme paramètres le nom et le type du nœud à créer.
- **AddAction** : cette action permet d'ajouter un enfant au nœud courant.
- **AddActionResultAction** : cette action permet d'ajouter le résultat d'une action au nœud courant.
- **RemoveAction** : cette action permet d'enlever un enfant du nœud courant.
- **ApplyRuleSetAction** : cette action permet d'appliquer un RuleSet à un certain nœud.
- **ForEachAction** : cette action permet de répéter une ou plusieurs actions pour différents contextes.
- **CopyAction** : cette action permet de copier un nœud.

Une nouvelle action peut être ajouté à RAMA (ceci afin de favoriser l'extensibilité du logiciel). Néanmoins, l'ajout d'une nouvelle action élémentaire nécessite l'écriture d'un nouveau RuleSetFactory, ce qui requiert des compétences spécifiques en programmation.

I.4.4. La création de la liste d'actions à partir du résultat

Comme on l'a vu dans l'écriture des règles, un résultat est composé de quatre types d'éléments différents. Chacun de ces éléments peut être traduit par une suite d'actions élémentaires.

RAMA:MOM est traduit par la suite :

AddActionResultAction

InstanciateAction

puis la liste des actions pour créer la sous structure dont ce nœud est la racine

RAMA:apply-ruleset est traduit la suite :

AddActionResultAction

ApplyRuleSetAction

RAMA:for-each est traduit par la suite :

AddActionResultAction

ForEachAction

l'action à répéter

RAMA :copy est traduit par la suite :

AddActionResultAction

CopyAction

Une fois le fichier de règles analysé, toutes les règles pour faire le traitement sont en mémoire. Le ruleset est donc complet et prêt à être utilisé.

I.4.5. Fonctionnement du moteur d'application de règles

Le fonctionnement du moteur d'application de règle est illustré sur la figure 9 :

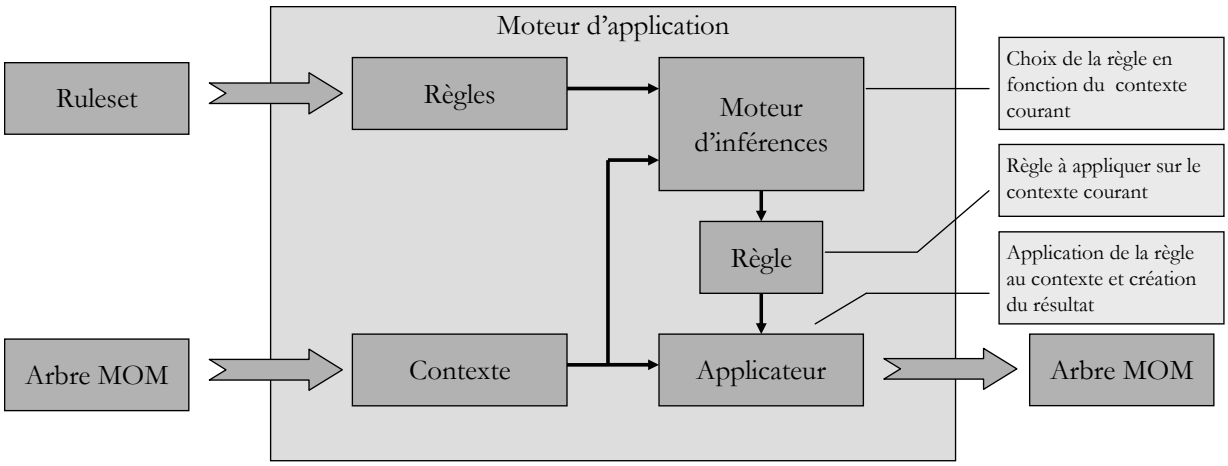


Figure I.9 : Principe de fonctionnement du moteur d'application de règles

L'arbre MOM représentant l'expression de départ est stocké en tant que contexte d'application. A partir du ruleset, les règles le composant sont stockées en mémoire. En connaissant le contexte d'application et les différentes règles constituant le ruleset, le moteur d'inférence sélectionne là règle à appliquer. Il commence par comparer le contexte d'application propre à chaque règle constituant le ruleset et le compare à la structure du MOM d'entrée. Si le MOM est compatible avec le contexte d'application d'une règle, alors cette règle est applicable au MOM. Dans le cas où le MOM d'entrée est compatible avec plusieurs contextes, c'est alors la priorité de la règle qui détermine quelle règle sera appliquée. Si le MOM d'entrée n'est compatible avec aucun contexte, une erreur est alors générée.

Puis la règle est appliquée au contexte et l'arbre résultant du traitement est crée. L'applicateur prend le contexte et applique une à une la suite d'actions élémentaires déduite du ficher de règles.

I.5. Conclusion

Avec RAMA, nous disposons d'un outil de traitement formel léger et rapide, permettant d'effectuer des actions mathématiques telles que la dérivation d'expressions, la séparation des parties réelles et imaginaires des expressions complexes, ou d'autres. RAMA est également indépendant de tout autre pièce logicielle.

L'intérêt principal de RAMA est que les règles sont codées de manière simple et rapide. La "programmation" d'un traitement est donc facilement réalisable sans pré requis particulier en informatique ou programmation.

RAMA est donc facilement extensible, tant au niveau des traitements possibles (écriture d'un nouveau fichier de règles sans pré requis informatique) qu'au niveau de ses principales composantes qui peuvent être changées ou adaptées pour des besoins particuliers (nécessité d'une bonne connaissance de la programmation et de Java).

I.6. Exemple d'un fichier de règles

Afin d'illustrer l'écriture d'un fichier de règles, nous donnons ici le fichier de règles pour la dérivation.

```
<RAMA:RULESET name="differentiate">
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
              <RAMA:MOM name="+"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
              <RAMA:MOM name="+" type="operator">
                <RAMA:for-each select="self/child::*">
                      <RAMA:apply-ruleset name="differentiate" select="self"/>
                </RAMA:for-each>
              </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
              <RAMA:MOM name="-"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
              <RAMA:MOM name="-" type="operator">
                <RAMA:for-each select="self/child::*">
                      <RAMA:apply-ruleset name="differentiate" select="self"/>
                </RAMA:for-each>
              </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
              <RAMA:MOM name="*"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
              <RAMA:MOM name="+" type="operator">
                <RAMA:for-each select="self/child::*">
                      <RAMA:MOM name="*" type="operator">
                        <RAMA:apply-ruleset name="differentiate" select="self"/>
                        <RAMA:for-each select="self/brother::*">
                              <RAMA:copy select="self"/>
                        </RAMA:for-each>
                      </RAMA:MOM>
                </RAMA:for-each>
              </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
              <RAMA:MOM name="/"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
              <RAMA:MOM name="/" type="operator">
                <RAMA:MOM name="-" type="operator">
                      <RAMA:MOM name="*" type="operator">
                        <RAMA:apply-ruleset name="differentiate" select="self/child::[1]"/>
                        <RAMA:copy select="self/child::[2]"/>
                      </RAMA:MOM>
                      <RAMA:MOM name="*" type="operator">
                        <RAMA:apply-ruleset name="differentiate" select="self/child::[2]"/>
                        <RAMA:copy select="self/child::[1]"/>
                      </RAMA:MOM>
                </RAMA:MOM>
                <RAMA:MOM name="pow" type="function">
```

```
                <RAMA:copy select="self/child::[2]"/>
                <RAMA:MOM name="2" type="constant"/>
            </RAMA:MOM>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="++"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="++" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="--"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="--" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="cos"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="*" type="operator">
            <RAMA:MOM name="--" type="operator">
                <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
            </RAMA:MOM>
            <RAMA:MOM name="sin" type="function">
                <RAMA:copy select="self/child::*"/>
            </RAMA:MOM>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="sin"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="*" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
            <RAMA:MOM name="cos" type="function">
                <RAMA:copy select="self/child::*"/>
            </RAMA:MOM>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="tan"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="*" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
            <RAMA:MOM name="+" type="operator">
                <RAMA:MOM name="1" type="constant"/>
                <RAMA:MOM name="pow" type="function">
                  <RAMA:MOM name="tan" type="function">
                      <RAMA:copy select="self"/>
                  </RAMA:MOM>
                  <RAMA:MOM name="2" type="constant"/>
                </RAMA:MOM>
            </RAMA:MOM>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="arccos"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="--" type="operator">
            <RAMA:MOM name="/" type="operator">
                <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
                <RAMA:MOM name="sqrt" type="function">
                  <RAMA:MOM name="-" type="operator">
                      <RAMA:MOM name="1" type="constant"/>
                      <RAMA:MOM name="pow" type="function">
                        <RAMA:copy select="self/child::*"/>
                        <RAMA:MOM name="2" type="constant"/>
```

```
                        </RAMA:MOM>
                    </RAMA:MOM>
                </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
            <RAMA:MOM name="arcsin"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="/" type="operator">
              <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
              <RAMA:MOM name="sqrt" type="function">
                  <RAMA:MOM name="-" type="operator">
                    <RAMA:MOM name="1" type="constant"/>
                    <RAMA:MOM name="pow" type="function">
                        <RAMA:copy select="self/child::*"/>
                        <RAMA:MOM name="2" type="constant"/>
                    </RAMA:MOM>
                  </RAMA:MOM>
              </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
            <RAMA:MOM name="arctan"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="*" type="operator">
              <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
              <RAMA:MOM name="/" type="operator">
                  <RAMA:MOM name="+" type="operator">
                    <RAMA:MOM name="1" type="constant"/>
                    <RAMA:MOM name="pow" type="function">
                        <RAMA:copy select="self/child::*"/>
                        <RAMA:MOM name="2" type="constant"/>
                    </RAMA:MOM>
                  </RAMA:MOM>
              </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
            <RAMA:MOM name="cosh"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="*" type="operator">
              <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
              <RAMA:MOM name="sinh" type="function">
                  <RAMA:copy select="self/child::*"/>
              </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
            <RAMA:MOM name="sinh"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="*" type="operator">
              <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
              <RAMA:MOM name="cosh" type="function">
                  <RAMA:copy select="self/child::*"/>
              </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
            <RAMA:MOM name="tanh"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="*" type="operator">
              <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
              <RAMA:MOM name="-" type="operator">
                  <RAMA:MOM name="1" type="constant"/>
                  <RAMA:MOM name="pow" type="function">
                    <RAMA:copy select="self"/>
                    <RAMA:MOM name="2" type="constant"/>
                  </RAMA:MOM>
              </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
```

```
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="arccosh"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="*" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
            <RAMA:MOM name="/" type="operator">
                  <RAMA:MOM name="sqrt" type="function">
                    <RAMA:MOM name="+" type="operator">
                          <RAMA:MOM name="pow" type="function">
                            <RAMA:copy select="self/child::*"/>
                            <RAMA:MOM name="2" type="constant"/>
                          </RAMA:MOM>
                          <RAMA:MOM name="--" type="operator">
                            <RAMA:MOM name="1" type="constant"/>
                          </RAMA:MOM>
                    </RAMA:MOM>
                  </RAMA:MOM>
            </RAMA:MOM>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="arcsinh"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="*" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
            <RAMA:MOM name="/" type="operator">
                  <RAMA:MOM name="sqrt" type="function">
                    <RAMA:MOM name="+" type="operator">
                          <RAMA:MOM name="pow" type="function">
                            <RAMA:copy select="self/child::*"/>
                            <RAMA:MOM name="2" type="constant"/>
                          </RAMA:MOM>
                          <RAMA:MOM name="1" type="constant"/>
                    </RAMA:MOM>
                  </RAMA:MOM>
            </RAMA:MOM>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="arctanh"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="*" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
            <RAMA:MOM name="/" type="operator">
                  <RAMA:MOM name="+" type="operator">
                    <RAMA:MOM name="1" type="constant"/>
                    <RAMA:MOM name="--" type="operator">
                          <RAMA:MOM name="pow" type="function">
                            <RAMA:copy select="self/child::*"/>
                            <RAMA:MOM name="2" type="constant"/>
                          </RAMA:MOM>
                    </RAMA:MOM>
                  </RAMA:MOM>
            </RAMA:MOM>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="exp"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="*" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
            <RAMA:MOM name="exp" type="function">
                  <RAMA:copy select="self/child::*"/>
            </RAMA:MOM>
          </RAMA:MOM>
    </RAMA:RESULT>
</RAMA:RULE>
<RAMA:RULE>
    <RAMA:CONTEXT priority="2">
          <RAMA:MOM name="log"/>
    </RAMA:CONTEXT>
    <RAMA:RESULT>
          <RAMA:MOM name="*" type="operator">
            <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
            <RAMA:MOM name="/" type="operator">
```

```
                    <RAMA:copy select="self/child::*"/>
                </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
            <RAMA:MOM name="sqrt"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="*" type="operator">
              <RAMA:apply-ruleset name="differentiate" select="self/child::*"/>
              <RAMA:MOM name="/" type="operator">
                    <RAMA:MOM name="2" type="constant"/>
                    <RAMA:copy select="self"/>
              </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="2">
            <RAMA:MOM name="pow"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="*" type="operator">
              <RAMA:copy select="self/child::[2]"/>
              <RAMA:MOM name="*" type="operator">
                    <RAMA:apply-ruleset name="differentiate" select="self/child::[1]"/>
                    <RAMA:MOM name="pow" type="function">
                      <RAMA:copy select="self/child::[1]"/>
                      <RAMA:MOM name="+" type="operator">
                            <RAMA:copy select="self/child::[2]"/>
                            <RAMA:MOM name="--" type="operator">
                              <RAMA:MOM name="1" type="constant"/>
                            </RAMA:MOM>
                      </RAMA:MOM>
                    </RAMA:MOM>
              </RAMA:MOM>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="1">
            <RAMA:MOM type="function"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="+" type="operator">
              <RAMA:for-each select="self/child::*">
                    <RAMA:MOM name="*" type="operator">
                      <RAMA:apply-ruleset name="differentiate" select="self"/>
                      <RAMA:MOM name="'D_'+self@name+'_'+self/father@name" type="function">
                            <RAMA:for-each select="self/father/child::*">
                              <RAMA:copy select="self"/>
                            </RAMA:for-each>
                            <RAMA:for-each select="self/father/child::*">
                              <RAMA:apply-ruleset name="differentiate" select="self"/>
                            </RAMA:for-each>
                      </RAMA:MOM>
                    </RAMA:MOM>
              </RAMA:for-each>
            </RAMA:MOM>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="1">
            <RAMA:MOM type="variable"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="'D_'+self@name" type="variable"/>
        </RAMA:RESULT>
    </RAMA:RULE>
    <RAMA:RULE>
        <RAMA:CONTEXT priority="1">
            <RAMA:MOM type="constant"/>
        </RAMA:CONTEXT>
        <RAMA:RESULT>
            <RAMA:MOM name="0" type="constant"/>
        </RAMA:RESULT>
    </RAMA:RULE>
</RAMA:RULESET>
```

ANNEXE II

Analyse inverse d'erreur de l'approximation de Padé

ANNEXE II

ANALYSE INVERSE D'ERREUR DE L'APPROXIMATION DE PADE

DE L'EXPONENTIELLE DE MATRICE

Lemme 1

Enoncé

Si $\|H\| < 1$, alors $\log(I+H)$ existe et : $\|\log(I+H)\| \le \dfrac{\|H\|}{1-\|H\|}$.

Démonstration

Si $\|H\| < 1$, alors $\log(I+H) = \sum_{k=1}^{+\infty} (-1)^{k+1} \left(\dfrac{H^k}{k} \right)$, et donc :

$$\|\log(I+H)\| \le \sum_{k=1}^{+\infty} \frac{\|H\|^k}{k} \le \|H\| \sum_{k=0}^{+\infty} \|H\|^k = \frac{\|H\|}{1-\|H\|}$$

Lemme 2

Enoncé

Si $\|A\| \le \dfrac{1}{2}$ et $p > 0$, alors $\left\| D_{p,q}(A)^{-1} \right\| \le \dfrac{p+q}{p}$.

Démonstration

A partir de la définition de $D_{p,q}(A)$, on en tire :

$$D_{p,q}(A) = I + F \text{ avec } F = \sum_{j=1}^{q} \frac{(p+q-j)!\,q!}{(p+q)!\,(q-j)!} \frac{(-A)^j}{j!}$$

En utilisant le fait que :

$$\frac{(p+q-j)!\,q!}{(p+q)!\,(q-j)!} \le \left[\frac{q}{p+q} \right]^j$$

On trouve :

$$\|F\| \le \sum_{j=1}^{q} \left[\frac{q}{p+q} \|A\| \right]^j \frac{1}{j!} \le \frac{q}{p+q} \|A\| (e-1) \le \frac{q}{p+q}$$

Et donc, il vient finalement :

$$\left\| D_{p,q}(A)^{-1} \right\| = \left\| (I+F)^{-1} \right\| \le \frac{1}{(1-\|F\|)} \le \frac{(p+q)}{p}$$

Lemme 3

Enoncé

Si $\|A\| \le \frac{1}{2}$ et $q \le p$ et $p > 1$, alors $R_{p,q}(A) = e^{A+F}$ où $\|F\| \le 8\|A\|^{p+q+1} \frac{p!q!}{(p+q)!(p+q+1)!}$.

Démonstration

A partir du théorème du reste pour les approximants de Padé [VAR], on tire :

$$R_{p,q}(A) = e^{A} - \frac{(-1)^q}{(p+q)!} A^{p+q+1} D_{p,q}(A)^{-1} \int_0^1 e^{(1-u)A} u^p (1-u)^q \, du$$

Et donc :

$$e^{-A} R_{p,q}(A) = I + H \text{ avec } H = \frac{(-1)^{q+1}}{(p+q)!} A^{p+q+1} D_{p,q}(A)^{-1} \int_0^1 e^{-uA} u^p (1-u)^q \, du$$

En prenant la norme, en utilisant le lemme 2, et en notant que $\frac{p+q}{p} e^{1/2} \le 4$, on obtient :

$$\|H\| \le \frac{1}{(p+q)!} \|A\|^{p+q+1} \frac{p+q}{p} \int_0^1 e^{1/2} u^p (1-u)^q \, du \le 4\|A\|^{p+q+1} \frac{p!q!}{(p+q)!(p+q+1)!}$$

Avec l'hypothèse $\|A\| \le \frac{1}{2}$, il est possible de montrer que pour tout les p et q admissibles, $\|H\| \le \frac{1}{2}$ et donc à partir du lemme 1 :

$$\|\log(I+H)\| \le \frac{\|H\|}{1-\|H\|} \le 8\|A\|^{p+q+1} \frac{p!q!}{(p+q)!(p+q+1)!}$$

En prenant $F = \log(I+H)$, on peut voir que $e^{-A} R_{p,q}(A) = I + H = e^F$

Comme A et F commutent, cela implique $R_{p,q}(A) = e^A e^F = e^{A+F}$

Lemme 4

Enoncé

Si $\|A\| \le \frac{1}{2}$ alors $R_{p,q}(A) = e^{A+F}$ avec $F \le 8\|A\|^{p+q+1} \frac{p!q!}{(p+q)!(p+q+1)!}$.

Démonstration

Le cas $p \ge q$ et $p > 1$ est traité par le lemme 1. Si $p+q=0$, alors $F=-A$ et l'inégalité ci-dessus est donc vérifiée. Si on considère le cas $q \ge p$, $q>1$, le lemme 3 nous donne : $R_{q,p}(-A) = e^{-A+F}$ avec F qui satisfait à l'inégalité. Le lemme se démontre alors par $\|-F\| = \|F\|$ et $R_{p,q}(A) = \left[R_{q,p}(-A)\right]^{-1} = \left[e^{-A+F}\right]^{-1} = e^{A-F}$.

Théorème

Enoncé

Si $\frac{\|A\|}{2^j} \le \frac{1}{2}$, alors $\left[R_{p,q}\left(\frac{A}{2^j}\right)\right]^{2^j} = e^{A+E}$ avec :

$$\frac{\|E\|}{\|A\|} \le 8\left(\frac{\|A\|}{2^j}\right)^{p+q} \frac{p!q!}{(p+q)!(p+q+1)!} \le \left(\frac{1}{2}\right)^{p+q-3} \frac{p!q!}{(p+q)!(p+q+1)!}$$

Démonstration

A partir du lemme 4, on peut écrire : $R_{p,q}\left(\frac{A}{2^j}\right) = e^{A+F}$ avec $F \le 8\left[\frac{\|A\|}{2^j}\right]^{p+q+1} \frac{p!q!}{(p+q)!(p+q+1)!}$.

Le théorème est prouvé en notant que si $E = 2^j F$ alors : $\left[R_{p,q}\left(\frac{A}{2^j}\right)\right]^{2^j} = \left[e^{\frac{A}{2^j}+F}\right]^{2^j} = e^{A+E}$

Corollaires

Si $\frac{\|A\|}{2^j} \le \frac{1}{2}$, alors $\left[T_k\left(\frac{A}{2^j}\right)\right]^{2^j} = e^{A+E}$ où $\frac{\|E\|}{\|A\|} \le 8\left(\frac{\|A\|}{2^j}\right)^k \frac{1}{k+1} \le \left(\frac{1}{2}\right)^{k-3} \frac{1}{k+1}$.

Si $\frac{\|A\|}{2^j} \le \frac{1}{2}$, alors $\left[R_{q,q}\left(\frac{A}{2^j}\right)\right]^{2^j} = e^{A+E}$ où $\frac{\|E\|}{\|A\|} \le 8\left(\frac{\|A\|}{2^j}\right)^{2q} \frac{(q!)^2}{(2q)!(2q+1)!} \le \left(\frac{1}{2}\right)^{2q-3} \frac{(q!)^2}{(2q)!(2q+1)!}$

ANNEXE III

Structure XML de description des systèmes d'état

ANNEXE III

STRUCTURE XML DE DESCRIPTION DES SYSTEMES D'ETAT

Les fichiers XML contenant une description d'un système d'état possèdent la structure suivante :

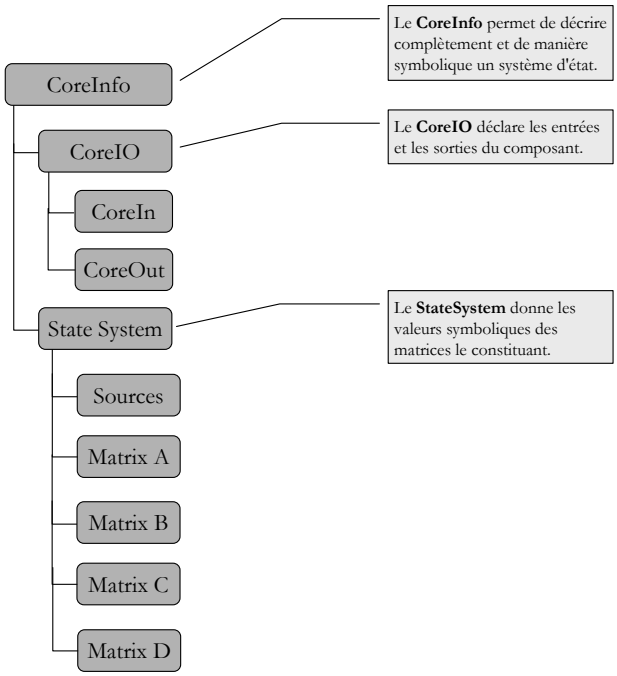


Figure III.1 : Structure du fichier XML de description des systèmes d'état

Chaque description de système d'état commence donc par la balise suivante :

```
<CoreInfo>
        description du système d'état
</CoreInfo>
```

On distingue ensuite deux éléments principaux dans la description du système d'état :

- La déclaration des entrées et sorties du composant que l'on va générer :

```
<CoreIO>
        déclaration des entrées et sorties du composant
</CoreIO>
```

- La description sous forme analytique du système d'état :

```
<StateSystem states="…" sources="…" outputs="…">
        description analytique du système d'état
</StateSystem>
```

On voit ici que l'on déclare le nombre d'états du système (attribut **states**), le nombre de sources (attribut **sources**), ainsi que le nombre de sorties (attribut **outputs**).

La déclaration d'une entrée du composant se fait de la manière suivante :

```
<CorIn name="…" default="…" />
```

Chaque entrée est définie par son nom (attribut **name**). On peut également lui attribuer une valeur par défaut (attribut **default**).

De même, la déclaration d'une sortie se fait de manière similaire :

```
<CorOut index="…" name="…" />
```

L'attribut **index** sert à identifier l'indice de l'élément du vecteur de sortie du système d'état qui donnera la valeur à cette sortie. L'attribut **name** permet de définir le nom de la sortie.

La description analytique du système d'état comporte plusieurs parties : les sources du système, puis l'état initial, et enfin les différentes matrices de l'équation d'état.

La déclaration des sources se fait de la manière suivante :

```
<Sources>

        déclaration des sources

</Sources>
```

On décrit ensuite chaque source :

```
<Source index ="…" name="…">
        <Pol d0="…" d1="…" d2="…"/>
        <Sin u="…" omega="…" phi="…" />
</Source>
```

L'attribut **index** sert à spécifier l'indice de la source dans le vecteur d'entrée. L'attribut **name** permet de spécifier le nom de la source. Ensuite, on peut spécifier la forme d'onde de la source, en deux parties. La première partie représente la partie polynomiale de la forme d'onde. Chaque attribut définit le coefficient de chaque degré du polynôme. Ainsi, l'attribut **d2** définit le terme de degré 2. La deuxième partie définit la partie sinusoïdale, sous la forme $u.sin(\omega t + \varphi)$. Chaque attribut permet de spécifier les différents paramètres de la source.

On définit ensuite l'état initial du système d'état :

```
<InitialState>

        déclaration de l'état initial

</InitialState>
```

L'état initial se déclare sous la forme d'une matrice :

```
<Matrix>
        <Line index="…">
                <Element index="…" value="…" />

                déclaration des éléments

        </Line>

        déclaration des lignes

</Matrix>
```

Pour chaque ligne, on donne son indice (attribut **index**). Puis pour chaque élément, on donne son indice (attribut **index**) ainsi qu sa valeur (attribut **value**).

Enfin, on donne la description des quatre matrices du système d'état sous forme analytique :

```
<Matrix name="…">
        <Line index="…">
                <Element index="…" value="…" />

                déclaration des éléments

        </Line>

        déclaration des lignes

</Matrix>
```

La déclaration des matrice se fait comme précédemment, à l'exception de l'attribut **name** qui permet de spécifier quelle matrice on est en train de décrire.

Le circuit utilisé au chapitre IV se présente de la manière suivante :

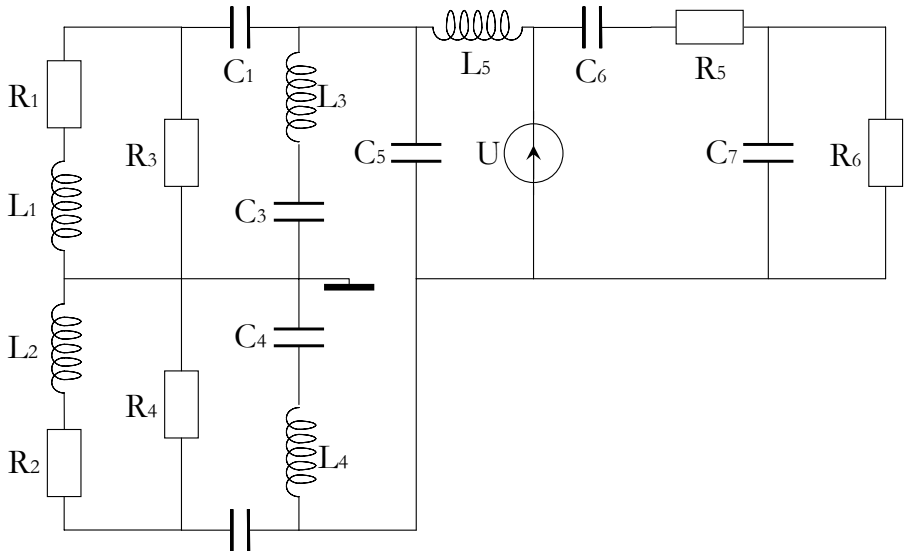


Figure III.2 : Schéma d'un convertisseur statique modélisé et de son filtre RSIL associé

Ce circuit conduit au fichier de description de système d'état suivant :

```
<CoreInfo>
   <CorIO>
        <CorIn name="r_1" default="5" />
        <CorIn name="r_2" default="5" />
        <CorIn name="r_3" default="50" />
        <CorIn name="r_4" default="50" />
        <CorIn name="r_5" default="1" />
        <CorIn name="r_6" default="400" />
        <CorIn name="c_1" default="220e-9" />
        <CorIn name="c_2" default="220e-9" />
        <CorIn name="c_3" default="1e-6" />
        <CorIn name="c_4" default="1e-6" />
        <CorIn name="c_5" default="874e-9" />
        <CorIn name="c_6" default="100e-9" />
        <CorIn name="c_7" default="278e-6" />
        <CorIn name="l_1" default="50e-6" />
        <CorIn name="l_2" default="50e-6" />
        <CorIn name="l_3" default="250e-6" />
        <CorIn name="l_4" default="250e-6" />
        <CorIn name="l_5" default="490e-6" />
        <CorIn name="uc" default="0" />
        <CorIn name="us" default="1" />
        <CorIn name="om" default="314" />
        <CorIn name="ph" default="0" />
        <CorOut index="0" name="I_L1" />
```

```
        <CorOut index="1" name="I_L2" />
        <CorOut index="2" name="I_L3" />
        <CorOut index="3" name="I_L4" />
        <CorOut index="4" name="I_L5" />
        <CorOut index="5" name="V_C1" />
        <CorOut index="6" name="V_C2" />
        <CorOut index="7" name="V_C3" />
        <CorOut index="8" name="V_C4" />
        <CorOut index="9" name="V_C5" />
        <CorOut index="10" name="V_C6" />
        <CorOut index="11" name="V_C7" />
    </CorIO>
    <StateSystem states="12" sources="1" outputs="12">
        <Sources>
           <Source index="0" name="u_v1">
                <Pol d0="uc" />
                <Sin u="us" omega="om" phi="ph" />
           </Source>
        </Sources>
        <InitialState>
           <Matrix>
                <Line index="0">
                  <Element index="0" value="0" />
                </Line>
                <Line index="1">
                  <Element index="0" value="0" />
                </Line>
                <Line index="2">
                  <Element index="0" value="0" />
                </Line>
                <Line index="3">
                  <Element index="0" value="0" />
                </Line>
                <Line index="4">
                  <Element index="0" value="0" />
                </Line>
                <Line index="5">
                  <Element index="0" value="0" />
                </Line>
                <Line index="6">
                  <Element index="0" value="0" />
                </Line>
                <Line index="7">
                  <Element index="0" value="0" />
                </Line>
                <Line index="8">
                  <Element index="0" value="0" />
                </Line>
                <Line index="9">
                  <Element index="0" value="0" />
                </Line>
                <Line index="10">
                  <Element index="0" value="0" />
                </Line>
                <Line index="11">
                  <Element index="0" value="0" />
                </Line>
           </Matrix>
        </InitialState>
        <Matrix name="A">
           <Line index="0">
                <Element index="0" value="-(r_1*(r_4+r_3)+r_3*r_4)/l_1/(r_4+r_3)" />
                <Element index="1" value="r_3*r_4/l_1/(r_4+r_3)" />
                <Element index="2" value="-r_3*r_4/l_1/(r_4+r_3)" />
                <Element index="3" value="r_3*r_4/l_1/(r_4+r_3)" />
                <Element index="4" value="0" />
                <Element index="5" value="r_3/l_1/(r_4+r_3)" />
                <Element index="6" value="-r_3/l_1/(r_4+r_3)" />
                <Element index="7" value="0" />
                <Element index="8" value="0" />
                <Element index="9" value="-r_3/l_1/(r_4+r_3)" />
                <Element index="10" value="0" />
                <Element index="11" value="0" />
           </Line>
           <Line index="1">
                <Element index="0" value="r_3*r_4/l_2/(r_4+r_3)" />
                <Element index="1" value="-(r_2*(r_4+r_3)+r_3*r_4)/l_2/(r_4+r_3)" />
                <Element index="2" value="r_3*r_4/l_2/(r_4+r_3)" />
                <Element index="3" value="-r_3*r_4/l_2/(r_4+r_3)" />
                <Element index="4" value="0" />
                <Element index="5" value="r_4/l_2/(r_4+r_3)" />
                <Element index="6" value="-r_4/l_2/(r_4+r_3)" />
                <Element index="7" value="0" />
                <Element index="8" value="0" />
                <Element index="9" value="-r_4/l_2/(r_4+r_3)" />
                <Element index="10" value="0" />
                <Element index="11" value="0" />
```

```
</Line>
<Line index="2">
    <Element index="0" value="-r_3*r_4/l_3/(r_4+r_3)" />
    <Element index="1" value="r_3*r_4/l_3/(r_4+r_3)" />
    <Element index="2" value="-r_3*r_4/l_3/(r_4+r_3)" />
    <Element index="3" value="r_3*r_4/l_3/(r_4+r_3)" />
    <Element index="4" value="0" />
    <Element index="5" value="-r_4/l_3/(r_4+r_3)" />
    <Element index="6" value="-r_3/l_3/(r_4+r_3)" />
    <Element index="7" value="-(-r_4-r_3)/l_3/(r_4+r_3)" />
    <Element index="8" value="0" />
    <Element index="9" value="-r_3/l_3/(r_4+r_3)" />
    <Element index="10" value="0" />
    <Element index="11" value="0" />
</Line>
<Line index="3">
    <Element index="0" value="r_3*r_4/l_4/(r_4+r_3)" />
    <Element index="1" value="-r_3*r_4/l_4/(r_4+r_3)" />
    <Element index="2" value="r_3*r_4/l_4/(r_4+r_3)" />
    <Element index="3" value="-r_3*r_4/l_4/(r_4+r_3)" />
    <Element index="4" value="0" />
    <Element index="5" value="r_4/l_4/(r_4+r_3)" />
    <Element index="6" value="r_3/l_4/(r_4+r_3)" />
    <Element index="7" value="0" />
    <Element index="8" value="-(-r_4-r_3)/l_4/(r_4+r_3)" />
    <Element index="9" value="-r_4/l_4/(r_4+r_3)" />
    <Element index="10" value="0" />
    <Element index="11" value="0" />
</Line>
<Line index="4">
    <Element index="0" value="0" />
    <Element index="1" value="0" />
    <Element index="2" value="0" />
    <Element index="3" value="0" />
    <Element index="4" value="0" />
    <Element index="5" value="0" />
    <Element index="6" value="0" />
    <Element index="7" value="0" />
    <Element index="8" value="0" />
    <Element index="9" value="-1/l_5" />
    <Element index="10" value="0" />
    <Element index="11" value="0" />
</Line>
<Line index="5">
    <Element index="0" value="-r_3/c_1/(r_4+r_3)" />
    <Element index="1" value="-r_4/c_1/(r_4+r_3)" />
    <Element index="2" value="r_4/c_1/(r_4+r_3)" />
    <Element index="3" value="-r_4/c_1/(r_4+r_3)" />
    <Element index="4" value="0" />
    <Element index="5" value="-1/(c_1*(r_4+r_3))" />
    <Element index="6" value="1/(c_1*(r_4+r_3))" />
    <Element index="7" value="0" />
    <Element index="8" value="0" />
    <Element index="9" value="1/(c_1*(r_4+r_3))" />
    <Element index="10" value="0" />
    <Element index="11" value="0" />
</Line>
<Line index="6">
    <Element index="0" value="r_3/c_2/(r_4+r_3)" />
    <Element index="1" value="r_4/c_2/(r_4+r_3)" />
    <Element index="2" value="r_3/c_2/(r_4+r_3)" />
    <Element index="3" value="-r_3/c_2/(r_4+r_3)" />
    <Element index="4" value="0" />
    <Element index="5" value="1/(c_2*(r_4+r_3))" />
    <Element index="6" value="-1/(c_2*(r_4+r_3))" />
    <Element index="7" value="0" />
    <Element index="8" value="0" />
    <Element index="9" value="-1/(c_2*(r_4+r_3))" />
    <Element index="10" value="0" />
    <Element index="11" value="0" />
</Line>
<Line index="7">
    <Element index="0" value="0" />
    <Element index="1" value="0" />
    <Element index="2" value="-1/c_3" />
    <Element index="3" value="0" />
    <Element index="4" value="0" />
    <Element index="5" value="0" />
    <Element index="6" value="0" />
    <Element index="7" value="0" />
    <Element index="8" value="0" />
    <Element index="9" value="0" />
    <Element index="10" value="0" />
    <Element index="11" value="0" />
</Line>
<Line index="8">
    <Element index="0" value="0" />
```

```
            <Element index="1" value="0" />
            <Element index="2" value="0" />
            <Element index="3" value="-1/c_4" />
            <Element index="4" value="0" />
            <Element index="5" value="0" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="0" />
            <Element index="10" value="0" />
            <Element index="11" value="0" />
        </Line>
        <Line index="9">
            <Element index="0" value="r_3/c_5/(r_4+r_3)" />
            <Element index="1" value="r_4/c_5/(r_4+r_3)" />
            <Element index="2" value="r_3/c_5/(r_4+r_3)" />
            <Element index="3" value="r_4/c_5/(r_4+r_3)" />
            <Element index="4" value="-(-r_4-r_3)/c_5/(r_4+r_3)" />
            <Element index="5" value="1/(c_5*(r_4+r_3))" />
            <Element index="6" value="-1/(c_5*(r_4+r_3))" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="-1/(c_5*(r_4+r_3))" />
            <Element index="10" value="0" />
            <Element index="11" value="0" />
        </Line>
        <Line index="10">
            <Element index="0" value="0" />
            <Element index="1" value="0" />
            <Element index="2" value="0" />
            <Element index="3" value="0" />
            <Element index="4" value="0" />
            <Element index="5" value="0" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="0" />
            <Element index="10" value="-1/(c_6*r_5)" />
            <Element index="11" value="1/(c_6*r_5)" />
        </Line>
        <Line index="11">
            <Element index="0" value="0" />
            <Element index="1" value="0" />
            <Element index="2" value="0" />
            <Element index="3" value="0" />
            <Element index="4" value="0" />
            <Element index="5" value="0" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="0" />
            <Element index="10" value="1/(c_7*r_5)" />
            <Element index="11" value="(-r_6-r_5)/c_7/r_5/r_6" />
        </Line>
    </Matrix>
    <Matrix name="B">
        <Line index="0">
            <Element index="0" value="0" />
        </Line>
        <Line index="1">
            <Element index="0" value="0" />
        </Line>
        <Line index="2">
            <Element index="0" value="0" />
        </Line>
        <Line index="3">
            <Element index="0" value="0" />
        </Line>
        <Line index="4">
            <Element index="0" value="1/l_5" />
        </Line>
        <Line index="5">
            <Element index="0" value="0" />
        </Line>
        <Line index="6">
            <Element index="0" value="0" />
        </Line>
        <Line index="7">
            <Element index="0" value="0" />
        </Line>
        <Line index="8">
            <Element index="0" value="0" />
        </Line>
        <Line index="9">
            <Element index="0" value="0" />
        </Line>
        <Line index="10">
```

```
            <Element index="0" value="-1/(c_6*r_5)" />
        </Line>
        <Line index="11">
            <Element index="0" value="1/(c_7*r_5)" />
        </Line>
    </Matrix>
    <Matrix name="C">
        <Line index="0">
            <Element index="0" value="2" />
            <Element index="1" value="0" />
            <Element index="2" value="0" />
            <Element index="3" value="0" />
            <Element index="4" value="0" />
            <Element index="5" value="0" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="0" />
            <Element index="10" value="0" />
            <Element index="11" value="0" />
        </Line>
        <Line index="1">
            <Element index="0" value="0" />
            <Element index="1" value="2" />
            <Element index="2" value="0" />
            <Element index="3" value="0" />
            <Element index="4" value="0" />
            <Element index="5" value="0" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="0" />
            <Element index="10" value="0" />
            <Element index="11" value="0" />
        </Line>
        <Line index="2">
            <Element index="0" value="0" />
            <Element index="1" value="0" />
            <Element index="2" value="2" />
            <Element index="3" value="0" />
            <Element index="4" value="0" />
            <Element index="5" value="0" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="0" />
            <Element index="10" value="0" />
            <Element index="11" value="0" />
        </Line>
        <Line index="3">
            <Element index="0" value="0" />
            <Element index="1" value="0" />
            <Element index="2" value="0" />
            <Element index="3" value="2" />
            <Element index="4" value="0" />
            <Element index="5" value="0" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="0" />
            <Element index="10" value="0" />
            <Element index="11" value="0" />
        </Line>
        <Line index="4">
            <Element index="0" value="0" />
            <Element index="1" value="0" />
            <Element index="2" value="0" />
            <Element index="3" value="0" />
            <Element index="4" value="2" />
            <Element index="5" value="0" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
            <Element index="9" value="0" />
            <Element index="10" value="0" />
            <Element index="11" value="0" />
        </Line>
        <Line index="5">
            <Element index="0" value="0" />
            <Element index="1" value="0" />
            <Element index="2" value="0" />
            <Element index="3" value="0" />
            <Element index="4" value="0" />
            <Element index="5" value="2" />
            <Element index="6" value="0" />
            <Element index="7" value="0" />
            <Element index="8" value="0" />
```

```
        <Element index="9" value="0" />
        <Element index="10" value="0" />
        <Element index="11" value="0" />
    </Line>
    <Line index="6">
        <Element index="0" value="0" />
        <Element index="1" value="0" />
        <Element index="2" value="0" />
        <Element index="3" value="0" />
        <Element index="4" value="0" />
        <Element index="5" value="0" />
        <Element index="6" value="2" />
        <Element index="7" value="0" />
        <Element index="8" value="0" />
        <Element index="9" value="0" />
        <Element index="10" value="0" />
        <Element index="11" value="0" />
    </Line>
    <Line index="7">
        <Element index="0" value="0" />
        <Element index="1" value="0" />
        <Element index="2" value="0" />
        <Element index="3" value="0" />
        <Element index="4" value="0" />
        <Element index="5" value="0" />
        <Element index="6" value="0" />
        <Element index="7" value="2" />
        <Element index="8" value="0" />
        <Element index="9" value="0" />
        <Element index="10" value="0" />
        <Element index="11" value="0" />
    </Line>
    <Line index="8">
        <Element index="0" value="0" />
        <Element index="1" value="0" />
        <Element index="2" value="0" />
        <Element index="3" value="0" />
        <Element index="4" value="0" />
        <Element index="5" value="0" />
        <Element index="6" value="0" />
        <Element index="7" value="0" />
        <Element index="8" value="2" />
        <Element index="9" value="0" />
        <Element index="10" value="0" />
        <Element index="11" value="0" />
    </Line>
    <Line index="9">
        <Element index="0" value="0" />
        <Element index="1" value="0" />
        <Element index="2" value="0" />
        <Element index="3" value="0" />
        <Element index="4" value="0" />
        <Element index="5" value="0" />
        <Element index="6" value="0" />
        <Element index="7" value="0" />
        <Element index="8" value="0" />
        <Element index="9" value="2" />
        <Element index="10" value="0" />
        <Element index="11" value="0" />
    </Line>
    <Line index="10">
        <Element index="0" value="0" />
        <Element index="1" value="0" />
        <Element index="2" value="0" />
        <Element index="3" value="0" />
        <Element index="4" value="0" />
        <Element index="5" value="0" />
        <Element index="6" value="0" />
        <Element index="7" value="0" />
        <Element index="8" value="0" />
        <Element index="9" value="0" />
        <Element index="10" value="2" />
        <Element index="11" value="0" />
    </Line>
    <Line index="11">
        <Element index="0" value="0" />
        <Element index="1" value="0" />
        <Element index="2" value="0" />
        <Element index="3" value="0" />
        <Element index="4" value="0" />
        <Element index="5" value="0" />
        <Element index="6" value="0" />
        <Element index="7" value="0" />
        <Element index="8" value="0" />
        <Element index="9" value="0" />
        <Element index="10" value="0" />
        <Element index="11" value="2" />
```

```
            </Line>
        </Matrix>
        <Matrix name="D">
            <Line index="0">
                <Element index="0" value="0" />
            </Line>
            <Line index="1">
                <Element index="0" value="0" />
            </Line>
            <Line index="2">
                <Element index="0" value="0" />
            </Line>
            <Line index="3">
                <Element index="0" value="0" />
            </Line>
            <Line index="4">
                <Element index="0" value="0" />
            </Line>
            <Line index="5">
                <Element index="0" value="0" />
            </Line>
            <Line index="6">
                <Element index="0" value="0" />
            </Line>
            <Line index="7">
                <Element index="0" value="0" />
            </Line>
            <Line index="8">
                <Element index="0" value="0" />
            </Line>
            <Line index="9">
                <Element index="0" value="0" />
            </Line>
            <Line index="10">
                <Element index="0" value="0" />
            </Line>
            <Line index="11">
                <Element index="0" value="0" />
            </Line>
        </Matrix>
    </StateSystem>
</CoreInfo>
```

ANNEXE IV

Modélisation analytique d'un transformateur

ANNEXE IV

MODELISATION ANALYTIQUE D'UN TRANSFORMATEUR

Le modèle de transformateur utilisé au paragraphe IV.3.1. est issu de [PO2] puis a été modifié pour satisfaire aux besoins spécifiques du dimensionnement [FAN]. Ce modèle permet de dimensionner le transformateur suivant :

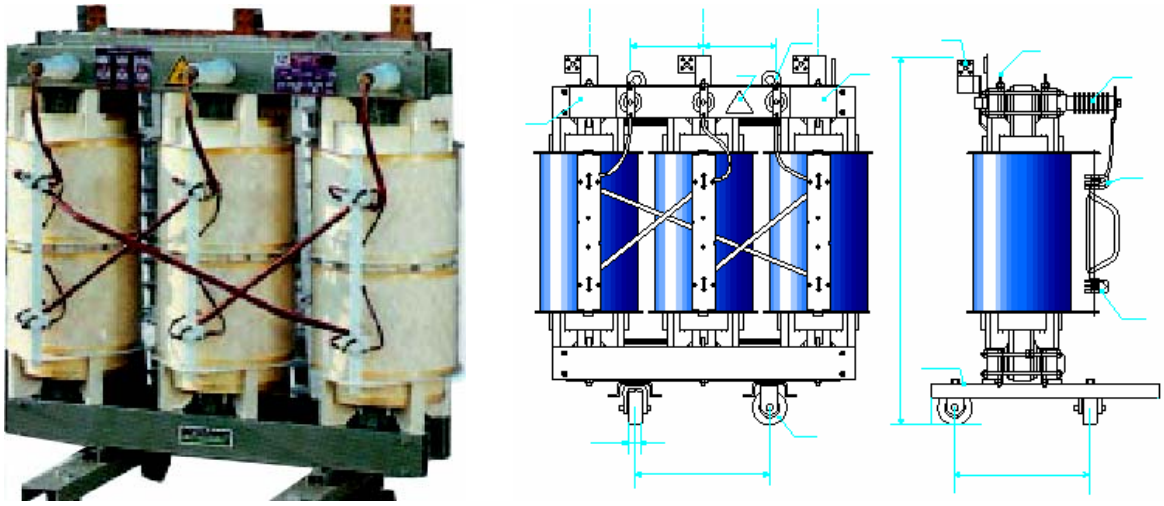


Figure IV.1 : Présentation du transformateur

Le modèle complet du transformateur est crée à partir de quatre sous-modèles :

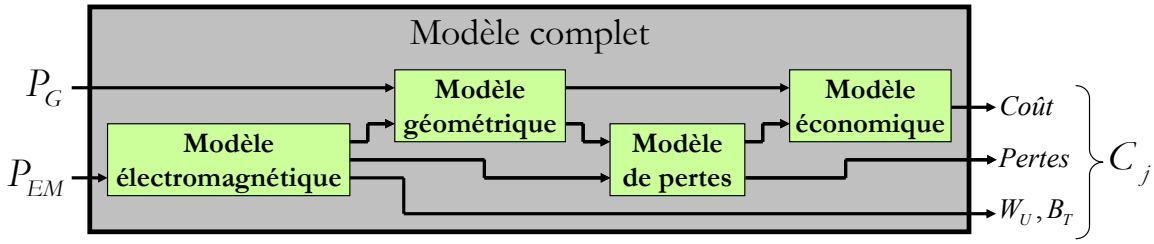


Figure IV.2 : Création du modèle complet

Ces quatre sous-modèles sont des modélisations analytiques s'intéressant à un aspect particulier du transformateur :

- Un modèle calculant les caractéristiques électromagnétiques du transformateur, comme l'induction, les pertes cuivre et fer, les réactances de fuite, …
- Un modèle calculant les paramètres géométriques du transformateur, comme la hauteur totale, le coefficient de remplissage des bobines, le volume total de cuivre, …
- Un modèle calculant les pertes en utilisation du transformateur (pertes cuivre et fer).
- Un modèle calculant le coût total du transformateur (coût des matières premières, coût des pertes capitalisées sur 30 ans).

Le modèle électromagnétique se présente de la manière suivante :

```
/* Modele electromagnetique du transformateur */

intern PI = 3.1415927; // Definition de Pi
intern mu_zero = 4e-7*PI; // Permeabilite du vide
FI=.8; // Coefficient de foisonnement du fer
F1=.7; // Coefficient de remplissage du primaire
F2=.7; // Coefficient de remplissage du secondaire

/* Distances d'isolation des bobines */

D1=.05;
D2=.05;
D3=.05;
D4=.05;
D5=.05;

/* Modele electromagnetique */

S = St/3.0; // Puissance par colonne
V1=U1/sqrt(3.0); // Tension par colonne à partir de la tension composee
A=(N1*S)/(V1*h*F1*J); // Largeur des bobines du primaire
g=(N1*S)/(V1*h*F2*J); // Largeur des bobines du secondaire
ld=sqrt((2.0*sqrt(2.0)*V1)/(pow(PI,2.0)*f*bt*N1*FI)); // Largeur d'une colonne
DM=ld+2.0*D1+2.0*A+D2; // Diametre moyen des bobines
AL=(PI/4.0)*pow(ld,2.0); // Surface d'une colonne
intern FF=(D2+((A+g)/3.0))/h;
X2=2.0*mu_zero*pow(PI,2.0)*DM*pow(N1,2.0)*f*FF; // Inductance de fuite
X2_pu=X2/(pow(V1,2.0)/S); /* Inductance de fuite P.U. */
```

Le modèle géométrique se présente de la manière suivante :

```
/* Modele geometrique du transformateur */

intern PI=3.1415927; // valeur de Pi
intern DC=8900.0; // masse volumique du cuivre
intern DI=7800.0; // masse volumique du fer

/* Calcul du volume et de la masse du fer en fonction des parametres geometriques */

vol_fer = AL*FI*(8.0*(D1+A+D2+g+D5)+6.0*ld+3.0*(h+D4+D3));
mas_fer = DI*vol_fer;

/* Calcul du volume de cuivre */

vol_cu = 3.0*PI*DM*h*(A*F1+g*F2);

/* Calcul de la longueur totale du transformateur */

lon_tot = 4.0*D5+3.0*(ld+2.0*D1+2.0*g+2.0*D2+2.0*A);
```

Le modèle des pertes est le suivant :

```
/* Modele des pertes */

intern rho_cu=2.6e-8;

/* Calcul des pertes fer en fonction de l'induction */

pfkg(bt)=1.996-8.125*bt+12.277*pow(bt,2.0)-7.502*pow(bt,3.0)+1.702*pow(bt,4.0);

Pertes_fer = Masse_fer*pfkg(bt);
Pertes_cu = rho_cu*vol_cu*pow(J,2.0);
```

Enfin, le modèle économique est écrit comme suit :

```
/* Modele economique */

intern pspc=5.0; // Valeur du cout de 1W de pertes cuivre capitalisees sur 30 ans
intern pspf=25.0; // Valeur du cout de 1W de pertes fer capitalisees sur 30 ans
intern PC=25.0; // Prix du cuivre au kg
intern DC=8900.0; // Masse volumique du cuivre
intern PF=12.0; // Prix du fer au kg

Prix_fer = PF*mas_fer;
Prix_cu = PC*DC*vol_cu;
Cout_pf = pspf*Pertes_fer;
Cout_pc = pspc*Pertes_cu;

/* Fonction objectif : cout total du transfo */

Cout_Total = Prix_fer+Prix_cu+Cout_pf+Cout_pc;
```

Les entrées et les sorties du modèle complet sont les suivantes :

Paramètre	E/S	Description
J	E	Densité de courant ($A.m^{-2}$)
bt	E	Densité du flux magnétique (T)
St	E	Puissance apparente totale (VA)
h	E	Hauteur des enroulements (m)
N1	E	Nombre de spires de l'enroulement primaire
f	E	Fréquence du courant (Hz)
U1	E	Amplitude de la tension d'alimentation du primaire (V)
vol_fer	S	Volume de fer total (m^3)
Prix_fer	S	Prix total du fer ($€$)
Prix_cu	S	Prix total du cuivre ($€$)
lon_tot	S	Longueur totale du circuit magnétique (m)
Cout_pf	S	Coût capitalisé sur 30 ans des pertes fer ($€$)
Cout_pc	S	Coût capitalisé sur 30 ans des pertes cuivre ($€$)
Cout_Total	S	Coût capitalisé sur 30 ans du transformateur (fabrication, utilisation) ($€$)

Table IV.1 : Paramètres du modèle complet du transformateur

Une fois composé, le modèle complet du transformateur se présente de la manière suivante :

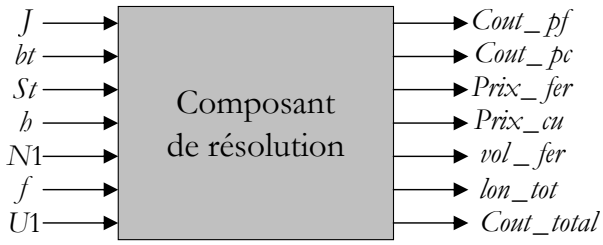


Figure IV.3 : Le modèle complet du transformateur

ANNEXE V

V.1. Utilisation de JavaDiff pour dériver du code **179**
V.1.1. Préparation d'une section de code à dériver 179
V.1.1.1. Déclaration des variables actives 179
V.1.1.2. Délimitation des sections de code actives 180
V.1.1.3. Sélection des variables dépendantes et indépendantes 180
V.1.1.4. Opérateurs et fonctions mathématiques 181
V.1.2. Procédure de modification d'un code de calcul 182
V.1.2.1. Balisage des sections de code 183
V.1.2.2. Spécification des variables actives 183
V.1.2.3. Exploitation des résultats 184
V.2. Fonctionnement de JavaDiff **185**

ANNEXE V

DERIVATEUR DE CODE JAVA : JAVADIFF

Afin de pouvoir tester les différentes approches de la dérivation de code étudiées au cours de ces travaux, nous avons développé un outil de dérivation de code Java : JavaDiff. Nous allons ici détailler le fonctionnement ainsi que l'utilisation de JavaDiff.

V.1. Utilisation de JavaDiff pour dériver du code

JavaDiff permet l'évaluation des dérivées du premier ordre de fonctions représentées par des codes Java, dans les deux modes de différentiation, direct et inverse. JavaDiff n'est pas à proprement parler un dérivateur de code, puisqu'il nécessite une adaptation du code à dériver. Cependant, il a été conçu afin que les changements à apporter au code soient minimaux.
Dans une section de code à dériver, trois types principaux de variables peuvent être rencontrés. Tout d'abord, on peut rencontrer des variables non différentiables, considérées comme constantes vis-à-vis de la dérivation du code, mais qui peuvent être variables pour le reste du programme. On peut aussi rencontrer des variables différentiables qui sont des paramètres d'entrée du code à dériver. Nous appèlerons ces variables les variables indépendantes. Enfin, les autres variables différentiables seront appelées variables dépendantes.

V.1.1. Préparation d'une section de code à dériver

JavaDiff a été conçu afin que les changements à apporter au code à dériver soient minimaux. Ces changements concernent principalement la déclaration des variables, les opérations d'entrées sorties et l'appel aux fonctions mathématiques.

V.1.1.1. Déclaration des variables actives

Le point clé du fonctionnement de JavaDiff est le concept de variable active. Toutes les variables qui peuvent être considérées, à un moment ou un autre de l'exécution du code à dériver, comme des quantités différentiables, doivent être déclarées comme des variables actives. JavaDiff utilise comme variables actives les variables de la classe `cdi.javadiff.ADouble`, dont la partie standard réelle est du type double. Typiquement, on déclarera actives les variables indépendantes ainsi que toutes les quantités qui en dépendent, directement ou indirectement. D'autres variables, ne dépendant pas des variables indépendantes mais intervenant comme des paramètres (les constantes), peuvent rester de type passif, comme `java.lang.Double`, `double`, `float` ou `int`.

Il n'y a pas de conversion implicite de type entre `ADouble` et les types passifs. Le fait de ne pas déclarer une variable comme active lorsqu'elle dépend d'autres variables actives conduira à une erreur de compilation. On peut dire que la liste des variables actives contient ses successeurs dans le graphe de dépendance.

La composante réelle d'une variable active `ADouble var` peut être récupérée par `var.x`. En particulier, cette récupération peut être nécessaire dans le cas de code utilisant la sortie système standard `System.out.println`.

Les `ADoubles` peuvent naturellement être les composantes de vecteurs ou de matrices.

V.1.1.2. Délimitation des sections de code actives

Tous les calculs impliquant des variables actives et devant être dérivés doivent être marqués en tant que sections actives. Pour cela, on utilisera les fonctions de la classe `cdi.javadiff.DifferentiationManager`.

Le début d'une section active peut être marquée de deux manières différentes, suivant que l'on veut dériver le code en utilisant le mode direct ou le mode inverse de différentiation. En mode direct, on utilisera la fonction `DifferentiationManager.activate()`. En mode inverse, on prendra la fonction `DifferentiationManager.activateTrace()`.

Dans les deux cas, la fin d'une section active est marquée par l'appel à la fonction `DifferentiationManager.desactivate()`.

Les sections actives peuvent contenir des appels à des fonctions ou à des codes récursifs, fournis par l'utilisateur. Naturellement, leurs paramètres doivent correspondre avec le code actif. En particulier, cela signifie que les fonctions doivent être compilées avec leurs variables actives déclarées en tant que `ADouble`.

Des variables du type `ADouble` peuvent être déclarées hors d'une section active (comme dans le cas d'une fonction par exemple), et ne nécessitent aucun traitement particulier.

V.1.1.3. Sélection des variables dépendantes et indépendantes

Pour initialiser une variable active à une valeur *x*, on utilisera le constructeur suivant :

```
ADouble var = new ADouble(x);
```

On s'assure ainsi du fait que `var.x = x`.

Une ou plusieurs variables actives qui sont initialisés avec les valeurs de constantes ou de variables passives doivent être distinguées en tant que variables indépendantes. Les autres variables actives seront considérées comme variables dépendantes. Pour faire ce distinguo, on n'utilisera pas un constructeur différent. Seule l'endroit de la déclaration des variables distingue les variables indépendantes et dépendantes.

En effet, le programme contenant le code à différentier doit avoir la structure suivante :

```
      code …
DifferentiationManager manager = new DifferentiationManager();
      initialisation des variables indépendantes
manager.activate();
ou
manager.activateTrace();
      code de calcul à dériver
manager.desactivate();
      récupération des valeurs des dérivées
```

Quand les variables sont construites entre la création du `DifferentiationManager` et son activation, elles sont alors considérées comme variables indépendantes. Quand elles sont construites dans une section active du code (entre les balises `activate` et `desactivate`), elles sont considérées comme dépendantes. Seul le moment de leur création détermine leur statut.
De plus, leur ordre de création est crucial, particulièrement pour les variables indépendantes. Leur ordre de création détermine l'ordre dans lequel les dérivées partielles sont calculées et donc l'indice permettant de récupérer les dérivées partielles relatives à telle ou telle variable indépendante.

V.1.1.4. Opérateurs et fonctions mathématiques

Les opérateurs et fonctions mathématiques ont étés redéfinis pour fonctionner avec les variables actives. Malheureusement, il n'est pas possible en Java d'utiliser la technique de surcharge d'opérateurs comme en C / C++. On ne peut donc pas utiliser directement les opérateurs +,-,*,/,…avec des variables actives. Lors de la modification du code en vue d'une dérivation, il faudra donc remplacer les opérateurs par les fonctions suivantes :

$a+b$	`sum(a,b)`
$a-b$	`substract(a,b)`
$a*b$	`product(a,b)`
a/b	`divide(a,b)`

Table V.1 : Adaptation des opérateurs mathématiques

Cette opération n'est évidemment pas anodine et impose des changements non négligeables au code de calcul. Deux solutions peuvent être utilisées pour pallier à ce problème. Tout d'abord, dans une optique de génération de code (comme nous le faisons dans ces travaux), nous utilisons

RAMA[1] pour transformer le code des équations et l'adapter aux `ADoubles`. La deuxième solution, adaptée pour un code déjà existant que l'on veut dériver, est d'utiliser un préprocesseur qui effectuera les modifications adéquates avant la compilation. Un tel préprocesseur existe déjà : SubJava [SUB].

Tout comme on doit adapter les opérateurs aux variables actives, on doit aussi adapter les fonctions. Ici l'adaptation sera plus facile, mais néanmoins fastidieuse, et c'est pourquoi nous nous appuyons sur RAMA pour l'effectuer. De même, SubJava peut également être utilisé. Les fonctions doivent être remplacées comme indiqué sur la table suivante :

Fonction Java	Fonction JavaDiff
java.lang.Math.abs(a)	cdi.javadiff.AMath.abs(a)
java.lang.Math.acos(a)	cdi.javadiff.AMath.acos(a)
java.lang.Math.asin(a)	cdi.javadiff.AMath.asin(a)
java.lang.Math.atan(a)	cdi.javadiff.AMath.atan(a)
java.lang.Math.cos(a)	cdi.javadiff.AMath.cos(a)
java.lang.Math.exp(a)	cdi.javadiff.AMath.exp(a)
java.lang.Math.log(a)	cdi.javadiff.AMath.log(a)
java.lang.Math.max(a,b)	cdi.javadiff.AMath.max(a,b)
java.lang.Math.min(a,b)	cdi.javadiff.AMath.min(a,b)
java.lang.Math.pow(a,b)	cdi.javadiff.AMath.pow(a,b)
-	cdi.javadiff.AMath.sign(a)
java.lang.Math.sqrt(a)	cdi.javadiff.AMath.sqrt(a)
java.lang.Math.tan(a)	cdi.javadiff.AMath.tan(a)

Table V.5 : Adaptation des fonction mathématiques

V.1.2. Procédure de modification d'un code de calcul

Afin d'illustrer la procédure de modification d'un code de calcul, nous allons maintenant la détailler pas à pas. Partons d'un code de calcul écrit normalement en Java :

```
public static void main(String[] args)
{
      double rau = 2, l = 3, s = .5, I = 5;
      double R = rau*l/s;
      double U = R*I;
      double P = U*I;
      System.out.println("Puissance dissipee : "+P);
}
```

[1] RAMA : Rule Applicator for Mathematical Analysis. Voir Annexe I.

V.1.2.1. Balisage des sections de code

On peut voir ici trois parties distinctes dans le code, l'initialisation des paramètres, le calcul proprement dit (ce sera la section active du code) et enfin l'exploitation des résultats. Afin de pouvoir dériver ce code, nous allons commencer par placer les balises de marquage des différentes parties.

On aboutit au code suivant :

```
public static void main(String[] args)
{
      DifferentiationManager manager = new DifferentiationManager();
      double rau = 2, l = 3, s = .5, I = 5;
      manager.activate();
      double R = rau*l/s;
      double U = R*I;
      double P = U*I;
      manager.desactivate();
      System.out.println("Puissance dissipee : "+P);
}
```

La construction d'une instance de `DifferentiationManager` indique à `JavaDiff` que les instructions suivantes seront les déclarations des variables indépendantes du code à dériver, jusqu'à ce que ce manager soit activé. A partir de ce moment là, on entre dans la partie active du code (le calcul proprement dit).

Pour activer le manager, deux fonctions peuvent être utilisées. Le choix d'une de ces deux fonctions conditionne le mode de différentiation utilisé :

- La fonction `activate()` lance le mode direct de différentiation
- La fonction `activateTrace()` lance le mode inverse de différentiation

Quelque soit le mode de différentiation utilisé, la fonction `desactivate()` indique la fin de la section active du code.

V.1.2.2. Spécification des variables actives

Il faut ensuite transformer les variables passives en variables actives, et subséquemment les expressions mathématiques correspondantes :

```
public static void main(String[] args)
{
      DifferentiationManager manager = new DifferentiationManager();
      Adouble rau = new ADouble(2);
      ADouble l = new ADouble(3);
      ADouble s = new ADouble(.5);
      ADouble I = new ADouble(5);
```

```
        manager.activate();
        ADouble R = AMath.product(rau,AMath.divide(l,s));
        ADouble U = AMath.product(R,I);
        ADouble P = AMath.product(U,I);
        manager.desactivate();
        System.out.println("Puissance dissipee : "+P.x);
}
```

V.1.2.3. Exploitation des résultats

Il ne reste ensuite qu'à exploiter les résultats. En mode direct, la récupération des dérivées partielles se fait directement grâce au champ `dx` de chaque variable active. Si le code calcule la valeur de la variable active `a`, alors on utilisera l'instruction suivante pour récupérer la $i^{ème}$ dérivée partielle (par rapport à l'ordre dans lequel elles ont été enregistrées lors de leur déclaration) :

```
a.dx[i]
```

En mode inverse, avant de récupérer les valeurs des dérivées partielles, il faut lancer le calcul qui permettra de les calculer. Pour cela, on utilise les fonctions suivantes :

- `public double[][] computeJacobian()` : cette fonction calcule le jacobien de la section de code active.
- `public void computeGrad(Adouble output)` : cette fonction calcule le gradient d'une sortie donnée par rapport aux entrées.
- `public double getGrad(Adouble intput)` : cette fonction permet de récupérer la valeur d'un élément du gradient calculé précédemment par la méthode `computeGrad`.

Au final, le code complet en mode direct se présente donc de la manière suivante :

```
public static void main(String[] args)
{
        DifferentiationManager manager = new DifferentiationManager();
        Adouble rau = new ADouble(2);
        ADouble l = new ADouble(3);
        ADouble s = new ADouble(.5);
        ADouble I = new ADouble(5);
        manager.activate();
        ADouble R = AMath.product(rau,AMath.divide(l,s));
        ADouble U = AMath.product(R,I);
        ADouble P = AMath.product(U,I);
        manager.desactivate();
        System.out.println("Puissance dissipee : "+P.x);
        System.out.println("dP/dRau = "+p.dx[0]);
        System.out.println("dP/dl = "+p.dx[1]);
        System.out.println("dP/ds = "+p.dx[2]);
        System.out.println("dP/dI = "+p.dx[3]);
}
```

En mode inverse, le code est légèrement différent :

```
public static void main(String[] args)
{
	DifferentiationManager manager = new DifferentiationManager();
	ADouble rau = new ADouble(2);
	ADouble l = new ADouble(3);
	ADouble s = new ADouble(.5);
	ADouble I = new ADouble(5);
	manager.activateTrace();
	ADouble R = AMath.product(rau,AMath.divide(l,s));
	ADouble U = AMath.product(R,I);
	ADouble P = AMath.product(U,I);
	manager.desactivate();
	System.out.println("Puissance dissipee : "+P.x);
	double[][] jacobian = manager.computeJacobian();
	System.out.println("dP/dRau = "+jacobian[2][0]);
	System.out.println("dP/dl = "+jacobian[2][1]);
	System.out.println("dP/ds = "+jacobian[2][2]);
	System.out.println("dP/dI = "+jacobian[2][3]);
	System.out.println("dP/dI = "+jacobian[1][3]);
}
```

V.2. Fonctionnement de JavaDiff

Le fonctionnement de JavaDiff est basé sur l'interaction entre trois classes :

- `cdi.javadiff.ADouble`, définissant une variable active dans le cadre de la différentiation
- `cdi.javadiff.AMath`, définissant les opérations mathématiques entre variables actives
- `cdi.javadiff.DifferentiationManager`, le gestionnaire de différentiation

Lors de l'exécution d'un code utilisant JavaDiff, deux modes de fonctionnement distincts existent, le mode direct et le mode inverse.

En mode direct, les gradients de chaque variable active sont calculés en même temps que la valeur de la variable est calculée. Ce mode de fonctionnement, bien que moins efficace qu'un calcul en mode direct basé sur l'analyse du graphe de Kantorovitch de la fonction à dériver, permet une exploitation directe et sans instructions supplémentaires de la différentiation. C'est pour cela que nous avons choisi ce mode direct de différentiation.

En mode inverse, un graphe de calcul est établi lors de l'exécution du code actif. Ce graphe de calcul contient des nœuds correspondant aux différentes variables actives entrant en jeu dans le code à différentier, et des connections, reliant un nœud à un autre et représentant le lien paramètre-résultat correspondant à l'utilisation d'un fonction sur une variable active. Lors de

l'évaluation des dérivées, ce graphe est parcouru par l'algorithme de différentiation et le jacobien de la fonction évalué. Cette approche est l'approche par le graphe de calcul. Les deux autres approches étudiées (approche par dualité, et par substitutions régressives) ne sont pas disponibles en tant que telles dans JavaDiff. En effet, les différences minimes entres les temps de calcul des différentes approches ne justifie pas leur mise à disposition (et donc la complexification de l'utilisation de JavaDiff).

ANNEXE VI

VI.1. Formalisme de description des modèles analytiques **189**
VI.1.1. Description de l'exemple 189
VI.1.2. Formalisme de description 190
VI.1.2.1. Equations simples 190
VI.1.2.2. Ajout de commentaires 191
VI.1.2.3. Statut des variables 191
VI.1.2.4. Utilisation de fonctions 191
VI.1.3. Modèle du dispositif 193
VI.2. Les différents générateurs de composants **194**
VI.2.1. Générateurs JavaDiff Direct Jacobian et JavaDiff Reverse Jacobian 194
VI.2.2. Générateur RAMA Differentials 194
VI.2.3. Générateur Adol-C Jacobian 195

ANNEXE VI

GENERATION DES COMPOSANTS DE RESOLUTION

Au cours de ces travaux de thèse, nous avons été amenés à développer et utiliser plusieurs générateurs de composants de résolution à partir de modèles analytiques. Nous allons ici présenter le formalisme de modélisation analytique ainsi que les différents générateurs prenant en compte ce formalisme.

VI.1. Formalisme de description des modèles analytiques

VI.1.1. Description de l'exemple

Les générateurs de composants de résolution s'appuient sur un formalisme particulier de description des modèles analytiques. Nous allons ici présenter ce formalisme en nous appuyant sur l'exemple suivant :

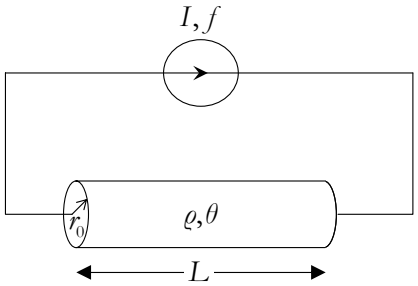


Figure VI.1 : Le dispositif à modéliser

Ce dispositif est constitué d'un fil de cuivre à une température θ, de longueur L et de rayon r_0, de résistivité $\varrho_0 = 1{,}69.10^{-5}\,\Omega m$ à 20°C, traversé par un courant I de fréquence f. La résistivité du fil à une température θ est donnée par la relation suivante :

$$\varrho = \varrho_0\left(1 + \theta - \theta_0\right) \tag{1}$$

Le courant traversant le fil étant de fréquence f, l'épaisseur caractéristique de l'effet de peau est donnée par :

$$e = \sqrt{\frac{\varrho}{\pi\,\mu_0\,f}} \tag{2}$$

En effet, le cuivre étant amagnétique on prend $\mu_r = 1$.

La résistance du fil de cuivre est donc donnée par la formule suivante :

$$R = \varrho \frac{L}{S} \tag{3}$$

En tenant compte de l'effet de peau, S est donné par :

$$S = \pi\left(e^2 - 2r_0 e\right) \tag{4}$$

La puissance dissipée par effet Joule dans le fil est finalement donnée par :

$$P_J = \frac{RI^2}{2} \tag{5}$$

Le but est d'obtenir le composant de résolution suivant :

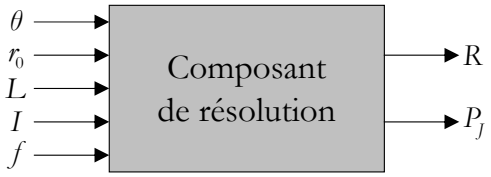


Figure VI.2 : Le composant à obtenir

VI.1.2. Formalisme de description

VI.1.2.1. Equations simples

Comme on l'a vu, les entrées et les sorties du composant de résolution sont liées par un jeu d'équations. A partir de ce jeu d'équation, les différents générateurs sont capables de générer le composant de résolution de ce modèle analytique. Un modèle analytique est donc majoritairement composé d'équations.

Le fichier contenant ce modèle est un fichier texte standard. L'élément de base d'un modèle analytique est l'équation simple, du type :

variable = equation

Les équations du modèle doivent être entrées au format standard ANSI-C. Les équations dans un modèle ne sont pas ordonnées. Cela signifie qu'on peut utiliser une variable avant de l'avoir calculée, ce qui n'est pas possible par exemple dans un code informatique. Cela permet au concepteur de se concentrer sur la modélisation, et non sur la calculabilité en terme informatique du modèle. Chaque équation doit se terminer par un point-virgule. Une variable ne doit être calculée qu'une seule fois (pas d'accumulation possible).

Dans le cas de notre exemple, le modèle est donc le suivant :

```
rau = rau0*(1+t-t0) ;
e = sqrt(rau/(Pi*mu0*f)) ;
Pi = 3.1415927;
mu0 = Pi*4e-7;
rau0 = 1.69e-5;
```

```
R = rau*L/S;
S = Pi*(pow(e,2)-2*r0*e);
Pj = R*pow(I,2)/2;
```

Ce modèle est valide pour nos générateurs. Néanmoins, plusieurs éléments peuvent venir s'ajouter sur ce modèle de base afin d'affiner le composant obtenu, ou de faciliter l'écriture du modèle.

VI.1.2.2. Ajout de commentaires

On peut ajouter des commentaires au modèle, non pris en compte lors de la génération. Ces commentaires peuvent être de deux types, similaires à ceux que l'on peut rencontrer en programmation :

- Situé entre `/*` et `*/`
- A la fin d'une ligne, après `//` et jusqu'à la fin de la ligne

VI.1.2.3. Statut des variables

Dans un modèle analytique, les variables peuvent être de trois types :

- des entrées
- des sorties
- des variables intermédiaires

Dans le cas de l'équation *variable* = *equation* , la variable calculée est par défaut considérée comme une sortie, et tous les paramètres de l'équation comme des entrées (à moins que ce ne soit une sortie d'une autre équation, auquel cas la variable est considérée comme une sortie). Dans le cas du modèle écrit au paragraphe précédent, toutes les variables sont soit des entrées (c'est le cas par exemple de `t` et `t0`), soit des sorties (comme pour `R` ou `mu0`). Evidemment, `mu0` ne nous intéresse pas comme sortie du composant. Il faut donc la déclarer en tant que variable intermédiaire (donc n'apparaissant pas comme sortie du composant). Pour cela, on dispose du mot clé `intern`. Ce mot clé peut s'utiliser de deux manières différentes :

- Utilisé indépendamment, à tout endroit du modèle : `intern mu0 ;`
  On peut ainsi déclarer une liste de variables internes : `intern mu0,Pi,rau0 ;`
- On peut aussi l'utiliser au moment du calcul de la variable : `intern mu0 = Pi*4e-7 ;`

VI.1.2.4. Utilisation de fonctions

Trois types de fonctions peuvent être utilisées dans les modèles analytiques. Les fonctions mathématiques standard peuvent être utilisées, les fonctions rentrées par l'utilisateur, et les fonctions extérieures.

Les fonctions mathématiques standard sont les suivantes :

Fonction	Description
abs(a)	Valeur absolue
acos(a)	Arccosinus
acosh(a)	Arccosinus hyperbolique
asin(a)	Arcsinus
asinh(a)	Arcsinus hyperbolique
atan(a)	Arctangente
atanh(a)	Arctangente hyperbolique
cos(a)	Cosinus
cosh(a)	Cosinus hyperbolique
exp(a)	Exponentielle
log(a)	Logarithme népérien
max(a,b)	Maximum
min(a,b)	Minimum
pow(a,b)	Puissance
sign(a)	Signe
sin(a)	Sinus
sinh(a)	Sinus hyperbolique
sqrt(a)	Racine carrée
tan(a)	Tangente
tanh(a)	Tangente hyperbolique

Le concepteur peut aussi déclarer ses propres fonctions et les utiliser dans son modèle. Les fonctions peuvent être déclarées de deux manières différentes :

- la manière simple : `f(x) = 2*x+3;`
- la manière étendue :

```
f(x)
{
   a = 2*x;
   return a+3;
}
```

Ces déclarations de fonctions sont parfaitement équivalentes. Elles peuvent être faite à n'importe quel endroit du modèle.

L'utilisation de ces fonctions engendre une catégorie de variables particulières : les variables globales. Ce sont des variables appartenant au modèle et utilisée dans le corps d'une fonction :

```
rau0 = 1.69e-5 ;
rau(t) = rau0*(1+t-20) ;
```

Ici, la variable `rau0` est déclarée dans le modèle et utilisée dans le corps de la fonction. C'est une variable globale. Il n'est pas nécessaire de faire une déclaration particulière pour ces variables. Le générateur les détecte et signale leur présence lors de la génération. Néanmoins, l'emploi de ces variables peut se révéler délicat. En effet, la valeur de cette variable peut changer lors de l'évaluation du modèle, et comme le modèle n'est pas ordonné (le générateur réordonnance les calcul afin d'assurer la calculabilité du modèle). C'est pourquoi nous conseillons de restreindre l'emploi de ce genre de variables aux constantes du modèle.

Enfin, le modèle peut utiliser des fonctions extérieures. Ces fonctions sont programmées de manière indépendante et peuvent être utilisées exactement comme les autres fonctions dans les modèles. Ces fonctions extérieures permettent de faire appel à des codes élaborés (algorithmes numériques, éléments finis, simulations numériques) au sein des modèles analytiques. La manière de programmer les fonctions extérieures est spécifique suivant les différents générateurs, et sera détaillée au paragraphe VI.2.

VI.1.3. Modèle du dispositif

Le modèle du dispositif présenté sur la figure 1 peut donc se présenter de la manière suivante :

```
/* Modele des pertes Joules dans un fil de cuivre.
* Fichier : fil_cuivre.cam
*/

/* Constantes physiques */

intern Pi = 3.1415927 ;  // Valeur de Pi
intern mu0 = Pi*4e-7 ;   // Permeabilite du vide
intern rau0 = 1.69e-5 ;  // Resistivite du cuivre a 20°
intern t0 = 20 ;         // Temperature de reference pour la resistivite du cuivre

/* Calcul de la resistivite du cuivre en fonction de la temperature */

rau(t) = rau0*(1+t-t0) ;

/* Calcul de la surface utile du fil en tenant compte de l'effet de peau */

S(f,r0,t)
{
  e = sqrt(rau(t)/(Pi*mu0*f)) ;
  return Pi*(pow(e,2)-2*r0*e);
}

/* Equations du modele.
* Calcul des sorties.
*/

R = rau(t)*L/S(f,r0,t);
Pj = R*pow(I,2)/2;
```

VI.2. Les différents générateurs de composants

VI.2.1. Générateurs JavaDiff Direct Jacobian et JavaDiff Reverse Jacobian

Ces deux générateurs créent des composants de résolution de modèles analytiques qui utilisent JavaDiff (voir Annexe V) pour calculer le jacobien des modèles. Ces deux générateurs prennent en entrée des modèles analytiques au format présenté précédemment.

Les fonctions extérieures utilisées dans le modèle doivent être programmée en Java (ce code Java pouvant faire appel à des codes programmés dans d'autres langages). Dans l'exemple du fil, on pourrait par exemple faire appel à une fonction extérieure pour calculer l'épaisseur de peau :

```
e = Epaisseur(rau,f) ;
```

Le code de cette fonction `Epaisseur` doit être encapsulé dans une classe de même nom, avec libre choix du package, par exemple `exemple.filcuivre.Epaisseur`. Cette classe doit comporter deux méthodes, `public double compute` et `public double[][] differentiate`, avec le même nombre de paramètres que dans le modèle, et dans le même ordre. La fonction `compute` renvoie la valeur de la fonction, et la fonction `differentiate` son jacobien.

La classe Epaisseur se présente donc de la manière suivante :

```
/* Fichier Epaisseur.java */

package exemple.filcuivre ;

public class Epaisseur
{
  public double compute(double rau,double f)
  {
    … code de calcul
  }

  public double[][] differentiate(double rau,double f)
  {
    … code de calcul
  }
}
```

VI.2.2. Générateur RAMA Differentials

Ce générateur crée des composants de résolution de modèles analytiques basés sur RAMA pour le calcul des différentielles des sorites du modèle. Les fonctions extérieures doivent être dans une classe de même nom que dans le modèle, exactement comme pour les générateurs JavaDiff. La fonction compute est définie de manière identique à celle des générateurs JavaDiff. La seule différence se situe au niveau de la fonction différentiate, qui renvoie la différentielle de la valeur

de la fonction en fonction de la valeur de ses paramètres et de leurs différentielles. La fonction différentiate devient donc :

```
public double differentiate(double rau,double d_rau,double f,double d_f)
{
  … code de calcul
}
```

VI.2.3. Générateur Adol-C Jacobian

Ce générateur crée des composants de résolution de modèles analytiques basés sur Adol-C pour le calcul du jacobien du modèle. A cause de l'emploi d'Adol-C, ce générateur ne permet pas l'utilisation de fonctions extérieures.

INDEX

CHAPITRE I : CADRE DE TRAVAIL **13**

Figure I.1 : Principe du dimensionnement **16**
Figure I.2 : Le modèle de dimensionnement, avec ses entrées P_i et ses sorties C_i **17**
Figure I.3 : Approche par génération de code **18**
Figure I.4 : Le fonctionnement général de l'optimisation **19**
Figure I.5 : Le circuit électrique à modéliser **23**
Figure I.6 : Représentation générale du système **24**
Figure I.7 : Processus lié à l'approche composant **27**
Figure I.8 : Représentation du composant logiciel **28**
Figure I.9 : Différence entre boite blanche et boite noire **30**
Figure I.10 : Objets et composants **30**
Figure I.11 : Structure interne d'un composant **32**
Figure I.12 : Principe de la projection **32**
Figure I.13 : Le modèle composé du transformateur **33**

CHAPITRE II : LA RESOLUTION DES MODELES DE DIMENSIONNEMENT **37**

Figure II.1 : La "bosse" de l'exponentielle **45**
Table II.1 : Paires optimales *(q,j)* pour différentes valeurs de précision et de norme **49**
Figure II.2 : Graphe de Kantorovitch d'une fonction **57**
Figure II.3 : Du code au graphe de calcul **57**
Figure II.4 : Fonctionnement du mode direct **58**
Figure II.5 : Application du mode direct **59**
Figure II.6 : Graphe de calcul pondéré **60**
Figure II.7 : Un graphe de calcul particulier **62**
Figure II.8 : Repondération du graphe de calcul **62**
Figure II.9 : Autre graphe de calcul particulier **63**
Figure II.10 : Application de l'approche par le graphe de calcul **64**
Figure II.11 : Ajout des fonctions ψ_i sur le graphe de calcul **64**
Figure II.12 : Algorithme ReG de calcul des dérivées en mode inverse **68**
Figure II.13 : Algorithme ReS de calcul des dérivées en mode inverse **69**

CHAPITRE III : APPROCHE COMPOSANT POUR LE DIMENSIONNEMENT **71**

Figure III.1 : Interface du composant de résolution des modèles **75**
Figure III.2 : Interface du composant d'optimisation **75**
Figure III.3 : Différence entre polymorphisme et schizomorphisme **78**
Figure III.4 : Représentation de l'interface `Component` **80**
Figure III.5 : Représentation de l'interface `Service` **81**
Figure III.6 : Principe du mécanisme d'aiguillage **82**

Figure III.7 : Représentation du fichier contenant le composant 83
Figure III.8 : L'outil de création des composants 84
Figure III.9 : Le chargeur des composants 84
Figure III.10 : Le service de procédure de chargement avancé 85
Figure III.11 : L'introspecteur de composants 85
Figure III.12 : Le composant de résolution 86
Figure III.13 : Interface associée au composant `CoRe` 86
Figure III.14 : Interface associée au service `CoRe` dans le cadre d'ICAr 87
Figure III.15 : Service de calcul des modèles 87
Figure III.16 : Interface associée au service de résolution des modèles 87
Figure III.17 : Service de calcul du modèle avec calcul du Jacobien 88
Figure III.18 : Interface associée 88
Figure III.19 : Service de calcul du modèle avec approche différentielle 88
Figure III.20 : Interface associée 88
Figure III.21 : Le service de visualisation 89
Figure III.22 : Représentation du service de gestion des algorithmes 90
Figure III.23 : Interface du service `AlgorithmManager` 90
Figure III.24 : Création et utilisation des composants 93
Figure III.25 : Architecture générale de CoreLab 94
Figure III.26 : Principe de fonctionnement du module de génération 96
Figure III.27 : Architecture du module de génération 96
Figure III.28 : Processus de génération des générateurs de CoreLab 97
Figure III.29 : Structure du composant généré 99
Figure III.30 : Synthèse de l'utilisation du module de génération 99
Figure III.31 : Composition de modèles simples au format `ModelSolver` 100
Figure III.32 : Composition de modèles au format `DifferentialSolver` 100
Figure III.33 : Composition hétérogène 101
Figure III.34 : Structure d'un composant résultant d'une composition 102
Figure III.35 : Architecture du module de composition 103
Figure III.36 : Projection de `JacobianSolver` vers `DifferentialSolver` 103

CHAPITRE IV : APPLICATIONS 105

Figure IV.1 : Interface de CoreLab 107
Figure IV.2 : Le convertisseur statique modélisé et son filtre RSIL 109
Figure IV.3 : Génération d'un composant de résolution d'équations différentielles 111
Figure IV.4 : Interface de la calculette pour composants de simulation de CoreLab 112
Figure IV.5 : Résultats de la simulation sous Simplorer 113
Figure IV.6 : Résultats de la simulation avec CoreLab 113
Table IV.1 : Comparaison des performances de résolution entre Simplorer et CoreLab 113
Figure IV.7 : Vue de l'alternateur à griffes 114

Figure IV.8 : Génération du composant de calcul dans CoreLab 116
Figure IV.9 : Spécification du cahier des charges dans CDIOptimizer 117
Figure IV.10 : Exploitation des résultats de l'optimisation 118
Figure IV.11 : Résultats de l'optimisation : cartographie du rendement de l'alternateur 118
Figure IV.12 : Présentation du transformateur 119
Figure IV.13 : Composition des sous-modèles définissant le modèle complet 120
Figure IV.14 : Composition du modèle complet du transformateur sous CoreLab 120
Figure IV.15 : Utilisation d'un composant de résolution contenant une visualisation de géométrie 121
Figure IV.16 : Représentation de l'actionneur linéaire 122
Figure IV.17 : Modélisation de l'actionneur linéaire 122
Table IV.2 : Temps d'optimisation et nombre d'itérations pour les différents calculs de dérivées 123
Table IV.3 : Comparaison des performances des différentes manières de calculer les dérivées 125
Figure IV.18 : Code de calcul et fonction correspondante 125

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[ALB] L. Albert, "Modélisation et optimisation d'un alternateur à griffes. Applications au domaine automobile.", Thèse de Doctorat de l'INPG, 13 Juillet 2004.

[ALL] L. Allain, "Capitalisation et traitement des modèles pour la conception en Génie Electrique", Thèse de Doctorat de l'INPG, 30 Septembre 2003.

[ARN] ARNO, un environnement d'optimisation des réseaux de téléphonie mobile, http://www.ec-nantes.fr/ectia/ARNO

[ATI] E. Atienza, "Méthodes et outils contre le dimensionnement", Thèse de Doctorat de l'INPG, 4 Juillet 2003

[BEA] JavaBeans : http://java.sun.com/products/javabeans/

[BLA] A. Blanguernon, "Evaluation de diodes en carbure de silicium dans une alimentation PC type PFC", Mémoire de maîtrise IUP UJF, 2 Septembre 2002

[CHA] P. Chapouille, "Maintenabilité. Maintenance.", Techniques de l'Ingénieur réf. T4305, 2001

[CHI] C. Chillet, J.Y. Voyant, "Design-oriented analytical study of a Linear Electromagnetic Actuator by means of reluctance network", IEEE Transactions on Magnetics, vol. 37, n° 4, pp 3004 – 3011, 2001

[CHE] H. Chetouani, "Choix de structures et outils pour les microsystèmes magnétiques", Master Recherche INPG, 24 Juin 2004.

[COM] Microsoft COM : http://www.microsoft.com/com/

[COR] Corba : http://www.corba.org

[DAZ] J. D'Azzo, C.H. Houpis, "Linear control system analysis and design, 4th Edition", McGraw-Hill Series in Electrical and Computer Engineering, 1995, ISBN 0-07-016321-9

[DEL] B. Delinchant, "Un environnement à base de composants, intégrant le concepteur et ses outils, pour de nouvelles méthodes de CAO", Thèse de Doctorat de l'INPG, 10 juillet 2003

[DEL2] B. Delinchant, F. Wurtz, E. Atienza, J. Bigeon, "Benefits of component methodology applied to electrical software", Electrimacs'02, 7th International Conference on Modelling and Simulation of Electrical Machines, Converters and Systems, Montréal, Quebec, August 18-21, 2002

[DEZ] E. Dezille, "Utilisation de la différentiation de code dans le cadre du dimensionnement sous contraintes", Master Recherche INPG, 29 Juin 2004.

[FAN] J. Fandino, F. Wurtz, J. Bigeon, "Nouvelle méthododologie de conception des dispositifs électriques", Revue Internationale de Génie Electrique, Vol. 2 n°1, 1999

[FIS] V. Fischer, L. Gerbaud, "Solving ODE for optimization : specific use of the matrix approach", Optimization and Inverse Problems in Electromagnetism, M. Rudnicki, S. Wiak Eds, pp 71 – 78, ISBN 1-4020-1506-2

[FI2] V. Fischer, L. Gerbaud, F. Wurtz, "Using automatic code differentiation for optimization", IEEE CEFC 2004, Seoul, Korea, June 6-9, 2004

[FI3] V. Fischer, B. Delinchant, F. Wurtz, L. Gerbaud, "Using software components for multi-disciplinary optimization", IEEE CEFC 2004, Seoul, Korea, June 6-9, 2004

[FI4] V. Fischer, L. Allain, L. Gerbaud, "RAMA : a lightweight rule-based tool for expressions analysis and code generation", ESS 2003, Delft, The Netherlands, October 26-29, 2003

[FI5] V. Fischer, L. Gerbaud, "From the electronic circuit to the simulation component : an automatic component building process", ESS 2003, Delft, The Netherlands, October 26-29, 2003

[FI6] V. Fischer, L. Gerbaud, "Solving ODE for Optimisation : Specific use of the Matrix Exponential Approach", OIPE 2002, Lodz, Poland, September 12-14, 2002

[FI7] V. Fischer, L. gerbaud, "CoreLab : a component-based integrated sizing environment", OIPE 2004, Grenoble, France, September 6-8, 2004

[GER] L. Gerbaud, "Aide à la conception des ensembles machine-convertisseur-commande. Apport d'une démarche d'expert." Thèse de Doctorat de l'INPG, 1993

[GE2] L. Gerbaud, "GENTIANE : une plateforme pour la conception des ensembles machine-convertisseur-commande", Rapport d'Habilitation à Diriger des Recherches INPG, 11 Juillet 2000

[GIL] J.C. Gilbert, G. Le Vey, J. Masse, "La différentiation automatique de fonctions représentées par des programmes", Rapport de Recherche n° 1557, INRIA, Unité de Rocquencourt, novembre 1991

[GOC] M.S. Gockenbach, D.R. Reynolds, P. Shen, W.W. Symes, "Efficient and automatic implementation of the adjoint state method", ACM Transactions On Mathematical Software, vol. 28, n°1, pp 22 – 44, March 2002

[GRI] A. Griewank, D. Juedes, H. Mitev, O. Vogel, A. Walther, "ADOL-C : a package for the automatic differentiation of algorithms written in C/C++", Institute for Scientific Computing, TU Dresden, march 1999

[GR2] A. Griewank, "Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation", no. 19, Frontiers in Applied Mathematics, SIAM

[HAR] Y. Harani, "Une approche multi-modèles pour la capitalisation des connaissances dans le domaine de la conception", Thèse de Doctorat de l'INPG, 1997

[HSL] Harwell Subroutine Library, http://www.cse.clrc.ac.uk/nag/hsl/

[IRI] M. Iri, "Simultaneous computation of functions, partial derivatives and estimates of rounding errors, complexity and practicality", Japan Journal of Applied Mathematics, no. 1, pp 223 – 252, 1984

[KAL] R.E. Kalman, "Mathematical Description of Linear Dynamical Systems", J. Soc. Ind. Appl. Math., Ser:A, Control, vol. 1, no. 2, pp 152 – 192, 1963

[KAN] L.V. Kantorovitch, "On a mathematical symbolism convenient for performing machine calculations", Dokl. Akad. Nauk. SSSR, 113, pp 738 – 741, 1957

[LAV] O. Lavoisy, "La matière dans l'action : le graphisme technique comme instrument de la coordination industrielle dans le domaine de la mécanique depuis trois siècles", Thèse de Doctorat de l'INPG / UJF, 14 décembre 2000

[LIO] M.L. Liou, "A novel method of evaluating transcient response", Proc. IEEE, 54, 1966, pp 20 – 23

[MAG] D. Magot, "Méthodes et outils logiciels d'aide au dimensionnement. Application aux composants magnétiques et aux filtres passifs.", Thèse de Doctorat de l'INPG, 28 septembre 2004

[MEI] T.D. Meijler, O. Nierstrasz, "Beyond Objects : Components", Cooperative Information Systems : Current Trends and Directions, M.P. Papazoglou, G. Schlateger Eds, pp 49 – 78, Academic Press, 97

[MUT] A.G.O. Mutambara, "Design and analysis of control systems", CRC Press LLC, 1999, ISBN 0-8493-1898-X

[NAJ] I. Najfeld, T.F. Havel, "Derivatives of the Matrix Exponential and their Computation", Advances in Applied Mathematics, 1994

[NOC] J. Nocedal, "Theory of Algorithms for Unconstrained Optimization", Acta Numerica, 1992, pp 199 – 242

[NOR] O. Normand, A. Bolopion, D. Roye, L. Gerbaud, "Object oriented simulation of electromechanical systems", European Simulation Symposium, SCS-ESS'94, Istanbul, Turkey, Octorber 94, pp 71 – 73

[PRE] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical Recipes in C : The Art of Scientific Computing, 2nd Ed.", Cambridge University Press, 1992, ISBN 0-521-43108-5

[POL] E. Polak, "Computational Methods in Optimization", New York Academic Press, 1971

[PO2] M. Poloujadoff, R.D. Findlay, "A procedure for illustrating the effect of variation of parameters on optimal transformer design", IEEE Transactions on Power Systems, vol. PWRS-1, no. 4, November 1986

[PSP] PSpice, http://www.pspice.com/

[PTO] Ptolemy Project, http://ptolemy.eecs.berkeley.edu/

[RAK] L. Rakotoarison, "Méthodes et outils pour le dimensionnement de microsystèmes magnétiques", Master Recherche INPG, 25 Juin 2004.

[REG] J. Regnier, "Conception optimale de systèmes hétérogènes en Génie Electrique à l'aide d'algorithmes génétiques multi-critères de Pareto", JCGE'03, St Nazaire, Juin 2003, pp 21 – 27

[RIB] V. Riboulet, B. Delinchant, L. Gerbaud, P. Marin, F. Noël, "Tools for dynamic sharing of collaborative design information", Proc. of IDMME 2002, Int. Conf. On MMEngineering, Clermont-Ferrand, May 14th-16th 2002

[SAW] J.W. Sawyer, "First partial differentiation by computer with an application to categorial data analysis", The American Statistician, no. 38, pp 300 – 308, 1984

[SPE] B. Speelpenning, "Compiling fast partial derivatives of functions given by algorithms", PhD Thesis, Department of Computing Science, University of Illinois at Urbana-Champaign, 1980

[SIM] Simplorer, http://www.ansoft.com/products/em/simplorer/

[SUB] SubJava, http://www.ipsi.smr.ru/IPSI.OLD/english/proposal/subjava_e.htm

[SZY] C. Szypersky, "Component Software : Beyond Object-Oriented Programming", Addison-Wesley, 1998

[TIT] O. Titaud, "Analyse et resolution numérique de l'équation de transfert. Application au problème des atmosphères stellaires", Thèse de Doctorat de l'Université de Saint-Etienne, 19 Décembre 2000

[TUR] C. Turpin, F. Richardeau, T. Meynard, F. Forest, "Mesure des pertes dans les convertisseurs de forte puissance par la méthode d'opposition", 8ème EPF, Lille, 29.XI-1.XII.2000, pp. 300-338

[VAN] C. Van Loan, C. Moler, "Nineteen dubious ways to compute the exponential of a matrix", SIAM Review, Vol. 20 No. 4, October 1978, pp 801 – 836

[VAR] R.S. Varga, "On higher order stable implicit methods for solving parabolic partial differential equations", Journal of Applied Physic no 2, 1961, pp 220 – 231

[WAT] Open Watcom, http://www.openwatcom.org/

[WUR] F. Wurtz, "Une nouvelle approche pour la conception sous contraintes de machines électriques", Thèse de Doctorat de l'INPG, 28 mai 1996