ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE MÉCANIQUE
M.Ing.

PAR
LUC PARENT

MODÉLISATION ET OPTIMISATION DE LA PERFORMANCE DES SYSTÈMES DE PRODUCTION PAR USINAGE

MONTRÉAL, LE 20 MARS 2007

CE MÉMOIRE A ÉTÉ ÉVALUÉ
PAR UN JURY COMPOSÉ DE :

M. Victor Songmene, directeur de mémoire
Département de génie mécanique à l'École de technologie supérieure

M. Jean-Pierre Kenné, codirecteur de mémoire
Département de génie mécanique à l'École de technologie supérieure

M. Martin Viens, président du jury
Département de génie mécanique à l'École de technologie supérieure

M. Marek Balazinski, membre du jury
Département de génie mécanique à l'École Polytechnique de Montréal

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC
LE 22 FÉVRIER 2007
À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

MODÉLISATION ET OPTIMISATION DE LA PERFORMANCE DES SYSTÈMES DE PRODUCTION PAR USINAGE

Luc Parent

SOMMAIRE

L'optimisation des conditions d'usinage fait partie des stratégies efficaces pour réduire les coûts d'usinage et demeurer compétitif tout en satisfaisant les besoins des clients. La plupart des modèles d'optimisation des paramètres d'usinage se consacrent à des opérations simples, avec un nombre limité de variables, comme le tournage par exemple. Le fraisage, et plus spécifiquement le fraisage en bout, est pourtant un procédé qui est largement utilisé dans l'industrie, comme dans la fabrication des moules pour l'industrie du plastique et des matériaux composites. Ce mémoire de maîtrise présente un modèle de programmation mathématique pour l'optimisation des paramètres d'une opération de fraisage en bout. Nous allons aussi proposer un algorithme génétique pour résoudre ce problème à dix variables pour les opérations d'ébauche et de finition. Nous allons commencer par décrire le procédé de fraisage en bout et situer l'opération dans un contexte industriel. Ensuite, nous allons décrire les équations de base qui vont permettre la construction des fonctions objectifs, qui incluent le coût d'usinage et le taux de production, avec les contraintes pratiques telles que le fini de surface et les capacités de la machine. Nous allons par la suite présenter une méthode pour résoudre ce problème à l'aide d'un algorithme génétique. Nous allons aussi fournir un exemple numérique, une exploitation du modèle et une analyse de sensibilité. Finalement, nous allons décrire un exemple d'utilisation dans le domaine des systèmes de commande. Ce modèle d'usinage généralisé que nous avons développé pourra facilement être adapté pour déterminer la fenêtre d'opération optimale de la plupart des opérations d'usinage, incluant le fraisage, le tournage et le perçage. Ce mémoire a aussi inspiré un article scientifique qui sera publié dans la revue Production Planning & Control sous le titre : *A generalized model for optimizing end milling operation.*

MODELING AND OPTIMIZATION OF THE PERFORMANCE OF THE SYSTEMS OF PRODUCTION BY MACHINING

Luc Parent

ABSTRACT

Machining condition optimization is one of the efficient strategies used to reduce machining costs and to remain competitive while meeting customer's needs. Most models used for optimizing machining parameters usually deal with simple operations, with limited numbers of variables, such as turning. Milling, especially end milling, operations account for a high percentage of operations commonly used in industry, particularly in mould making. This master's thesis presents a generalized mathematical programming model for optimizing the machining parameters of an end milling operation. A genetic algorithm is also proposed for solving this non-linear model and providing manufacturers with optimal cutting conditions, taking into account ten machining variables for roughing and finishing operations. The work starts with the description of the end milling process and situates it in an industrial context. We then describe the basic equations allowing the construction of the objective functions, including machining cost and productivity, with practical constraints such as the required surface finish and machine capability. We then present a method to solve the problem using a genetic algorithm. We will also provide a numerical example, model exploitation and sensitivity analysis. Finally, we describe an example of application of this model in a hierarchical control system. The generalized machining model developed can easily be adapted to determine optimal operational windows for most machining processes, including milling, turning and drilling. This master's thesis inspired a scientific paper for the Production Planning & Control journal with the title: *A generalized model for optimizing end milling operation.*

REMERCIEMENTS

J'ai eu la chance d'élaborer ce travail sous la direction de M. Victor Songmene et M. Jean-Pierre Kenné, qui m'ont guidé dans mes recherches et aidé à développer ma démarche scientifique. Je les remercie pour le soutien financier, le temps qu'ils m'ont consacré et tous les conseils qu'ils m'ont prodigués dans mes études de maîtrise.

Je tiens aussi à remercier tous les professeurs et le personnel du département de génie mécanique pour leur aide ponctuelle.

J'adresse un merci tout particulier à ma conjointe Mélanie, pour ses encouragements et sans qui mes études supérieures n'auraient été possibles.

Merci à mes amis, à ma famille et à ma belle-famille pour leur compréhension et leur support moral.

TABLE DES MATIÈRES

Page

SOMMAIRE I

ABSTRACT II

REMERCIEMENTS III

TABLE DES MATIÈRES IV

LISTE DES TABLEAUX VI

LISTE DES FIGURES VII

LISTE DES ABRÉVIATIONS ET SIGLES IX

INTRODUCTION 1

CHAPITRE 1 INTRODUCTION ET PROBLÉMATIQUE 2

1.1 Le système de production par usinage 2
1.1.1 Les stocks tampons 3
1.1.2 Exemple de système de production par usinage 3
1.2 Le fraisage en bout 6
1.2.1 La fraise en bout 6
1.3 Revue de la littérature 9
1.4 Les objectifs de l'optimisation et la stratégie de commande optimale 14
1.5 Problématique de recherche 16

CHAPITRE 2 MODÉLISATION ET OPTIMISATION 18

2.1 Modélisation 18
2.1.1 Les fonctions objectifs 20
2.1.2 Les contraintes 23
2.2 Optimisation 27
2.2.1 Un algorithme génétique 27
2.2.2 La fonction d'adaptation 31

CHAPITRE 3 RÉSOLUTION ET ANALYSES 34

3.1 L'implantation 34

3.1.1 Interface et fonctionnement du logiciel 35
3.1.2 Fonctionnement interne du logiciel 39
3.1.3 Vecteurs et matrices de taille variable 40
3.1.4 Description des fonctions de la classe *view* 41
3.1.5 Description des fonctions de la classe document 42
3.2 Exemple numérique 47
3.2.1 Résolution 48
3.2.2 Comparaison des résultats 49
3.2.3 Analyse de l'optimum 50
3.3 Variation des paramètres 53
3.4 Analyse de sensibilité 59
3.4.1 Objectifs et méthodologie de cueillette de données 60
3.4.2 Résultats et analyses 64

CHAPITRE 4 FABRICATION ET GESTION DE LA PRODUCTION 71

4.1 Exploitation dans les systèmes de contrôle 71
4.1.1 Les systèmes de production flexibles 71
4.1.2 La stratégie de commande optimale 73
4.2 Autres applications 77
4.2.1 Capacité et faisabilité d'un système 77
4.2.2 Planification et gestion des stocks de soutien 78
4.2.3 Extension des modèles de contrôle 79

CONCLUSION 80

ANNEXE 1 LE MODÈLE MATHÉMATIQUE 83

ANNEXE 2 LE CODE DU LOGICIEL 87

BIBLIOGRAPHIE 123

LISTE DES TABLEAUX

Page

Tableau I Le génome 29
Tableau II Résultats des solveurs 49
Tableau III Résultats comparatifs 49
Tableau IV Temps et coûts d'usinage selon P_{max} 64
Tableau V Temps et coûts d'usinage selon F_{max} 65
Tableau VI Temps et coût d'usinage selon P_{max} et F_{max} 67

LISTE DES FIGURES

Page

Figure 1 Exemple de système de production par usinage 4
Figure 2 Fraise en bout (vue de côté) 7
Figure 3 Fraise en bout (vue d'en bas) 7
Figure 4 Fraise en bout (détail de la dent) 8
Figure 5 La pièce et l'outil 8
Figure 6 Fraisage en opposition et en concordance 9
Figure 7 Trajectoire des stocks en fonction du temps 15
Figure 8 Les fonctions objectifs 19
Figure 9 Schéma algorithmique 30
Figure 10 Les paramètres de l'opération et de la machine 35
Figure 11 Les paramètres de l'algorithme et les fonctions objectifs 37
Figure 12 Le déroulement de l'optimisation 38
Figure 13 Coût d'usinage U en fonction de la vitesse d'avance fr et de coupe vr 50
Figure 14 Force de coupe fcr en fonction de la vitesse d'avance fr 51
Figure 15 Coût d'usinage U en fonction de la vitesse de coupe v_r 51
Figure 16 Fini de surface R_{ae} en fonction des vitesses d'avance f_f et de coupe v_f 52
Figure 17 Coût d'usinage U selon α 53
Figure 18 Temps d'usinage Tu selon α 54
Figure 19 Coût d'usinage U selon β 54
Figure 20 Coût d'usinage U selon γ 55
Figure 21 Coût d'usinage U selon C_P 56
Figure 22 Coût d'usinage U selon Ra 56
Figure 23 Coût d'usinage U selon Ts 57
Figure 24 Coût d'usinage U selon N_L 58
Figure 25 Force en fonction de de la vitesse d'avance en ébauche 60
Figure 26 Puissance (3D) en fonction de l'avance et de la vitesse de coupe en ébauche 61

Figure 27 Puissance (courbe de niveau) en fonction de l'avance et de la vitesse de coupe en ébauche 62
Figure 28 Temps d'usinage, évolution de la moyenne avec les réplications 63
Figure 29 Temps et coût d'usinage selon F_{max} 66
Figure 30 Temps et coût d'usinage selon P_{max} et F_{max} 68
Figure 31 Système de contrôle 74
Figure 32 Coût d'usinage minimum pour le temps d'usinage requis 75

LISTE DES ABRÉVIATIONS ET SIGLES

Abréviations :

AG Algorithme(s) génétique(s)

FMS Systèmes de production flexibles (Flexible Manufacturing systems en anglais)

HSS Acier rapide (High Speed Steel en anglais)

SMED Échange rapide de mise en course (Single Minute Exchange of Die en anglais)

WIP Stocks en-cours (Work In Progress en anglais)

Notation du modèle et unités :

(Les indices $_r$ et $_f$ représentent respectivement les symboles pour les passes d'ébauche et de finition.)

C_A	Coût d'aiguisage de l'outil ($)
C_{Hr}, C_{Hf}	Coût horaire de l'outil ($/min)
C_{pr}, C_{pf}	Coût d'achat de l'outil ($/outil)
C_B	Coût du matériel brut ($)
D_r, D_f	Diamètre de l'outil (mm)
d_r, d_f	Profondeur (axiale) de la passe (mm)
f	Avance par dent (mm/dent)
F_{rmax}, F_{fmax}	Force de coupe maximale (N)
H	Hauteur du volume à usiner (mm)
h_{rmax}, h_{fmax}	Longueur des flûtes de l'outil (mm)
k	Constante de Taylor (matériau de la pièce)

k_0	Coût d'opération, incluant salaire et machine ($/min)
L	Longueur du volume à usiner (mm)
N_{Ar}, N_{Af}	Nombre d'aiguisages avant de jeter l'outil (entier)
N_L	Nombre de pièces par lot (entier)
N_{max}	Vitesse maximale de rotation de la broche (tour/min)
N_p	Nombre de passes en profondeur (entier)
N_r, N_f	Vitesse de rotation de la broche (tour/min)
N_w	Nombre de passes axiales (entier)
P	Taux de production (unités/heure)
P_{max}	Puissance maximale de la machine (kW)
P_r	Puissance requise (kW)
P_s	Puissance spécifique (du matériau de la pièce) (Watt/mm^3/min)
R	Distance à parcourir en avance rapide, par opération (mm)
R_{ae}	Fini de surface en bout (rugosité moyenne, µm)
R_{ap}	Fini de surface en périphérie (rugosité moyenne, µm)
t_{0r}, t_{0f}	Durée de vie d'un outil avant aiguisage (Taylor) (min)
t_{cr}, t_{cf}	Temps de coupe, ébauche et finition (min/unité)
t_e	Temps pour changer un outil émoussé (min)
t_L	Temps pour charger et enlever une pièce (min)
t_s	Temps de mise en course pour l'opération (min)
T_U	Temps d'usinage unitaire (min)
U	Coût d'usinage unitaire ($/unité)
v_r, v_f	Vitesse de coupe (m/min)
v_{fr}, v_{ff}	Vitesse d'avance (en unité de temps) (mm/min)
v_{fmax}	Vitesse d'avance maximale de la machine (en unité de temps) (mm/min)
v_t	Vitesse d'avance rapide (mm/min)
V_u	Volume à usiner (mm^3)
W	Largeur du volume à usiner (mm)
w_r, w_f	Largeur de la passe (mm)

X_0 à X_4	Facteurs de l'expression du fini de surface (sans unité)
Z	Nombre de dents de l'outil (entier)
Z_w	Débit du copeau (mm^3/min)
α	Exposant de Taylor (vitesse) (sans unité)
β	Exposant de Taylor (avance) (sans unité)
γ	Exposant de Taylor (profondeur de coupe) (sans unité)
η	Efficacité énergétique de la machine (%)
λ	Exposant de Taylor (largeur de coupe) (sans unité)

INTRODUCTION

L'étude économique des procédés de fabrication est un domaine d'intérêt constant pour les chercheurs depuis le début du 20^{ième} siècle (Ermer 1997). Dans l'industrie de fabrication moderne, la coupe de métal sur machine outil à commande numérique est le procédé par excellence pour la production en interrompu (par lot). Les tâches de support pour une opération d'usinage incluent la sélection des machines, la conception des montages, la sélection des outils et des paramètres d'usinage et la détermination des trajectoires d'outils. Dans la pratique, un expert en programmation d'usinage peut concevoir un plan d'usinage en analysant les données graphiques et en sélectionnant, selon son expérience, les différents paramètres qui satisferont aux exigences de fini et de précision. Mais le procédé doit être productif, compétitif et rentable, ce qui demande des connaissances approfondies du procédé de fabrication, des capacités des machines, du dessin et de la programmation assistée par ordinateur ainsi que de l'optimisation.

Dans le but d'alléger la tâche de ces experts et d'assurer aux gestionnaires des conditions d'usinage optimales selon leurs objectifs de gestion, il est nécessaire de développer un modèle d'optimisation pour les paramètres d'usinage. Ce modèle, une fois développé, devra être facilement utilisable en pratique afin qu'il soit vraiment utile dans l'industrie. Sachant qu'ils disposent de conditions d'usinage optimales, les gestionnaires pourront se consacrer à d'autres aspects de la productivité tels les temps de mise en course, la manutention, le balancement des opérations, la gestion de la qualité, etc. Ce modèle pourra aussi, par la suite, servir d'interface entre un système de contrôle et l'opération de fraisage dans un environnement de production automatisée, comme un système de production flexible par exemple.

Le modèle d'optimisation doit s'occuper de deux fonctions objectifs : minimiser le coût de production et minimiser le temps de production. Nous allons, dans ce document, développer ce modèle et résoudre le problème d'optimisation.

CHAPITRE 1

INTRODUCTION ET PROBLÉMATIQUE

Dans ce chapitre, nous allons décrire un système de production par usinage typique. Nous allons décrire en détail l'opération de fraisage et la situer dans le système de production afin de bien comprendre les enjeux de l'optimisation et l'importance des deux fonctions objectifs. Nous allons ensuite faire une revue de la littérature pour voir les recherches qui ont déjà été faites sur le sujet. Finalement, nous allons situer notre modèle dans une des applications possibles dans le domaine des systèmes de contrôle avec les politiques de commandes optimales.

1.1 Le système de production par usinage

Le système de production que l'on va décrire ici n'est pas nécessairement inspiré d'un système réel. Il s'agit d'un exemple assez complexe pour comprendre les implications industrielles de l'opération, mais assez simple pour permettre de se concentrer sur l'essentiel. Le but de cette section est de bien faire comprendre que lorsqu'on se concentre sur l'optimisation d'une opération, il ne faut pas oublier qu'il peut y avoir des considérations extérieures à l'opération elle-même. Le fait de situer l'opération de fraisage dans un contexte plus large va aussi aider à considérer le rôle des politiques de commandes optimales présenté au chapitre 4.

Un système de production par usinage comprend généralement plusieurs opérations d'usinage ou autre, aménagé en série. Les opérations d'usinage peuvent être : sciage, perçage, tournage, surfaçage, fraisage, meulage, brochage, etc. Les autres opérations peuvent être : nettoyage, peinture, hydroformage, traitement thermique, traitement de surface, inspection, etc. Les opérations d'usinage ou autre ne sont pas les seuls éléments

d'une chaîne de production. Il faut aussi tenir compte des stocks tampons entre les opérations.

1.1.1 Les stocks tampons

Dans la plupart des aménagements de production, il y a des accumulations de stocks entre les différentes opérations. On appelle ces stocks des « en-cours » ou « work in progress » (WIP). Ces stocks en-cours permettent un flux continu de pièces dans le système de différentes façons :

- Permets d'éviter les attentes de machines quand les temps de cycles sont différents.
- Permets d'éviter un arrêt de la ligne de production en cas de panne d'une machine.
- Permets d'éviter un arrêt de la ligne lors d'une mise en course ou d'un remplacement d'outil.
- Permets de limiter les effets des maintenances préventives et correctives.

Un des objectifs de la gestion de la production est de minimiser la taille des stocks, y compris les en-cours. Il reste néanmoins qu'un minimum d'en-cours est souvent nécessaire en pratique et la gestion de ceux-ci, de la production globale et de chaque opération est interdépendante.

1.1.2 Exemple de système de production par usinage

Notre exemple de système se compose de quatre opérations :

- Sciage : Les barres de matériaux bruts sont coupées en blocs.
- Fraisage : Les blocs sont usinés sur une fraiseuse verticale (nous allons nous concentrer sur cette opération plus loin).

- Traitement thermique : Sans entrer dans le détail de cette opération, mentionnons que les pièces sont chargées dans un four en assez grande quantité pour un certain temps. Les tailles de lots et les temps d'opération sont en général assez importants pour les traitements thermiques.
- Meulage : Les surfaces fonctionnelles de la pièce sont amenées à la précision et au fini de surface requis.

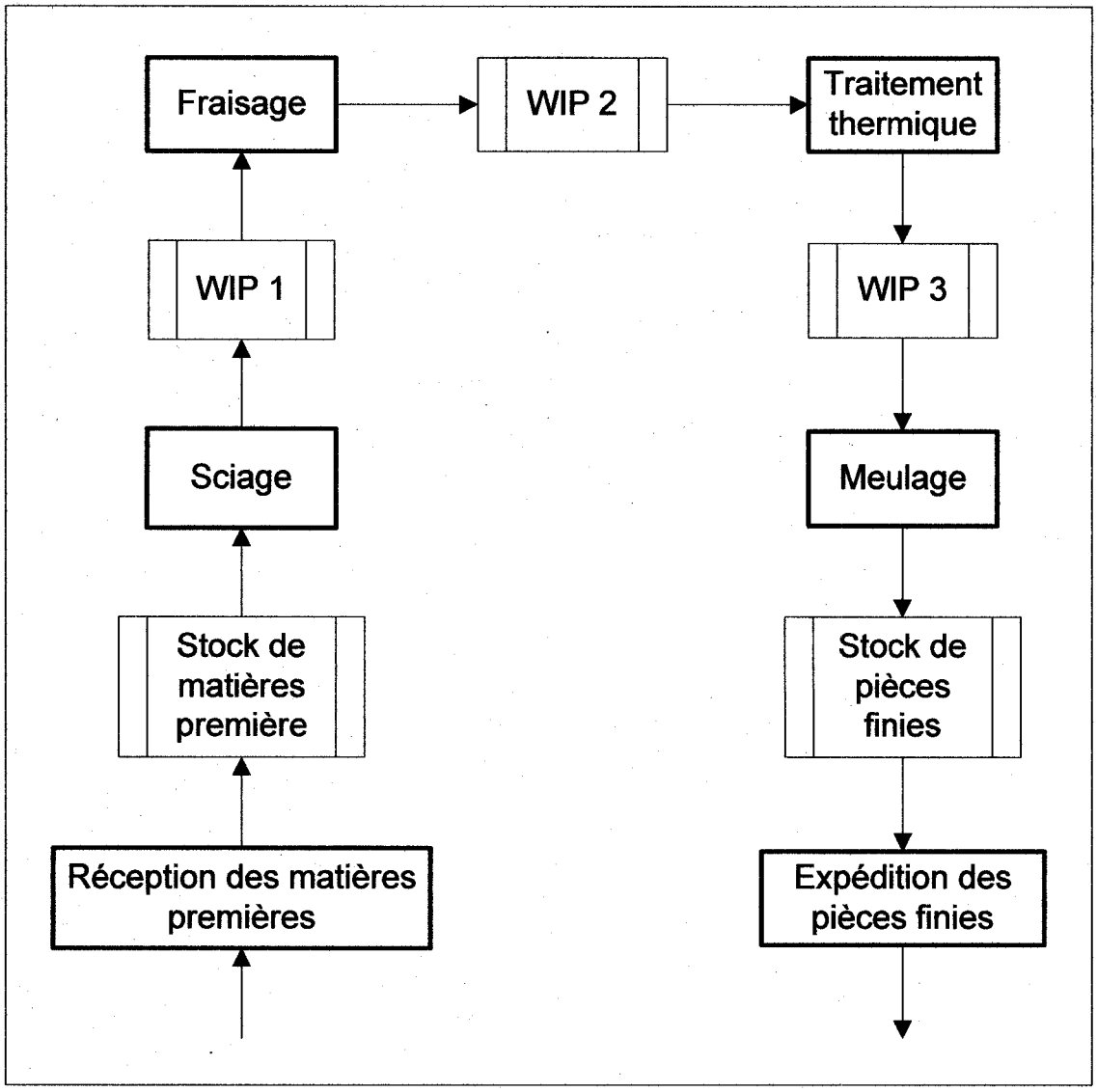

Figure 1 Exemple de système de production par usinage

Pour compléter notre système, nous avons la réception des matières premières et l'expédition des produits finis. La réception peut provenir d'un autre département, d'un système plus vaste ou d'un fournisseur externe. De même, l'expédition peut se faire vers un autre département, l'assemblage par exemple, ou vers un client externe. La Figure 1 illustre schématiquement notre système.

Le gestionnaire qui doit s'assurer du bon fonctionnement d'un système comme celui décrit plus haut a besoin d'information de la part de chacune des opérations. Il doit au moins savoir le temps de production de chacune des étapes afin de balancer les lignes de production, gérer les stocks et s'assurer de la faisabilité d'une commande. Pour connaître les temps de production, et spécialement des opérations complexes comme le fraisage en bout, les opérations doivent être modélisées afin de mieux les connaître et de les optimiser. Aussi, un tel gestionnaire n'est pas nécessairement un spécialiste de l'usinage. Il aura donc besoin d'un outil objectif afin de traduire les différentes variables du procédé en caractéristiques qu'il peut comprendre et contrôler, soit les temps et les coûts d'usinage. Pour ces raisons, nous croyons qu'il est impératif de développer un modèle mathématique qui représente bien le procédé réel et une méthode qui permet l'optimisation de ce modèle. De plus, étant donné que le modèle va comprendre des contraintes, il permettra d'identifier les limites de l'opération et de fournir des pistes de solution afin d'augmenter les taux de production et de diminuer les coûts.

Nous allons revenir sur notre exemple de système au chapitre 4 où nous allons traiter de système de contrôle et de commande optimale.

1.2 Le fraisage en bout

Il existe plusieurs types de fraisage. Le fraisage horizontal, le surfaçage et le fraisage en bout en sont les grandes familles. D'un point de vue mécanique, l'opération qui comporte le plus de paramètres est le fraisage en bout parce qu'il permet de générer des surfaces en trois dimensions, c'est-à-dire des surfaces finies autant verticales qu'horizontales et même obliques sur certains centres d'usinage. La raison pour laquelle nous avons choisi l'opération de fraisage en bout pour notre modèle est que cette opération nous permettra de développer un modèle le plus général possible qui pourra facilement être adapté aux autres types de fraisage.

1.2.1 La fraise en bout

La fraise en bout est un outil de coupe cylindrique qui coupe autant sur le bout que sur la périphérie. Les figures 2 à 5 illustrent le procédé en action. On peut voir sur ces figures les paramètres de coupe que notre modèle d'optimisation nous permettra de déterminer soit :

N : La vitesse de rotation de l'outil
f : L'avance de l'outil par dent
d : La profondeur de coupe (axiale)
w : La largeur de coupe (radiale)
D : Le diamètre de l'outil

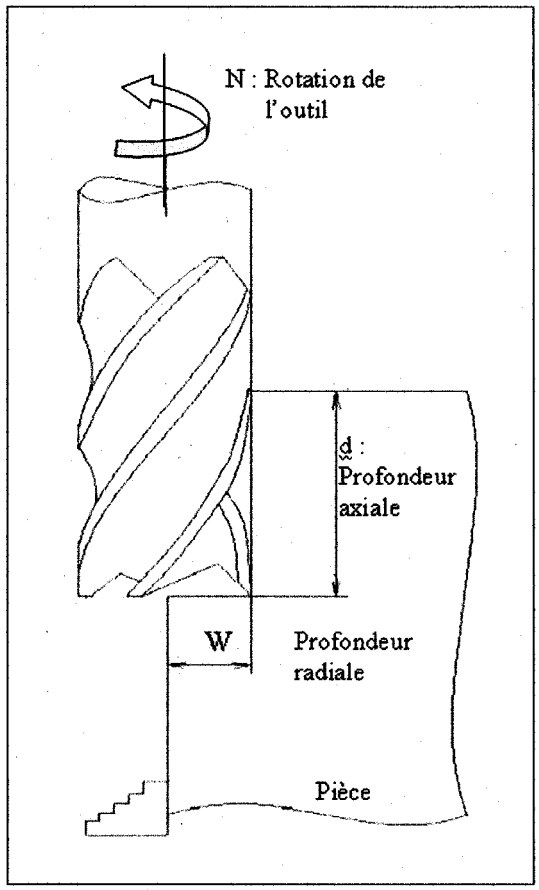

Figure 2 Fraise en bout (vue de côté)

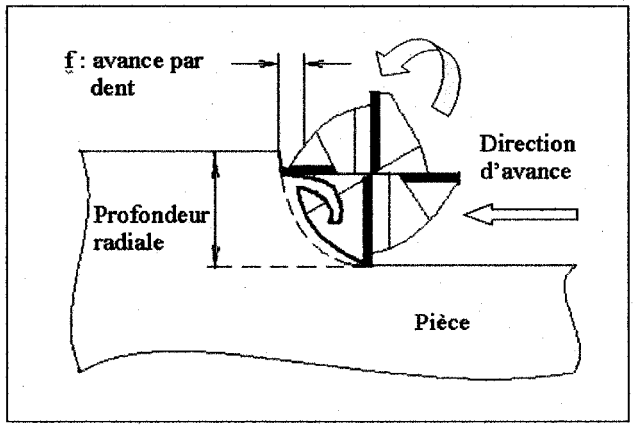

Figure 3 Fraise en bout (vue d'en bas)

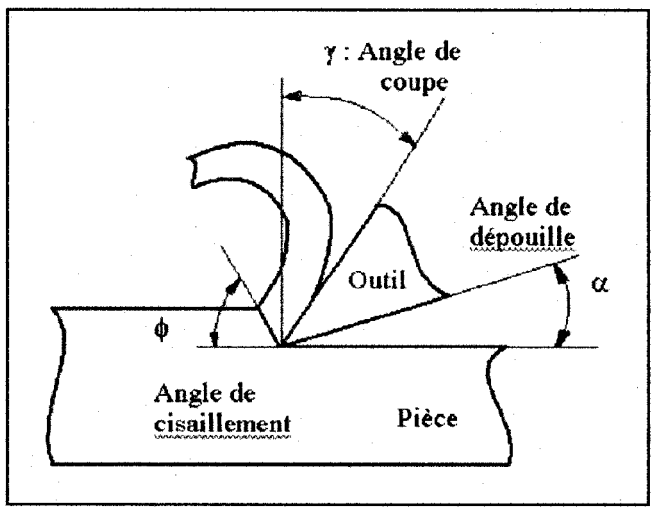

Figure 4 Fraise en bout (détail de la dent)

Nous allons considérer un volume de matériau V_u à enlever tel qu'illustré à la Figure 5.

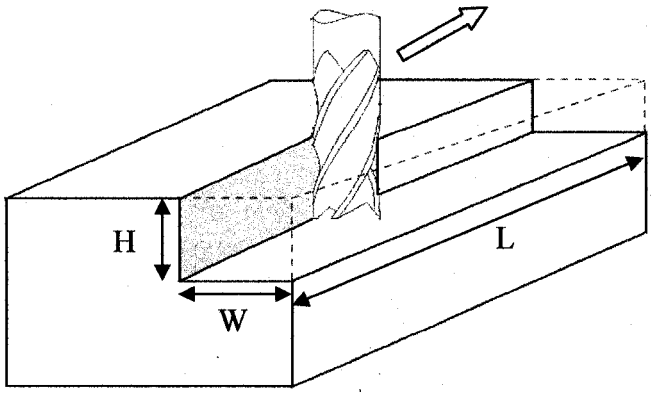

Figure 5 La pièce et l'outil

On peut y voir le matériel brut, le volume à usiner, ainsi que la position de l'outil en déplacement dans le volume à usiner.

Il y a deux façons de faire avancer la fraise dans le matériau : en concordance ou en opposition, comme illustré à la figure 6. On peut utiliser un ou l'autre de ces modes, et la décision dépend de plusieurs facteurs. Nous n'allons pas discuter de cela dans ce

mémoire. Ce qui est important de mentionner pour notre modèle, c'est qu'il faut utiliser le même mode pour usiner que celui qui a été utilisé dans les plans d'expérience.

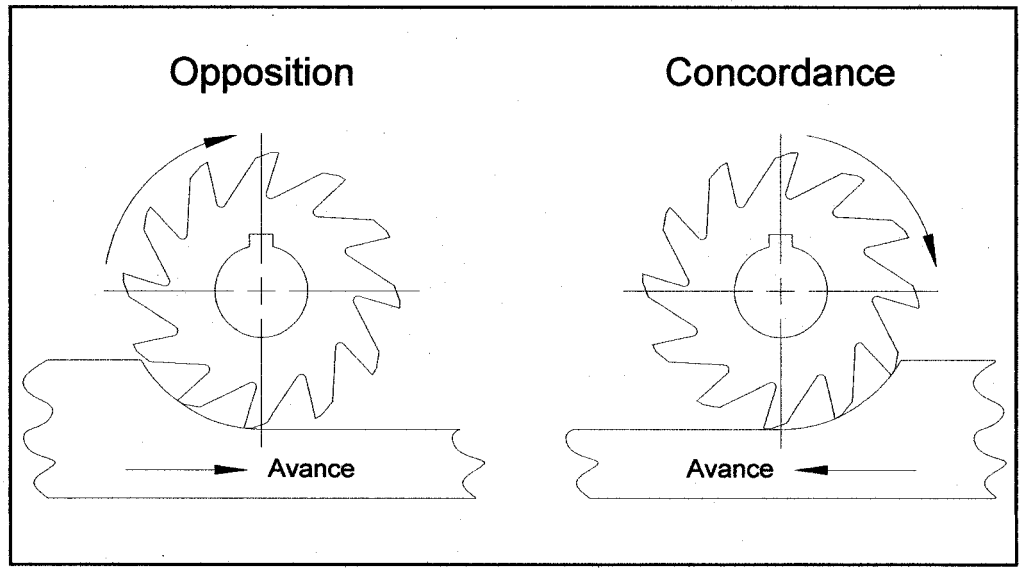

Figure 6 Fraisage en opposition et en concordance

Ces plans d'expérience sont requis afin de déterminer les paramètres des équations 2.6, 2.7, et 2.10 (chapitre 2). À défaut d'utiliser le même mode, on peut rendre les prévisions du modèle imprécises, voire même erronées.

Nous allons utiliser une pièce et un outil typique pour construire notre modèle. La pièce est un prisme rectangulaire auquel on doit enlever un volume, qui est lui aussi prismatique. L'outil est une fraise en bout à quatre flûtes (dents) standard, c'est-à-dire sans rayon de coin.

1.3 Revue de la littérature

Nous allons commencer cette section par un petit historique de la recherche dans le domaine de l'optimisation des procédés d'usinage. Cet historique est principalement

basé sur l'article de Donald S. Ermer, *« A Century of Optimizing Machining Operations »* (1997), qui a été écrit dans le cadre du 75^{ième} anniversaire du « Journal of Manufacturing Science and Engineering ». En effet, il y a maintenant plus d'un siècle que les chercheurs travaillent sur l'optimisation des procédés d'usinage. Le pionnier dans le domaine est sans nul doute Frederic Winslow Taylor qui, dès 1893, s'est intéressé à la durée de vie des outils de coupe d'un point de vue économique. Taylor est aussi bien sûr reconnu et critiqué comme étant le fondateur de l'organisation scientifique du travail. Celui-ci a publié en 1907 son modèle empirique de la durée de vie de l'outil dans l'article *On the first art of cutting metals*. Il y présente une équation de la durée de vie d'un outil de tournage qui lui permet de faire la première analyse économique en utilisant le temps minimum d'usinage comme objectif. Ce modèle ne tient compte que d'une variable : la vitesse de coupe. Ensuite, il n'y a plus eu de développement majeur dans la recherche scientifique sur la modélisation et l'optimisation des procédés d'usinage durant la première moitié du XX ^{ième} siècle.

En 1950, W. W. Gilbert propose pour sa part que la minimisation du coût et la maximisation du taux de production soient les critères adéquats pour l'étude économique des procédés d'usinage. Il considère lui aussi la vitesse de coupe comme seule variable sur une opération de tournage. Il produit aussi plusieurs monographies pour faciliter l'application de son modèle et donne quelques méthodes informelles pour vérifier les capacités en puissance des machines, ce qui annonce les futures contraintes d'optimisation. Cette approche fut popularisée pendant les deux décennies suivantes par la technique « Hi-E » de Carboloy. Cette technique consistait en des abaques et des graphiques qui tiennent compte des coûts et des temps : non productifs, d'usinages et des outils. Il ressort de cette approche qu'il y a une vitesse de coupe pour le coût d'usinage minimum et une autre, plus élevée, pour le taux de production maximum. Entre ces deux points, on a la plage d'usinage optimale, dont nous discuterons plus loin.

C'est en 1966 que Wu et Ermer introduisent un troisième critère d'optimisation : la maximisation du taux de profit. Ils considèrent le principe de l'offre et de la demande qui fait en sorte que plus le prix de revient est élevé, plus la demande baisse. Pour notre part, nous allons considérer que les principes de l'offre et de demande sont extérieurs à l'opération d'usinage et qu'ils n'entrent donc pas dans notre modèle.

Toutes les études dont nous avons parlé jusqu'à maintenant se concentraient sur le tournage. Bien sûr, ces modèles peuvent être adaptés au fraisage dans une certaine mesure puisque la vitesse de coupe y est aussi la variable la plus significative. Par contre, les autres variables, telles que la vitesse d'avance et les profondeurs de passes ont une influence qui n'est pas négligeable. Très peu de travaux ont traité de l'optimisation du fraisage en général, et ce, même depuis ces dernières années.

C'est en 1983 que Ermer et Araj présentent une solution complète de l'optimisation d'une opération de fraisage périphérique (en bout) avec des contraintes pratiques. Ils ont démontré que l'optimisation du coût et du temps d'usinage donne généralement des résultats différents, sauf dans certains cas où les contraintes forcent les solutions à un même point. En général donc, les vitesses de coupe et d'avance sont plus élevées si on minimise le temps d'usinage que si on minimise le coût d'usinage. Cette solution était considérée comme complète à l'époque parce qu'elle tenait compte des deux variables principales, soit les vitesses de coupe et d'avance, en plus de déterminer le diamètre de l'outil et le nombre de dents. Néanmoins, ce modèle ne tenait compte que de quatre variables, et ce, pour une seule passe.

Plus récemment, Agapiou, en 1992, déterminait les conditions d'usinage pour des opérations de passe unique et multi passe en minimisant une combinaison linéaire du coût et du temps de production. Son modèle est donc multicritère et permet de résoudre le problème à l'aide d'une méthode de simplexe pour une passe unique. Il utilise la

programmation dynamique pour résoudre son modèle multi passes. Encore ici, l'auteur ne considère que l'opération de tournage avec seulement trois variables.

B. Malakooti et J. Deviprasad (1989) proposent une méthode basée sur la programmation en nombre entier pour résoudre le problème d'optimisation d'une opération de tournage. Leur méthode n'est pas un modèle complètement automatisé d'optimisation, mais plutôt un système informatique d'aide à la décision. Leur approche multicritère tente de résoudre le problème en minimisant simultanément le temps d'usinage, le coût d'usinage et le fini de surface. Leur méthode utilise un heuristique qui repose sur le gradient. Malheureusement, ils considèrent le fini de surface comme un objectif plutôt que comme une contrainte et leur modèle de fini de surface, s'il est bien adapté pour le tournage, ne convient pas du tout au fraisage en bout.

B. White et A. Houshyar (1992) proposent un modèle d'optimisation algébrique basée sur la méthode de Khun-Tucker pour résoudre l'optimisation d'une opération de tournage. Ils sous-entendent que leur méthode peut être adaptée à d'autres opérations d'usinage, mais ceci s'annonce très difficile, et ce, pour deux raisons. Premièrement, ils utilisent l'équation de base pour la durée de vie de l'outil développée par Taylor (1907). Cette équation de tiens compte que de la vitesse de coupe et n'est donc pas complètement adaptée au fraisage en bout. Deuxièmement, leur équation du fini de surface utilise le rayon de coin de l'outil de tournage. Cette équation peut être adéquate pour le tournage, mais ne s'applique pas du tout au fraisage en bout. D'autre part, leur méthode de résolution algébrique s'applique bien et s'utilise facilement pour optimiser une variable, mais devient beaucoup plus complexe s'il y a plus d'une variable.

P. Munoz-Escalona et Z. Cassier (1998) proposent un modèle qui semble très performant pour prédire le fini de surface de plusieurs aciers machinés sur un tour. Leur modèle tient compte de la vitesse de coupe, de la vitesse d'avance, de la profondeur de coupe et du rayon de coin de l'outil. Ils utilisent un plan d'expérience pour vérifier leurs

résultats. Ils ne font pas d'optimisation, mais se contentent de modéliser le fini de surface et, encore une fois, leur modèle n'est pas adapté au fraisage en bout.

C'est finalement en 1999 que Lou, Chen et Caleb proposent un modèle adéquat pour modéliser le fini de surface d'une opération de fraisage en bout. Il s'agit d'un modèle empirique basé sur un plan d'expérience qui tient compte des trois variables les plus importantes pour le fraisage en bout : la vitesse de coupe, la vitesse d'avance et la profondeur de coupe. C'est le modèle le plus adéquat qui a été développé à ce jour. Nous allons donc l'utiliser dans le modèle que nous développons au chapitre 2.

Al-Ahmari (2001) présente un modèle très complet, le plus complet à ce jour, pour déterminer les paramètres d'usinage d'une opération de tournage multi passes. Comme il s'agit d'un modèle hautement non linéaire, il propose d'utiliser un logiciel d'optimisation commercial pour le résoudre.

Comme on l'a décrit dans cette revue de la littérature, plusieurs articles très pertinents ont été publiés pour modéliser et optimiser en déterminant les paramètres de coupe d'une opération de tournage, mais très peu ont traité des opérations de fraisage. Nous nous proposons donc, dans le présent ouvrage, de développer un modèle complet décrivant l'opération de fraisage en bout. Ce modèle devra contenir les cinq variables de décision suivantes : vitesse de coupe, vitesse d'avance, diamètre de l'outil, profondeur de coupe radiale et profondeur de coupe axiale. Ces cinq variables devront être considérées à la fois pour une passe d'ébauche et une passe de finition, ce qui nous fait un total de dix variables. Notre modèle devra aussi tenir compte de toutes les contraintes physiques de l'opération. Il faudra aussi fournir une approche de solution efficace et robuste qui puisse être programmée et pilotée de façon à être utilisable, même pour des lignes de production hautement automatisées.

1.4 Les objectifs de l'optimisation et la stratégie de commande optimale

Plusieurs articles très pertinents ont été publiés sur les stratégies de commande optimale dans le domaine des systèmes de contrôle. Celui de Kenné et Gharbi (2004) en est un bon exemple. Dans ce domaine, chaque machine a un statut. Par exemple : la machine peut être en opération ou en panne, en maintenance préventive ou corrective, etc. Selon le statut de la machine et d'autres informations comme le niveau des stocks, le système de contrôle va émettre une commande pour la machine. Cette commande peut être de produire ou non, de faire une maintenance préventive, etc. Quand une commande de production est donnée, si le système est assez sophistiqué, on peut demander à la machine de produire à un certain taux précis, qui tient compte de facteurs externes à l'opération elle-même.

Prenons un exemple de base afin de bien comprendre le fonctionnement de la stratégie de commande. Dans un système à une machine et un produit, où la machine est sujette à des pannes aléatoires, le problème consiste en la détermination du niveau du stock tampon après la machine qui permet de minimiser le coût de maintien des inventaires et de rupture de stock. Akella et Kumar (1986) ont prouvé que la politique à seuil critique est optimale pour ce problème de contrôle. La politique de commande pour un tel système prend donc la forme suivante :

$$u(t) = \begin{cases} U_m & si\ x(t) < X \\ d & si\ x(t) = X \\ 0 & si\ x(t) > X \end{cases} \tag{1.1}$$

Où u(t) est le taux de production requis au temps t, U_m est le taux de production maximal, d est le taux de demande, x(t) le niveau du stock au temps t et X est le seuil critique. Cette politique de contrôle signifie que si le niveau du stock est inférieur au

seuil critique, il faut produire au taux de production maximal, si le niveau du stock est égal au seuil critique, il faut produire au même taux que la demande et si le niveau du stock est supérieur au seuil critique il ne faut pas produire du tout.

Le niveau des stocks varie donc dans le temps et le taux de production commandé dépend de ce niveau de stocks. La Figure 7 illustre la variation des stocks dans le temps pour un tel système avec X=15.

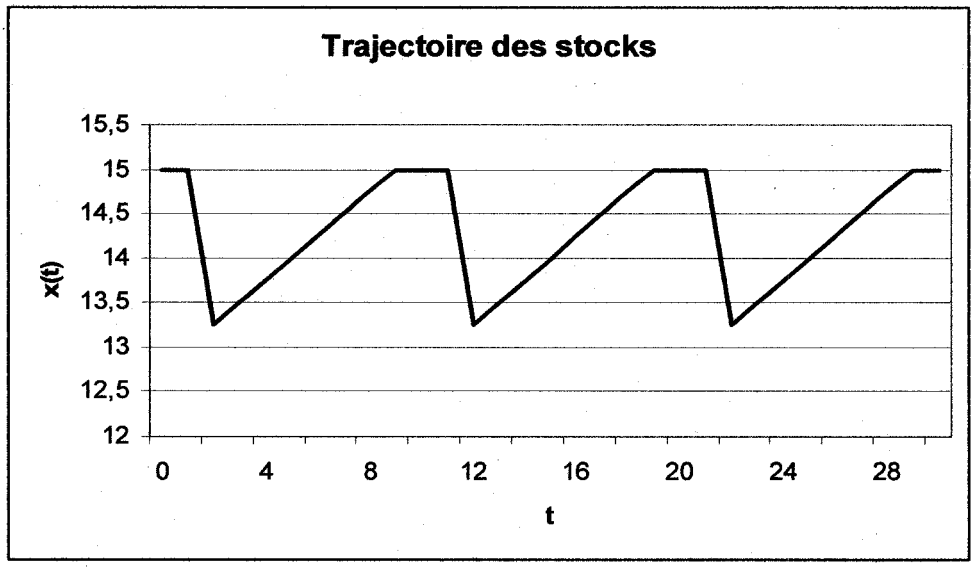

Figure 7 Trajectoire des stocks en fonction du temps

Nous n'allons pas entrer dans le détail de la résolution d'un tel problème, mentionnons seulement qu'on procède généralement par simulation. On peut néanmoins noter que pour le résoudre, il faut connaître le taux maximal de production et pour l'appliquer, il faut savoir comment produire à ce taux et au taux de la demande.

Si cette opération en est une de fraisage en bout, il faut :

- Déterminer quel est le taux maximal de production de la machine en minimisant le temps d'usinage.

- Minimiser le coût d'usinage ce qui nous donnera le taux de production minimum économique de l'opération.
- S'assurer que le taux de la demande est situé entre les taux de production minimum et maximum.
- Déterminer les conditions de coupes optimales pour produire au taux précis de la demande.

1.5 Problématique de recherche

Notre modèle d'optimisation devra donc fonctionner dans les deux sens, c'est-à-dire que la minimisation des deux fonctions objectifs va fournir les limites de performance de l'opération au système de contrôle, définissant ainsi les commandes admissibles. On appelle cette zone opérationnelle la plage ou la fenêtre d'opération optimale. À partir de ces informations, le système de contrôle peut émettre une commande de lancer la production à un certain taux précis de production, soit au taux de la demande ou au taux maximum. Notre modèle pourra ensuite servir d'intermédiaire en fournissant les paramètres de fraisage adéquats afin de minimiser le coût d'usinage au taux de production demandé par le système de contrôle. De la même manière, si on s'intéresse principalement au coût d'usinage, on peut minimiser le temps d'usinage pour un coût de production donné.

Nous allons donc avoir deux fonctions objectifs à minimiser : le temps d'usinage et le coût d'usinage (ces fonctions sont décrites au chapitre 2). Dans un premier temps, à partir d'une série de données sur l'opération, nous allons déterminer le temps d'usinage minimum pour lequel l'opération peut être effectuée, cela nous fournira le taux de production maximal. À cette vitesse, on obtient un certain coût d'usinage. De même, nous allons minimiser le coût d'usinage auquel sera associé un temps d'usinage (plus élevé que le minimum). Après cette première étape, on dispose des limites de performance de notre opération. Si on usine pour obtenir le coût minimum, on aura un

temps d'usinage maximum, c'est-à-dire le taux de production minimal auquel l'opération peut être effectuée économiquement. Si on essaie de produire moins vite, il en coûtera plus cher par pièce. D'autre part, si on usine pour avoir le temps d'usinage minimum, on aura un coût d'usinage maximal et toute tentative pour augmenter le taux de production ne fera que le diminuer.

CHAPITRE 2

MODÉLISATION ET OPTIMISATION

Que ce soit par souci de simplicité ou parce que des outils informatiques adéquats n'étaient pas encore disponibles, aucun modèle général complet n'a été développé à notre connaissance jusqu'à maintenant pour les opérations de fraisage. Comme nous allons le voir plus loin, la résolution de ce modèle demande une puissance informatique importante. Les ordinateurs personnels disponibles aujourd'hui sont assez puissants pour résoudre ce problème dans un temps raisonnable (de 25 à 70 secondes). Avec les ordinateurs personnels disponibles en l'an 2000, il aurait fallu plus de dix minutes pour résoudre le modèle et en 1994, plus d'une heure. Seuls des ordinateurs très puissants dans les centres de recherche auraient pu résoudre, à l'époque, ce type de modèle en un temps raisonnable et ces outils n'étaient pas disponibles pour la majorité des ateliers d'usinage.

Une des contributions majeures de ce mémoire est d'utiliser une équation exhaustive de la durée de vie de l'outil qui va comprendre quatre des cinq variables de décision et une expression du fini de surface spécialement développée pour le fraisage en bout par Lou, Chen et Caleb (1999).

2.1 Modélisation

Les fonctions objectifs généralement utilisées dans la littérature et utiles aux gestionnaires sont le coût de production minimum, qui est un objectif strictement économique en $/unité, et le temps d'opération minimum (taux de production maximum), qui est un objectif de temps en minutes (ou unité/heure si on l'exprime en taux de production). Il peut être intéressant pour les gestionnaires de considérer l'une ou

l'autre de ces fonctions objectifs pour des raisons qui relèvent de la gestion des opérations. Il ne faut pas oublier qu'une opération de fraisage est la plupart du temps une opération parmi d'autres dans une chaîne de production et que les objectifs de gestion globaux ont besoin d'alternatives de la part des opérations. Ainsi, si l'opération de fraisage constitue un goulot d'étranglement dans le flux de la chaîne de production, on voudra le temps d'opération minimum afin d'augmenter le taux de production de la chaîne.

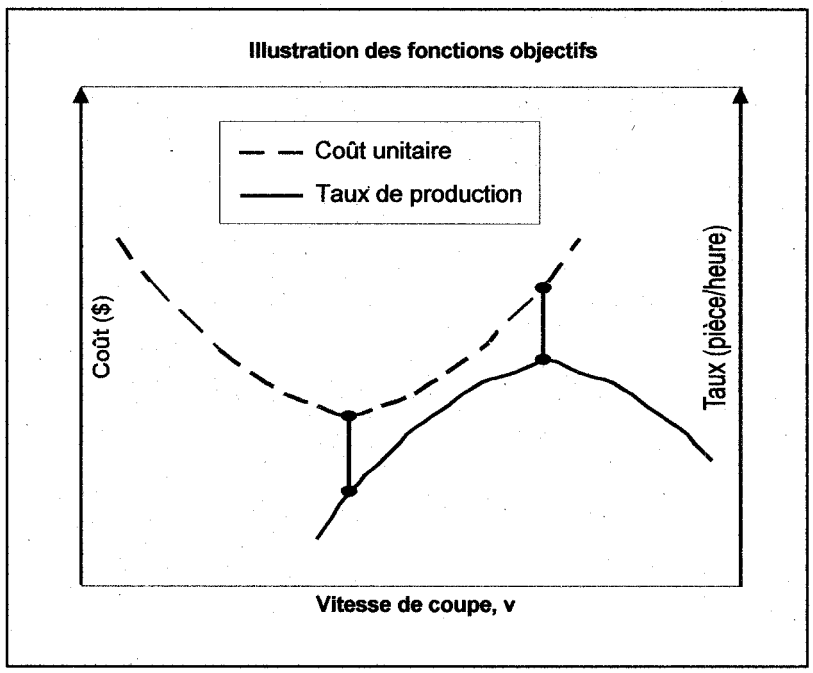

Figure 8 Les fonctions objectifs

D'autres part, si le temps de production n'est pas critique, on voudra utiliser le coût de production minimum afin de réduire les coûts d'opération. Ces deux derniers critères sont bien sûr en contradiction à l'intérieur des paramètres opérationnels d'une machine. La Figure 8 illustre sommairement le comportement des fonctions objectifs entre elles par rapport à la vitesse de coupe, une des variables de décision.

Nous allons utiliser beaucoup de symboles pour construire nos équations. La plupart d'entre eux sont des données qu'il faut recueillir sur la pièce à usiner, sur l'outil de coupe ou sur la machine qui est utilisée. Il y a aussi bien sûr les variables à déterminer et les différentes équations intermédiaires. La liste exhaustive des symboles que nous allons utiliser dans ce document est présentée dans la section « liste des abréviations et des sigles » (page ix). Dans tout ce document, les indices $_r$ et $_f$ représentent respectivement les symboles pour les passes d'ébauche et de finition.

2.1.1 Les fonctions objectifs

Le temps d'usinage est l'élément-clé des deux fonctions objectifs. Il comprend t_c, le temps de coupe, R/v_r, le temps d'approche et de retrait en avance rapide, t_L, le temps pour charger et enlever une pièce, t_s/ N_L, le temps de mise en course réparti à la pièce et t_c*t_e/ t_0, le temps pour changer un outil émoussé lui aussi réparti à la pièce.
Le temps d'usinage unitaire :

$$T_u = t_{cr} + t_{cf} + \frac{R}{v_t} + t_L + \frac{t_s}{N_L} + \frac{t_{cr} \cdot t_e}{t_{0r}} + \frac{t_{cf} \cdot t_e}{t_{0f}} \quad \text{(minutes)} \qquad (2.1)$$

Nous allons présenter les deux fonctions objectifs qui sont le coût d'usinage unitaire et le taux de production. Le coût d'usinage comprend : C_B, le coût du matériel brut, k_0, le coût d'opération, incluant salaire et machine, T_U, le temps d'usinage, C_H, le coût horaire de l'outil et t_c, le temps de coupe.

L'équation du coût d'usinage :

$$U = C_B + k_0 T_u + C_{Hr} t_{cr} + C_{Hf} t_{cf} \tag{2.2}$$

Le taux de production horaire quant à lui ne tient compte que du temps d'usinage unitaire :

$$P = \frac{60}{T_u} \quad \text{(unités/heure)} \tag{2.3}$$

Dans le reste de ce document, nous allons minimiser le temps d'usinage au lieu de maximiser le taux de production, parce que cela facilite l'implémentation et que les résultats sont les mêmes.

Nous allons détailler un peu les éléments des fonctions objectifs. Les temps de coupe sont définis par le volume à usiner et le débit du copeau ou d'enlèvement de matière. Le volume à usiner dépend de la géométrie de la pièce qui, dans notre exemple, est un prisme rectangulaire, et pour la passe d'ébauche, du volume qu'on prévoit enlever par la passe de finition. Le débit du copeau dépend des variables de décision : les largeurs de passe w_r et w_f, les profondeurs de passe d_r et d_f et de v_{fr} et v_{ff}, les vitesses d'avance. On présume que la longueur des flûtes de l'outil est suffisante pour usiner la surface verticale, s'il y en a une, en une seule passe.
Les temps de coupe se calculent ainsi :

$$t_{cr} = \frac{LWH - (W - w_f)Ld_f - HLw_f}{w_r d_r v_{fr}} \tag{2.4}$$

$$t_{cf} = \frac{WL}{D_f v_{ff}} \tag{2.5}$$

La durée de vie de l'outil est utilisée pour calculer la contribution du coût de l'outil et du temps de changement de l'outil émoussé. Le modèle empirique de Taylor (1907) est

généralement considéré comme adéquat pour calculer la durée de vie de l'outil. Il existe plusieurs versions de ce modèle. La plus simple est utilisée par B. White and A. Houshyar (1992).

Cette version ne tient compte que la vitesse de coupe :

$$vt_0^n = k$$

Al-Ahmari (2001) et Malakooti and Deviprasad (1989) ont utilisé le modèle le plus général qui s'applique à une opération de tournage. Cette expression tient compte de la vitesse d'avance et de la profondeur de passe en plus de la vitesse de coupe :

$$t_0 = \frac{k}{v^\alpha f^\beta d^r}$$

Comme nous modélisons une opération de fraisage en bout, il faut étendre encore l'expression du modèle. Nous allons donc ajouter la largeur de passe radiale afin d'avoir un modèle général convenable pour le fraisage en bout :

$$t_0 = \frac{k}{v^\alpha f^\beta d^r w^\lambda}$$

où k,α, β, γ et λ sont des constantes. Il y a deux équations en réalité, une pour l'ébauche et une pour la finition, soit t_{0r} et t_{0f}.

$$t_{0r} = \frac{k}{v_r^{\ \alpha} f_r^{\ \beta} d_r^{\ r} w_r^{\ \lambda}} \tag{2.6}$$

$$t_{0f} = \frac{k}{v_f^{\ \alpha} f_f^{\ \beta} d_f^{\ r} w_f^{\ \lambda}} \tag{2.7}$$

Ce modèle nécessite une expérimentation pour déterminer les constantes d'abord parce qu'il s'agit d'un modèle empirique et ensuite à cause des conditions d'usinage tel que les matériaux de la pièce et de l'outil, le fluide de coupe, la rigidité du montage et les

vibrations qui varient beaucoup d'une machine à l'autre. Un plan d'expérience statistique peut être utilisé pour déterminer ces constantes.

Les coûts horaires des outils, C_{Hr} et C_{Hf}, sont calculés en utilisant les durées de vies t_{0r} et t_{0f}, le coût d'achat initial C_p, le coût d'aiguisage C_A, et le nombre de fois qu'il peut être aiguisé N_A :

$$C_{Hr} = \frac{C_{pr} + (C_A \cdot N_{Ar})}{t_{0r}(N_{Ar} + 1)} \tag{2.8}$$

$$C_{Hf} = \frac{C_{pf} + (C_A \cdot N_{Af})}{t_{0f}(N_{Arf} + 1)} \tag{2.9}$$

2.1.2 Les contraintes

Les principales contraintes de notre modèle sont les finis de surface, la puissance de la machine et la force maximale admissible de l'opération.

Dans une opération de fraisage en bout, la surface de coupe axiale n'est pas générée par un rayon d'outil comme dans une opération de tournage ou de surfaçage. Il n'existe pas d'équation fondamentale adéquate pour exprimer le fini de surface en bout, mais Lou, Chen et Caleb (1999) ont démontré que l'on peut utiliser une relation empirique en déterminant les paramètres à l'aide d'un plan d'expérience :

$$\begin{aligned} & X_0 + X_1 \cdot v_{ff} - X_2 \cdot N_f \cdot v_{ff} \\ & \quad + X_3 \cdot N_f \cdot d_f - X_4 \cdot v_{ff} \cdot d_f \leq R_{ae} \end{aligned} \tag{2.10}$$

où X_0, X_1, X_2, X_3 et X_4 sont des facteurs à déterminer à l'aide d'un plan d'expérience. C'est, à notre connaissance, la seule expression du fini de surface qui soit bien adaptée au fraisage en bout.

Pour le fini de surface radiale ou en périphérie de l'outil, étant donné que l'on peut facilement modéliser la fraise en bout comme un cylindre en déplacement (voir la Figure 3, à la page 7 du chapitre 1), nous utiliserons l'équation présentée par Tolouei-Rad et Bidhendi (1997) qui ne dépend que de l'avance par dent et du diamètre de l'outil :

$$\frac{318 f_f}{4 D_f} \leq R_{ap} \tag{2.11}$$

Les finis de surfaces, R_{ae}, et R_{ap}, sont les rugosités moyennes, exprimés en micromètres (μm).

La puissance requise dépend du débit du copeau et de la puissance spécifique du matériau de la pièce :

$$P_r = \frac{P_s Z_w}{\eta}$$

La puissance maximale de la machine détermine le débit maximal de matière qui peut être enlevée d'une pièce :

$$Z_w \leq \frac{P_{\max} \eta}{P_s}$$

Le débit est lié à la profondeur et la largeur de passe et à la vitesse d'avance ce qui nous donne finalement deux contraintes, une pour l'ébauche et une pour la finition :

$$d_f \cdot w_f \cdot v_{ff} \leq \frac{P_{\max} \eta}{P_s} \tag{2.12}$$

$$d_r \cdot w_r \cdot v_{fr} \leq \frac{P_{\max} \eta}{P_s} \tag{2.13}$$

La puissance maximale n'est pas suffisante pour décrire la physique de l'opération. En regardant cette contrainte de plus près, on remarque que rien n'empêche le fait que pour une même puissance, il se peut que la vitesse soit très faible et l'avance ainsi que la profondeur très grandes, ce qui pose des problèmes en pratique. Il convient donc d'ajouter une contrainte de force. Compte tenu de la rigidité du montage de la pièce et de l'outil, la force maximale est celle à laquelle l'opération peut être effectuée de manière sécuritaire. Les équations de force sont facilement calculées à partir des équations de puissance requise :

$$\frac{60 \cdot P_r}{v_r} \leq F_{r\max} \tag{2.14}$$

$$\frac{60 \cdot P_r}{v_f} \leq F_{f\max} \tag{2.15}$$

La largeur de passe radiale est une variable de décision et est reliée au nombre de passes radiales et à la largeur du volume à usiner moins la passe de finition :

$$w_r = \frac{W - w_f}{N_w}$$

Ce qui nous donne la contrainte :

$$w_r \cdot N_w + w_f = W \tag{2.16}$$

De même, la profondeur de la passe axiale est aussi une variable de décision et est reliée au nombre de passes axiales et à la hauteur du volume à usiner moins la passe de finition :

$$d_r = \frac{H - d_f}{N_p}$$

Ce qui nous donne la contrainte :

$$d_r \cdot N_p + d_f = H \tag{2.17}$$

Les largeurs de passe radiale sont limitées par le diamètre de l'outil :

$$w_r \leq D_r \tag{2.18}$$

$$w_f \leq D_f \tag{2.19}$$

Les profondeurs de passe axiale sont limitées par la longueur des flûtes de l'outil :

$$d_r \leq h_{r\max} \tag{2.20}$$

$$d_f \leq h_{f\max} \tag{2.21}$$

Les vitesses d'avance sont limitées par la vitesse d'avance maximale de la machine :

$$v_{fr}, v_{ff} \leq v_{f\max} \tag{2.22}$$

Les vitesses de coupe et de rotation de la broche dépendent essentiellement du diamètre de l'outil :

$$v = \frac{\pi \cdot D \cdot N}{1000} \quad \Rightarrow \quad N = \frac{1000 \cdot v}{\pi \cdot D}$$

Les vitesses de coupe sont limitées par la vitesse de rotation maximale de la machine :

$$\frac{1000 \cdot v_r}{\pi \cdot D_r} \leq N_{\max} \tag{2.23}$$

$$\frac{1000 \cdot v_f}{\pi \cdot D_f} \leq N_{\max} \tag{2.24}$$

Finalement, il y a les contraintes de non-négativité sur les variables de décision :

$$v_r, f_r, d_r, w_r, D_r, v_f, f_f, d_f, w_f, D_f \geq 0 \tag{2.25}$$

2.2 Optimisation

On peut facilement remarquer que notre modèle d'optimisation présenté est hautement non linéaire. Les variables de décision se multiplient entre elles autant dans les fonctions objectifs que dans les contraintes. Il y a aussi plusieurs exposants différents sur les variables de décision à cause du modèle de durée de vie de l'outil de Taylor. Les méthodes de base en optimisation : résolution graphique, programmation linéaire, quadratique ou dynamique, énumération implicite ou résolution algébrique, ne sont pas applicables ou alors difficilement applicables à ce modèle. Il ne reste donc que l'énumération exhaustive, les heuristiques et les techniques numériques pour résoudre ce problème. On peut bien sûr s'en remettre à un logiciel d'optimisation commercial avec les avantages et les inconvénients que ceux-ci apportent. Nous avons choisi de développer un algorithme génétique (AG) de base capable de résoudre ce problème dans un temps raisonnable.

2.2.1 Un algorithme génétique

Nous avons choisi d'utiliser un algorithme génétique pour plusieurs raisons. Premièrement, quand la conception de l'algorithme est faite, on a la possibilité d'utiliser n'importe quel langage informatique ou mathématique pour l'implantation. Un logiciel

d'optimisation spécifique n'est pas nécessaire. Ensuite, ce type d'algorithme est reconnu pour être très robuste à l'utilisation. Cette robustesse se traduit principalement par deux caractéristiques intéressantes : il n'est pas nécessaire de fournir un point de départ pour démarrer l'optimisation et l'algorithme résiste assez bien au blocage dans des extremums locaux. Une autre caractéristique intéressante de l'AG est qu'il améliore les solutions en cours d'optimisation. Plus on lui donne de temps, plus il s'approche de l'optimum. Mais si on a besoin d'une réponse, même s'il n'a pas eu le temps de converger, il est généralement capable de nous fournir une solution valide, même si elle n'est pas optimale, ce qui vaut toujours mieux que ne pas avoir de réponse du tout.

Les algorithmes génétiques sont inspirés de la théorie de l'évolution de Charles Darwin (1859). Ces algorithmes génèrent une population de solutions et les font évoluer en favorisant la survie et la reproduction des solutions qui ont le plus de chances de converger vers l'optimum.

La tâche la plus importante pour l'élaboration d'un AG est la détermination du génome. Ce génome est en général un vecteur de valeurs (binaires, entières, réelles, etc.) contenant la représentation des variables de décision. En fait, chaque individu contient une solution plus ou moins faisable selon les contraintes et plus ou moins performante selon la ou les fonctions objectifs.

Une solution au problème décrit au début du chapitre doit contenir les dix variables de solution, les valeurs des fonctions objectifs U et Tu, la valeur de la fonction d'adaptation « Fit » (expliquée plus loin) et une étiquette d'identification « # ID » qui permet de retrouver une solution dans une liste. Nous avons donc besoin d'un vecteur de 14 éléments. Le
Tableau I illustre les variables à l'intérieur du vecteur représentant le génome.

Tableau I

Le génome

0	1	2	3	4	5	6	7	8	9	10	11	12	13
# ID	Fit	U	Tu	D_r	D_f	v_r	v_f	f_r	f_f	d_r	d_f	w_r	w_f

Les valeurs de la fonction d'adaptation et des fonctions objectifs sont incluses dans le génome afin de simplifier la structure de donnée du programme. Quand des calculs à partir des variables de décision ou des modifications sont requis sur celles-ci, on adresse le vecteur sur les positions 4 à 13 seulement.

La valeur de la fonction d'adaptation (fitness) est un réel compris entre 0 et 1. Plus la valeur est proche de 1, plus l'individu est performant. La section 2.2.2 explique ce calcul en détail.
Nous avons fait l'implémentation de cet algorithme en langage C++, ce qui explique le fait que nous avons numéroté les éléments du vecteur de 0 à 13, mais cet algorithme peut être programmé avec n'importe quel langage informatique ou mathématique, comme Matlab® par exemple. Quand le génome est bien conçu, on peut passer au design des étapes de l'algorithme.

L'algorithme génétique de base comprend sept étapes. La Figure 9 illustre schématiquement l'algorithme. Pour commencer l'algorithme, on génère une population initiale aléatoirement.

Dès l'initialisation, on rejette les individus qui sont trop loin hors contraintes, mais on tolère les individus qui ne sont pas trop hors contraintes parce que ceux-ci peuvent être près de la solution optimale. Notez que la taille de la population donne la variété de celle-ci et aide à ce que l'algorithme ne se bloque pas dans un optimum local. Plus la

taille est grande, plus les itérations demandent de temps de calcul, il convient donc de bien ajuster ce paramètre.

À la sélection, on choisit des couples parmi la population qui vont être utilisés pour la reproduction. La sélection est aléatoire, mais les individus qui ont une plus grande valeur d'adaptation ont plus de chances d'être choisis et donc, de se reproduire. Les individus les plus performants sont donc favorisés pour la reproduction.

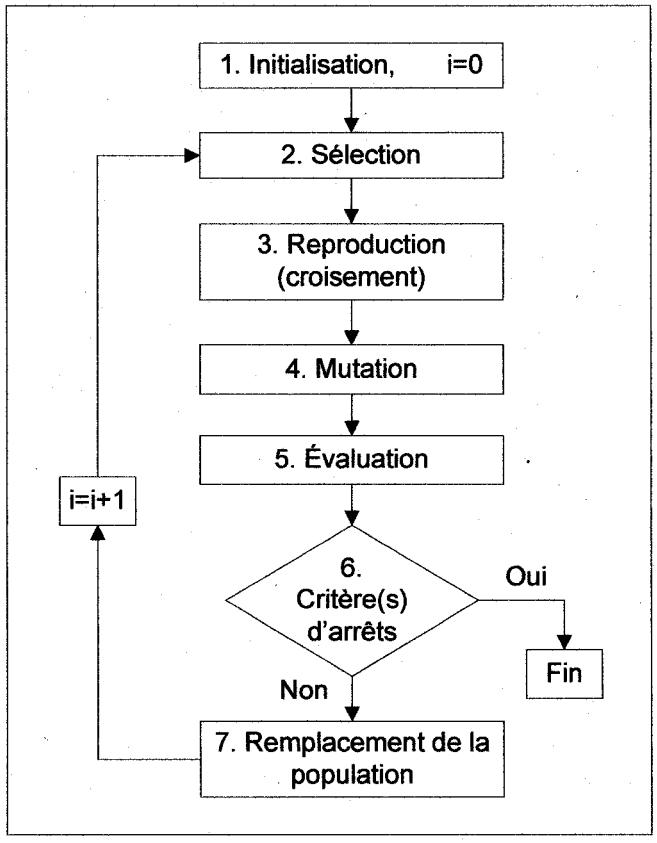

Figure 9 Schéma algorithmique

La reproduction, ou croisement, permet de mélanger le génome des parents pour produire la prochaine génération. Pour chaque variable du génome, on fait une moyenne

pondérée des valeurs des deux parents. La pondération est aléatoire alors la variable calculée pour l'enfant peut tenir d'un parent plus que l'autre.

À la mutation, comme dans la nature, on introduit une perturbation d'une ou plusieurs variables de l'enfant produit au croisement. Le nombre d'enfants subissant la mutation ainsi que le nombre de variables à muter et l'amplitude des mutations sont des paramètres à ajuster pour améliorer la performance de l'algorithme.

On évalue ensuite la valeur des fonctions objectifs et de la fonction d'adaptation de chaque enfant produit. La vérification des critères d'arrêt détermine si on arrête l'algorithme ou si on fait une autre itération. Si aucun des critères n'est atteint, on fait une autre itération à partir de l'étape de la sélection. Il y a trois critères d'arrêt que l'on peut ajuster, deux absolus et un relatif. Les critères absolus sont le temps de calcul et le nombre d'itération maximum que l'on donne à l'algorithme pour résoudre le problème. Le critère relatif est le nombre maximum d'itération que l'on fait sans qu'il n'y ait d'amélioration de la meilleure solution. En effet, si la meilleure solution ne s'améliore pas pendant un certain nombre d'itération, c'est que l'algorithme a convergé vers une solution. Celle-ci est généralement très près de l'optimum.

2.2.2 La fonction d'adaptation

La fonction d'adaptation est l'élément qui nous permet d'orienter la population vers le but que l'on fixe, c'est-à-dire la résolution du problème. C'est un élément qui peut avoir l'air d'un fourre-tout, mais qui est en fait très puissant. Globalement, la fonction d'adaptation contient la performance de la solution sur le plan de la fonction objectif par rapport au reste de la population et une pénalité qui s'occupe de gérer le respect des contraintes. Dans notre implantation, cette fonction prend la forme suivante :

$$F = P \cdot C + (1 - P) \cdot O \tag{2.26}$$

Avec F : Fonction d'adaptation

P : Poids des contraintes

C : Valeur d'adaptation selon les contraintes

O : Valeur d'adaptation selon la fonction objectif

Où le facteur P représente l'importance que l'on donne au respect des contraintes. On peut ajuster le facteur P pour améliorer la performance de l'algorithme. Le terme O est égal au rapport du minimum rencontré dans la population sur la valeur de la fonction objectif pour l'individu.

$$O = \frac{Min\,U}{U} \quad ou \quad \frac{Min\,Tu}{Tu} \tag{2.27}$$

Si l'individu est le meilleur de la population au moment de l'évaluation, O=1, sinon, O<1.

Le terme C représente le respect des contraintes et est calculé comme suit :

$$C = \prod \left\{ Min\left(\frac{V}{L}, \frac{L}{V}\right) \right\} \tag{2.28}$$

pour toutes les contraintes.

Avec V : Valeur de la contrainte pour l'individu

L : Limite de la contrainte (min ou max)

Si l'individu respecte toutes les contraintes, C=1, sinon, C<1.

Après le calcul de la fonction d'adaptation, si sa valeur pour un individu est égale à 1, cet individu produit le minimum de la fonction objectif et respecte toutes les contraintes.

CHAPITRE 3

RÉSOLUTION ET ANALYSES

Nous allons commencer ce chapitre en décrivant comment nous avons programmé le logiciel d'optimisation, pour ensuite présenter un exemple numérique. On va continuer en analysant le comportement de notre modèle lorsqu'on fait varier différents paramètres. Finalement, nous allons faire une analyse de sensibilité qui va nous éclairer un peu plus sur l'impact des contraintes sur les fonctions objectifs.

3.1 L'implantation

Nous avons utilisé le logiciel Microsoft Visual C++® pour implanter notre algorithme génétique. Plusieurs raisons ont motivé ce choix. Premièrement, l'auteur avait déjà au départ une expérience avec ce langage de programmation ce qui limitait l'apprentissage nécessaire à l'utilisation d'un nouveau langage. Ensuite, ce langage permet de compiler et donc de produire des applications autonomes qui s'exécutent dans un environnement Microsoft Windows®. Ceci apporte deux avantages. Premièrement, comme le programme est compilé, il s'exécute beaucoup plus rapidement qu'un programme dans un environnement d'interprétation de commande. Deuxièmement, le logiciel de recherche que nous développons pourra éventuellement être converti en version commercialisable s'il y a une demande de la part de l'industrie. Le listage complet du code que nous avons développé est présenté en annexe 2. Bien sûr, comme le développement d'un logiciel commercial n'est pas le but premier de ce travail, le logiciel que nous avons développé n'est pas pourvu d'une interface utilisateur complète et facile à utiliser. Nous avons seulement créé les boîtes de dialogue qui permettent de modifier les paramètres de l'algorithme et du modèle que nous avions à modifier fréquemment. Les paramètres qui ne figurent pas dans les boîtes de dialogue doivent être modifiés directement dans le code du programme.

3.1.1 Interface et fonctionnement du logiciel

Au démarrage du logiciel, il y a seulement un écran d'accueil et rien ne se passe tant qu'on n'active pas les boîtes de dialogue. Il y un menu « optimisation » qui contient deux options : « modèle » et « optimiser ».

L'option « modèle » ouvre la boîte de dialogue qui permet de modifier les paramètres de l'opération et de la machine, tandis que l'option « optimiser » ouvre la boîte de dialogue qui permet de modifier les paramètres de l'algorithme et le choix des fonctions objectifs à optimiser.

La boîte de dialogue des paramètres de l'opération et de la machine, illustrée à la Figure 10, contient en fait les valeurs limites des contraintes principales. La section plage des variables permet d'ajuster les valeurs minimales et maximales que peuvent prendre les variables.

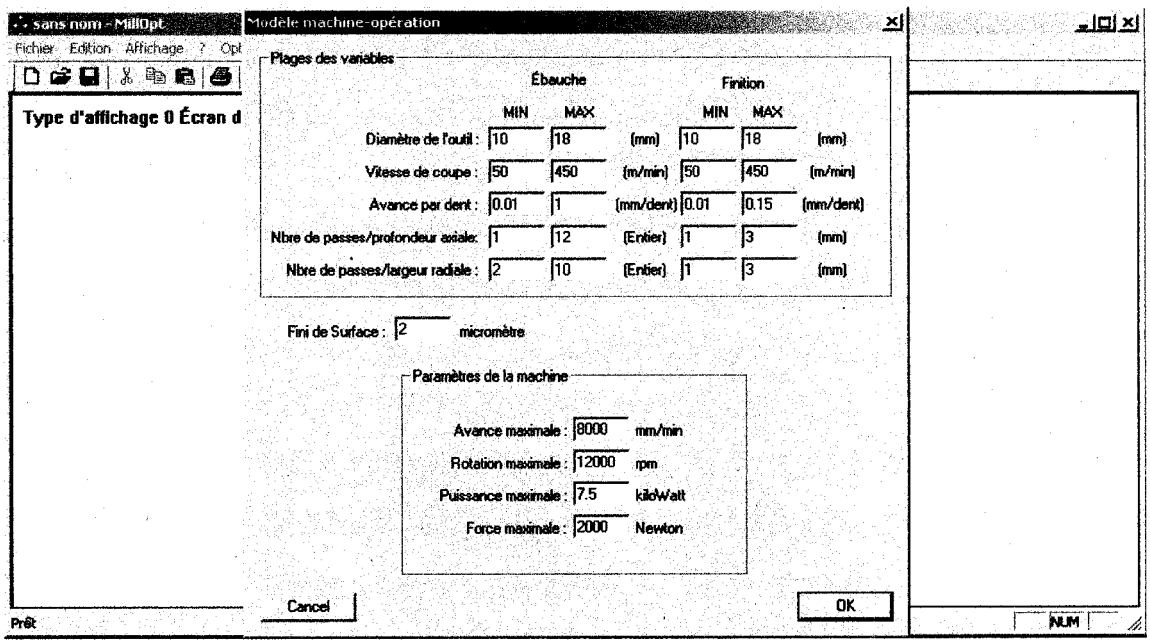

Figure 10 Les paramètres de l'opération et de la machine

Notez que l'on peut fixer des variables ici en mettant la même valeur pour le minimum et le maximum. Si on dispose par exemple d'un lot d'outils de 15 mm et qu'on désire les utiliser pour l'opération, on n'a qu'à mettre 15 mm comme minimum et comme maximum du diamètre de l'outil et l'algorithme va optimiser les autres variables en respectant cette contrainte. La section paramètres de la machine rassemble les contraintes qui concernent les performances de la machine. Il convient de bien ajuster ces valeurs, en particulier la puissance et la force maximale puisque ce sont les contraintes qui influencent le plus l'optimum.

La force maximale qu'il faut considérer ici n'est pas directement la capacité de la machine, mais dépend aussi de la rigidité du montage ou du gabarit qui immobilise la pièce. Lorsqu'on ferme cette boîte de dialogue, rien ne se passe, mais les modifications des valeurs sont mises à jour si on clique sur « OK » et sont annulées si on clique sur « Cancel ».

La Figure 11 illustre la boîte de dialogue des paramètres de l'algorithme et le choix des fonctions objectifs à optimiser. Les paramètres de l'AG permettent d'ajuster les performances de l'algorithme. Plus la taille de la population est grande et plus il y aura de la variété dans la population ce qui contribue à ce que l'algorithme ne reste pas bloqué dans un optimum local. Par contre, plus la taille de la population est grande et plus cela prend de temps pour effectuer une itération. Les mutations permettent deux choses : explorer l'espace de solution et raffiner les solutions pour les rapprocher de l'optimum. Il faut avoir un certain taux de mutation qui permet de converger vers l'optimum, surtout vers la fin de l'optimisation. Cependant, si le taux de mutation est trop important, les solutions vont se retrouver trop souvent hors contraintes et seront rejetées, annulant ainsi la contribution des mutations. Il y a trois critères d'arrêt, le nombre de générations maximum, le temps maximum et le pourcentage d'amélioration.

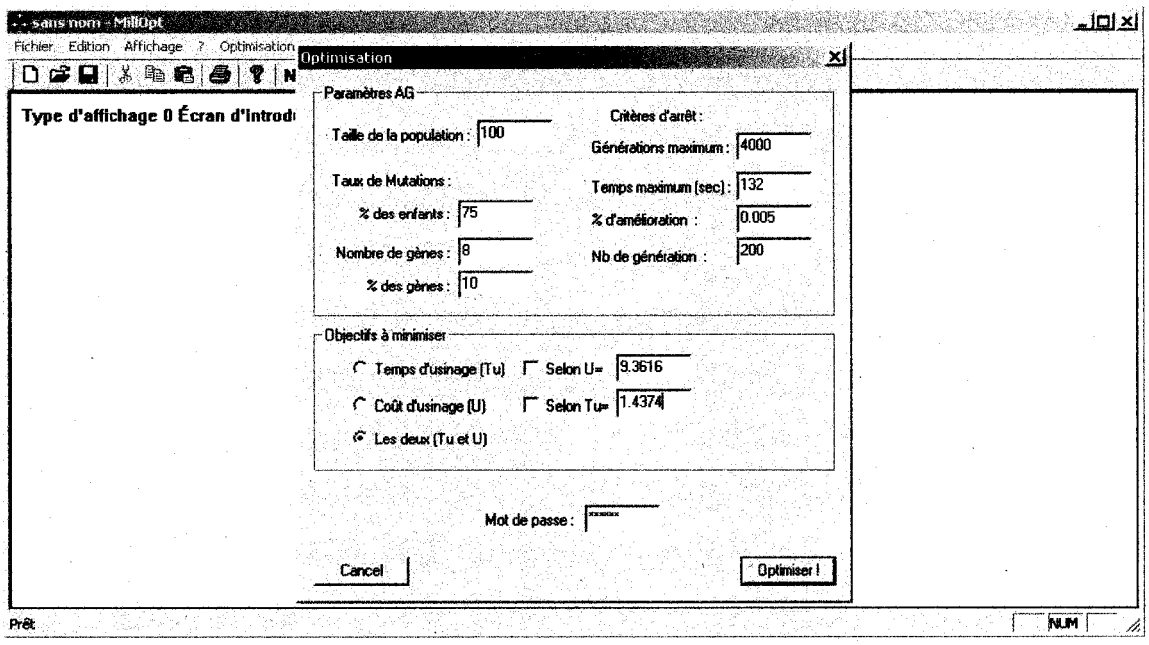

Figure 11 Les paramètres de l'algorithme et les fonctions objectifs

Le pourcentage d'amélioration est un critère relatif qui permet de détecter si l'algorithme a convergé vers une solution. En effet, si l'algorithme n'arrive pas à améliorer la meilleure solution pendant un certain nombre de générations, on peut considérer qu'il a convergé à l'optimum. La section objectif à minimiser permet de sélectionner les fonctions objectifs que l'on veut minimiser, soit le temps d'usinage minimum, le coût d'usinage minimum ou les deux. Si on choisit un seul objectif, une option supplémentaire est disponible : on peut minimiser un objectif en fonction d'une valeur précise de l'autre objectif. Par exemple, on peut minimiser le coût d'usinage pour un certain temps d'usinage donné. Il n'y a pas de bouton « OK » dans cette boîte de dialogue, mais un bouton « Optimiser! ». En appuyant sur ce bouton, l'optimisation commence immédiatement et l'écran d'initialisation s'affiche.

Lors de l'initialisation, la population initiale est générée aléatoirement. L'initialisation prend un certain temps parce que l'on calcule le respect des contraintes pour chaque individu et on rejette ceux qui sont trop loin hors contraintes. On tolère toutefois une certaine violation des contraintes parce que l'on sait qu'il y en aura une ou plusieurs

actives et que le croisement et la mutation vont guider les générations suivantes vers les limites des contraintes actives. Quand l'initialisation est complétée, les itérations de l'algorithme commencent et la fenêtre principale du logiciel affiche le déroulement de l'optimisation comme l'illustre la Figure 12.

En haut de l'écran, on affiche le nombre de générations et le temps qui s'est écoulé depuis le début de l'optimisation. Ensuite, on affiche différentes informations sur le déroulement de l'opération :

FitMoy : La moyenne des valeurs de fonction d'adaptation rencontrée dans la population. Plus la population est performante plus cette valeur tend vers 1.

Deltafit : La différence entre la plus haute et la plus petite valeur de fonction d'adaptation rencontrée dans la population. Plus la différence tend vers 0, plus la population est homogène.

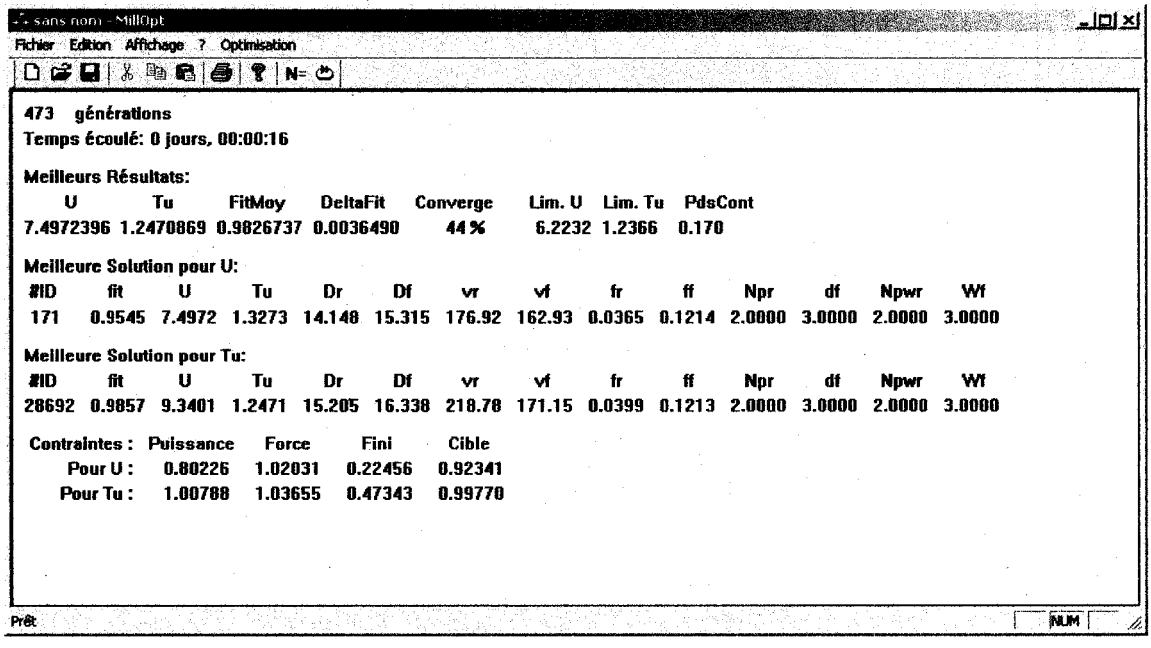

Figure 12 Le déroulement de l'optimisation

Converge : Le pourcentage de générations (par rapport au maximum fixé) qui se sont déroulées sans amélioration de la meilleure solution.
Lim. U et **Lim. Tu** : Les limites inférieures qui permettent de calculer la performance d'une solution selon les fonctions objectifs.
PdsCont : Le poids des contraintes dans le calcul de la fonction d'adaptation.

Au centre de l'écran, on affiche la meilleure solution qui a été générée pour chacune des fonctions objectifs. Notez que l'algorithme commence par minimiser le temps d'usinage, mais on garde au passage la meilleure solution trouvée pour le coût d'usinage.

Finalement, en bas de l'écran, on affiche le respect des contraintes principales des solutions affichées plus haut. Le respect des contraintes est affiché en pourcentage, c'est-à-dire que si la solution est à la limite d'une contrainte, le nombre tend vers 1. Une valeur inférieure à 1 signifie que la contrainte n'est pas active et si elle est supérieure à 1, c'est que la contrainte n'est pas respectée. La contrainte notée « Cible » illustre la différence, elle aussi en pourcentage, entre la valeur de l'objectif et celle qui est demandée quand on minimise un objectif selon une valeur précise de l'autre objectif. En plus de l'information affichée à l'écran, à la fin de l'optimisation, le logiciel transfère les solutions optimales vers un fichier texte Excel®. On peut ensuite traiter ces solutions à notre guise.

3.1.2 Fonctionnement interne du logiciel

L'environnement de développement de Microsoft Visual C++® comprend principalement les classes et les ressources. Dans les ressources on fait la conception des boîtes de dialogue et des menus. C'est dans les classes que l'on ajoute le code du programme. Nous utilisons principalement deux classes : la classe *view* et la classe *document*. La classe *view* gère l'affichage, les entrées utilisateur (clavier et souris,

principalement) et les minuteurs. La classe *document* gère le contenu et les transferts des variables et contient le code des différentes fonctions. Nous n'allons pas commenter le code ligne par ligne ici, mais plutôt expliquer le fonctionnement général. Au moment où l'utilisateur clique sur le bouton « Optimiser! », un minuteur est déclenché dans la classe *view*. Ce minuteur envoie un signal périodique (plusieurs fois par seconde dans notre cas) qui permet de déclencher des événements. Ce minuteur est nécessaire pour afficher des informations dans la fenêtre principale du logiciel pendant que l'optimisation s'exécute. En effet, le logiciel ne peut pas mettre à jour la fenêtre principale pendant qu'il exécute une fonction dans la classe *document*. Alors, si les itérations sont seulement des boucles dans une fonction, le logiciel a l'air bloqué pendant qu'il optimise et on ne peut pas visualiser l'évolution. À chaque fois qu'un signal est envoyé par le minuteur, une des deux fonctions principales de la classe *document* est exécutée. Ces deux fonctions principales sont Initialise() et Optimise(). Pour commencer, c'est la fonction Initialise() qui s'exécute et génère aléatoirement un candidat adéquat pour la population initiale. Quand la population initiale est complète, c'est la fonction Optimise() qui s'exécute. Cette fonction contient toutes les étapes pour faire une itération complète de l'algorithme décrit à section 2.2.

3.1.3 Vecteurs et matrices de taille variable

Une grande quantité de variables, vecteurs et matrices sont nécessaires à la réalisation d'un logiciel qui exécute l'algorithme d'optimisation. Nous n'allons pas les décrire ici en détail, mais plutôt inviter le lecteur intéressé à consulter le listing du code qui figure en annexe 2. Il faut cependant noter que nous avons utilisé une classe de vecteurs et de matrices dont la particularité est d'être de tailles variables. Nous avons tiré ces classes de *Numerical recipes in C++*. Mentionnons que lorsqu'une matrice ou un vecteur est déclaré en C++, il faut mentionner sa taille, ce qui permet au compilateur de réserver la mémoire nécessaire pendant l'exécution du logiciel. Dans notre logiciel, la taille de

certaines matrices dépend de paramètres de l'algorithme que l'on doit ajuster, comme la taille de la population par exemple. Alors, on a deux choix quant à l'implantation de cette caractéristique : soit on fixe une taille maximale et l'utilisateur doit entrer une valeur inférieure à celle-ci, soit on utilise une classe de vecteurs et de matrices à tailles variables qui s'ajuste aux besoins de l'utilisateur. Si on fixe simplement une taille maximale, cela implique deux désavantages majeurs. Premièrement, l'utilisateur est limité par la taille maximale qui lui est imposée, deuxièmement, le logiciel ne va utiliser réellement qu'une partie de la mémoire réservée, le reste étant réservé pour rien. Pour ces raisons, nous avons choisi d'utiliser une classe de matrices de tailles variables qui ne va utiliser que la mémoire nécessaire en fonction des paramètres requis par l'utilisateur.

3.1.4 Description des fonctions de la classe *view*

Il y a plusieurs fonctions programmées dans le code du logiciel, certaines sont des routines utilitaires, d'autres servent à réaliser l'algorithme. Il y a aussi des fonctions qui assurent le bon fonctionnement dans l'environnement Windows®. Ces fonctions sont générées automatiquement par le logiciel de programmation et nous n'allons pas en tenir compte. Il y a quatre fonctions dans la classe *view* et 19 fonctions dans la classe *document*.

Comme nous l'avons déjà mentionné, la classe *view* s'occupe de l'affichage dans la fenêtre principale, des minuteurs et des événements Windows comme les mouvements et les clics de la souris par exemple. Voici la liste et une brève description des fonctions de la classe *view*.

void OnDraw(CDC pDC)*
Cette fonction réalise l'affichage dans la fenêtre principale de l'application et est appelée à rafraîchir celle-ci chaque fois que des informations sont mises à jour ou que de

nouvelles informations sont disponibles sur le déroulement de l'algorithme. À chaque passage dans cette fonction, on vérifie s'il faut démarrer ou arrêter le minuteur qui va s'exécuter pour toute la durée de l'optimisation. Le minuteur est piloté par une variable d'état qui est activée à la fermeture de la boîte de dialogue des paramètres de l'algorithme et est désactivée quand un critère d'arrêt est atteint.

void OnRButtonDown(UINT nFlags, CPoint point)
Si l'utilisateur clique le bouton de droite de la souris, l'optimisation s'interrompt. C'est la seule façon d'arrêter l'algorithme avant que les critères d'arrêt soient atteints.

void OnTimer(UINT nIDEvent)
C'est la fonction qui s'exécute à chaque fois que le minuteur envoie un signal. Cette fonction appelle alors une des deux fonctions principales de la classe *document* soit : *Initialise()* lors de l'initialisation ou *Optimise()* si l'initialisation est terminée. C'est par cette fonction que le programme passe de la classe *view* à la classe *document*. Cette alternance permet de faire la mise à jour de l'affichage par la fonction *OnDraw()*.

CString SetTrailingZeros(char str[], int digits)
Cette routine utilitaire s'occupe de standardiser l'affichage texte des nombres à une quantité déterminée de caractères numériques. Cette standardisation permet un affichage plus clair des informations dans la fenêtre principale.

3.1.5 Description des fonctions de la classe document

La classe *document* contient les variables globales, les deux fonctions qui transfèrent l'information des boîtes de dialogue, les dix fonctions qui servent à l'algorithme ainsi que cinq fonctions utilitaires. Voici la liste et une brève description des fonctions de la classe *document* :

Les deux fonctions qui transfèrent l'information des boîtes de dialogue :
void OnOptimisationModle()

Cette fonction est appelée par la boîte de dialogue du modèle et transfère les informations qui y sont saisies si on clique sur le bouton <OK> et annule les changements si on clique sur <Annuler>.

void OnOptimiser()
Cette fonction est appelée par la boîte de dialogue du modèle des paramètres de l'algorithme et transfère les informations qui y sont saisies si on clique sur le bouton <Optimiser!>. La variable d'état qui pilote le minuteur est aussi activée à la fermeture de la boîte de dialogue. Si on clique sur <Annuler>, on annule les changements fait dans la boîte de dialogue et on revient à la fenêtre principale sans démarrer l'optimisation.

Les dix fonctions de l'algorithme :
void Initialise()
Cette fonction est appelée par la fonction *OnTimer()* de la classe *view* au début de l'optimisation. À chaque exécution, elle appelle la fonction *InduviduAlea()* qui retourne un individu généré complètement aléatoirement. On vérifie ensuite si l'individu n'est pas trop loin hors contraintes et on le garde si c'est le cas. Si l'individu est trop loin hors contraintes, on en génère un autre. Quand la population initiale est complètement générée, une variable d'état signale à la fonction *OnTimer()* de la classe *view* d'appeler la fonction *Optimise()*.

void Optimise()
C'est la fonction principale qui pilote toute l'optimisation. Elle est appelée à chaque itération de l'algorithme. Au début de la fonction, on commence par vérifier si les critères d'arrêt sont atteints. Si c'est le cas, on met fin à l'optimisation. Sinon, chacune

des étapes de l'algorithme est exécutée, soit la sélection où l'on choisit les parents pour la reproduction, la reproduction où l'on mélange les génomes des parents pour créer les enfants, la mutation où l'on introduit des perturbations aléatoires au génome des enfants, l'évaluation où l'on calcule la valeur de la fonction d'adaptation des enfants et finalement, le remplacement de la population ou l'on ne garde que les meilleurs individus. À la fin de l'itération, le contrôle retourne à la classe *view* où l'affichage est mis à jour.

void InduviduAlea(Vec_IO_SP &individu)

Cette fonction génère un individu aléatoirement en appelant la fonction alea() selon le type de variable et les valeurs maximums et minimums de chaque variable. On appelle ensuite la fonction Set_U_P_Fit() afin de calculer les valeurs des fonctions objectifs.

void Set_U_P_Fit(Vec_IO_SP &individu)

Cette fonction calcule les valeurs des fonctions objectifs et complète ainsi les informations de base manquantes dans le génome de l'individu.

double SetFitCont(Vec_IO_SP individu)

Cette fonction retourne la valeur de la fonction d'adaptation selon les contraintes. Rappelons que cette valeur est égale à 1 si la solution est à l'intérieur des contraintes et entre 0 et 1 si ce n'est pas le cas.

void SetPopFitness(Mat_IO_SP &Population)

Cette fonction calcule ou recalcule la valeur de la fonction d'adaptation selon les valeurs minimales (limites inférieures) des fonctions objectifs. On doit recalculer ces valeurs à chaque itération, puisque les limites inférieures sont susceptibles de changer lors du remplacement de la population.

void Set_minU_minTu(Mat_IO_SP Population)

Cette fonction détermine les valeurs minimales des fonctions objectifs rencontrées dans la population. Elle est utilisée pour mettre à jour les limites inférieures dans la fonction SetLimitInf() et est appelée à différentes reprises au cours de l'itération.

void CheckVarMinMax(Vec_IO_SP &individu)

Cette fonction vérifie que les variables d'un individu ne sont pas à l'extérieur de leurs limites et les ramène à la limite si c'est le cas. Ces limites sont déterminées par l'utilisateur dans la boîte de dialogue des paramètres de l'opération et de la machine.

void SetContAct()

Cette fonction détermine lesquelles des contraintes principales (puissance, force et fini de surface) sont actives dans les meilleures solutions trouvées. Cette information n'est actuellement pas utilisée dans le logiciel, mais a été utilisée pour les tests en cours de développement et pourrait être utile dans de futurs développements du logiciel.

void SetLimitInf()

Cette fonction fixe les limites inférieures des fonctions objectifs, c'est-à-dire la plus petite valeur rencontrée dans la population sans tenir compte du respect des contraintes. Cette limite sert de référence pour calculer la valeur de la fonction d'adaptation selon les fonctions objectifs.

La classe *document* contient aussi les cinq fonctions utilitaires :

DP NR::ran1(int &idum)

Cette fonction tirée de l'ouvrage *Numerical recipes in C++* permet de générer des nombres aléatoires de distribution uniforme. On doit utiliser un bon générateur quand on a un besoin intensif de nombres aléatoires parce que le générateur inclus dans le

compilateur n'est pas très performant. Cette fonction n'est pas appelée directement dans le code, mais plutôt par la fonction *alea()* qui gère plus d'options.

CString RemplPoint(CString str)

Cette fonction sert à remplacer les points décimaux générés par le logiciel en virgule dans une chaîne de caractères (le point est le séparateur décimal en C++). Cette opération permet d'exporter les chaînes de caractères dans un fichier et de les ouvrir directement (sans conversion) avec une version française (métrique) de Microsoft Excel®. En effet, les paramètres régionaux du Canada français forcent l'utilisation de la virgule comme séparateur décimal.

void FichierLog()

Cette fonction a servi durant le développement pour écrire sur le disque dur, lors de l'exécution du logiciel, un fichier contenant différentes informations utiles au débogage. Cette fonction n'était qu'un outil de développement de logiciel et n'a pas d'utilité dans la version finale.

void FichierResult()

Cette fonction crée un fichier à la fin de l'optimisation qui contient les résultats de celle-ci. Ce fichier est au format « .csv » et peut donc être ouvert et modifié directement à partir de Microsoft Excel®. Étant un fichier texte de format standard, il peut aussi être ouvert, lu et modifié par n'importe quelle application informatique qui peut importer des fichiers texte.

void FichierPop(Mat_IO_SP Population)

Cette fonction écrit un fichier texte sur le disque dur (aussi en format « .csv ») contenant une population complète. Ce peut être une population d'enfants, de parents ou de survivants. Cette fonction a servi au développement du logiciel en permettant de prendre

une image instantanée, à tout moment, d'une population quelconque. Cette fonction n'était qu'un outil de débogage de logiciel et n'a pas d'utilité dans la version finale.

double alea(int type, double min, double max, double param1)
Cette fonction retourne un nombre aléatoire en utilisant la fonction *DP NR::ran1()*. Elle complète la fonctionnalité de celle-ci en ajoutant plus d'options. Selon le type, elle retourne un entier ou un réel et elle renvoie une valeur entre les minimums et maximums spécifiés lors de l'appel.

int arrondi(double reel)
Cette fonction renvoie simplement un entier qui est la valeur arrondie du réel qui est fourni lors de l'appel. Le réel est arrondi à l'entier inférieur si la portion décimale est inférieure à 0,5 et à l'entier supérieur dans le cas contraire.

3.2 Exemple numérique

Voici l'ensemble des paramètres que nous avons utilisé pour notre exemple numérique. Il s'agit d'une opération de fraisage en bout typique sur une pièce en acier allié, avec une fraise à quatre flûtes en acier rapide (HSS pour High Speed Steel).

R=300 mm, v_t=36000 mm/min, t_l=0.5 min, t_s=30 min, nl=200, L=80 mm, W=30 mm, H=25 mm, P_s= 5.92E-02 kW/mm^3/min, η=0.85, α=5, β=1.75, γ=1.5, λ=0, k=5.00E+11, t_e=1 min, C_A=55 $, c_b=0.75 $, k_0=4 $/min, Z_r=4, Z_f=4, C_{pr}=90 $, C_{pf}=90 $, N_{ar}=7, N_{af}=4, P_{max} = 7,5 kW, F_{max} = 2 kN, Rap = 2 µm, Rae = 2 µm, N_{max} =12000 rpm, vf_{max} =8000 mm/min.

Le lecteur remarquera peut-être dans la liste des paramètres que l'exposant de l'équation de Taylor pour la largeur de coupe λ est égal à zéro. C'est que nous ne disposons pas

actuellement d'informations sur ce paramètre en pratique. Un plan d'expérience sur un outil spécifique permettra ultérieurement d'ajouter cette information au modèle. Nous allons néanmoins étudier son impact à la section 3.3.

3.2.1 Résolution

Pour les besoins de comparaison, nous avons résolu le modèle avec un logiciel commercial : le module What'sBest! de Lindo Systems inc. Nous avons choisi ce solveur parce que ce module s'intègre au chiffrier Microsoft Excel®, ce qui nous a permis de facilement soumettre notre modèle au solveur pour ensuite comparer nos résultats. Aussi, la compagnie offre une version d'essai complètement fonctionnelle pour des problèmes de taille académique.
Ce solveur a eu de la difficulté à résoudre le problème. Il a fallu lui donner une solution faisable, c'est-à-dire qui respecte toutes les contraintes, comme point de départ, pour qu'il soit capable de résoudre le problème sans rencontrer d'erreur. Notez que pour tous les résultats que le solveur Lindo nous a fournis, il déclarait la solution comme étant un optimum local, c'est-à-dire qu'il ne pouvait fournir un optimum global hors de tout doute.

Nous avons implanté l'algorithme génétique de base présenté au chapitre 2 en langage C++ dans un environnement Microsoft Windows® avec comme sortie des fichiers Excel® en réponse au problème. Nous avons implanté l'AG de base tel que présenté plus haut, sans ajout pour gérer la diversité, pour améliorer la vitesse de convergence ou raffiner l'optimum trouvé. Nous avons fait tous les essais sur le même micro-ordinateur afin de comparer les temps de résolution.

3.2.2 Comparaison des résultats

Les tableaux II et III illustrent les résultats que nous avons obtenus avec les deux solveurs. Comme il fallait s'y attendre, le solveur Lindo a pris plus de temps, mais a trouvé de meilleurs résultats que l'algorithme génétique. L'AG quant à lui, converge rapidement dans la zone optimale, mais a du mal à trouver précisément l'optimum.

Tableau II

Résultats des solveurs

	Obj.	Dr	Df	vr	vf	fr	ff	Npdr	df	Npwr	wf	U	Tu	Temps
Lindo:	min Tu	14,5	18,0	225,0	177,2	0,037	0,150	2	3,00	2	3,00	9,362	1,232	26 sec.
	min U	14,5	12,4	147,7	132,2	0,037	0,150	2	3,00	2	3,00	7,219	1,437	43 sec.
AG:	min Tu	14,5	15,8	226,7	161,3	0,036	0,141	2	3,00	2	3,00	9,392	1,244	14 sec.
	min U	14,5	14,0	146,7	153,8	0,037	0,115	2	3,00	2	3,00	7,309	1,453	13 sec.

Ceci s'explique facilement par le fait qu'en fin de résolution, les nouvelles solutions ne peuvent s'améliorer que par les mutations qui sont aléatoires. Il faut mentionner que dans la zone optimale, les fonctions objectifs ont une courbure assez faible (voir la Figure 15, page 51) et donc, de légères variations des variables de décision ont un impact mineur sur celles-ci.

Tableau III

Résultats comparatifs

	Min U	Min Tu	Temps (sec)
Selon Lindo:	7,2186	1,2320	69
Selon AG:	7,3091	1,2440	26
Différance (abs):	0,0905	0,0119	-43
Différance (%):	1,254%	0,968%	-62,3%

On pourrait facilement améliorer la précision de l'AG à l'aide d'une recherche par gradient ou par contrainte active. On pourrait aussi améliorer la rapidité de convergence par différentes techniques bien connues dans le domaine des AG, mais ce n'est pas notre but dans ce travail.

3.2.3 Analyse de l'optimum

Les deux plus importantes variables pour minimiser le coût sont les vitesses d'avance et de coupe en ébauche, v_r et f_r. En fixant les autres variables à leur valeur optimale, on peut tracer un graphique du coût d'usinage U en fonction de ces deux variables.

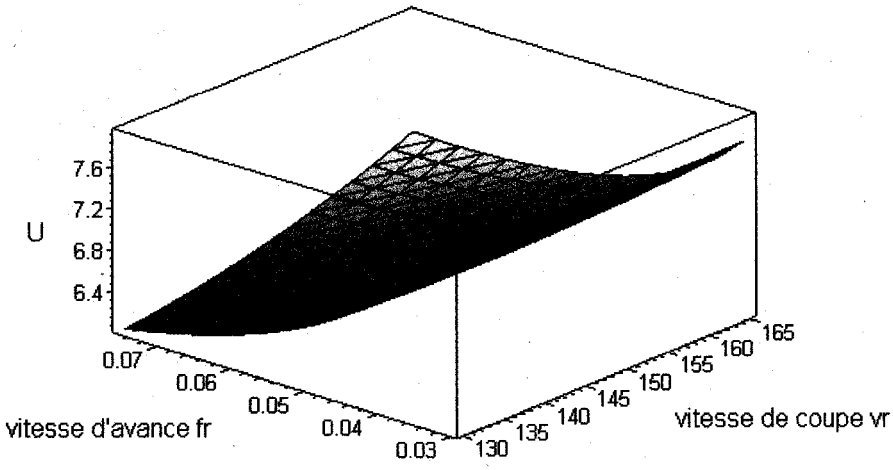

Figure 13 Coût d'usinage U en fonction de la vitesse d'avance fr et de coupe vr

On voit sur la Figure 13 que plus l'avance est grande et la vitesse petite, plus le coût est bas. Ceci s'explique facilement par le fait que plus la vitesse d'avance est grande, plus le temps d'usinage est court, ce qui diminue le coût relié au temps d'usinage. Quant à la vitesse de coupe, plus elle est basse, plus l'outil dure longtemps, ce qui diminue les coûts reliés à l'outil. La minimisation du coût U va être limitée par les contraintes actives. La plus importante contrainte concernant ces variables est la force de coupe.

La force de coupe limite seulement la vitesse d'avance, puisque les autres variables sont fixées. On peut voir sur la Figure 14 la force de coupe en fonction de la vitesse d'avance et la limite de cette contrainte (F_{max} = 2000 N).

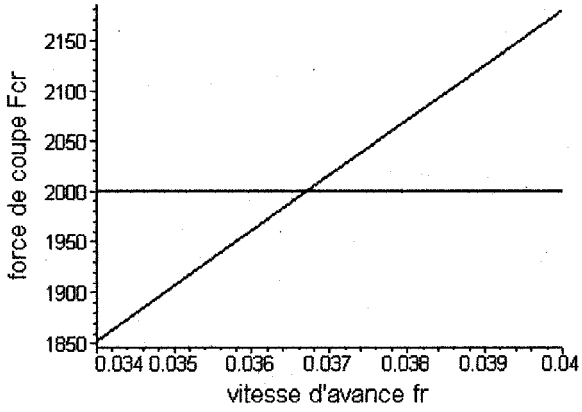

Figure 14 Force de coupe fcr en fonction de la vitesse d'avance fr

On voit bien sur ce graphique que la vitesse de coupe maximale pour respecter cette contrainte est f_r.= 0.0367 mm/dent.

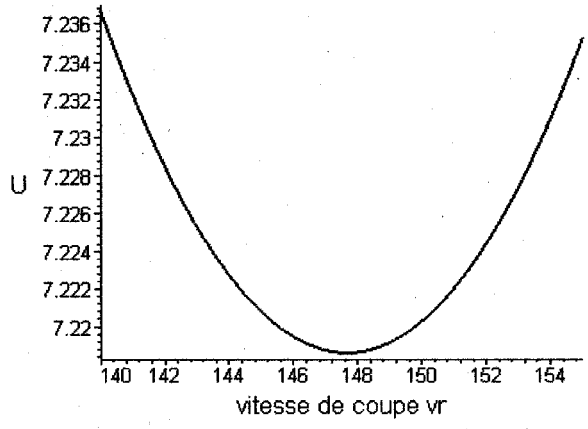

Figure 15 Coût d'usinage U en fonction de la vitesse de coupe v_r

En fixant la vitesse d'avance, on peut tracer un graphique à une seule variable qui permet d'illustrer le coût minimum U en fonction de la vitesse de coupe. On voit sur la Figure 15 que le minimum est à v_r= 147,7.

Une autre contrainte importante est le fini de surface en bout R_{ae}. Les variables qui l'affectent le plus sont les vitesses d'avance et de coupe en finition v_f et f_f. On peut voir sur la Figure 16 comment ces vitesses affectent le fini de surface.

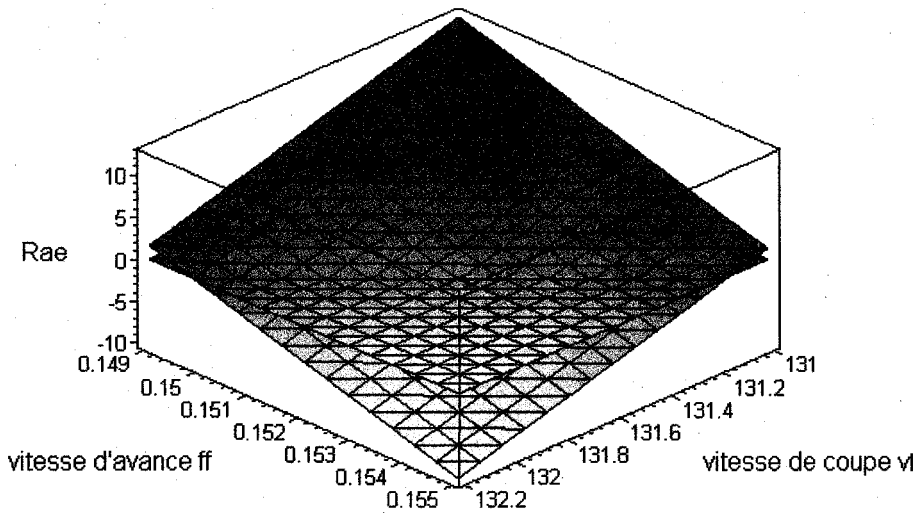

Figure 16 Fini de surface R_{ae} en fonction des vitesses d'avance f_f et de coupe v_f

On remarque que pour certaines combinaisons de variables, le fini de surface tombe dans le négatif. Ceci illustre une des limites de cette contrainte qui est obtenue par un plan d'expérience. Pour pouvoir explorer cette région de l'espace de solutions, il faudrait faire un plan d'expérience plus complet.

Nous allons exploiter notre modèle en deux parties. Tout d'abord, nous allons faire varier les paramètres afin d'illustrer leur impact sur les fonctions objectifs, ensuite, à la section 3.4, nous allons faire une analyse de sensibilité en relaxant certaines contraintes.

3.3 Variation des paramètres

Nous présentons dans cette section quelques exemples des analyses que l'on peut faire à partir du modèle. On peut faire l'essai de différents paramètres pour en mesurer l'impact sur le coût ou sur le temps d'usinage minimum. Notez que tous les points des graphiques de cette section ont été obtenus à l'aide de notre algorithme génétique. On peut donc remarquer que les courbes ont parfois de petites encoches qui sont dues au fait que quelquefois, l'optimiseur arrive à un point plus ou moins près de l'optimum théorique. Comme nous l'avons mentionné plus haut, l'AG converge rapidement vers la zone optimale, mais les petits ajustements ne reposent que sur les mutations aléatoires, ce qui explique de petites différences d'une fois à l'autre. En général, avec une dizaine de points, on peut facilement dégager une tendance. Nous allons commencer par regarder l'impact des exposants de l'équation de Taylor, (α, β, γ, λ) sur les optimums. Remarquez que le symbole $\otimes$ illustre le point obtenu à l'exemple numérique de la section précédente.

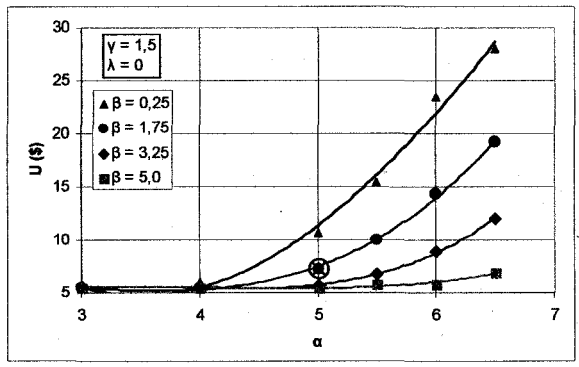

Figure 17 Coût d'usinage U selon α

On peut voir sur la Figure 17 que l'exposant de la vitesse de coupe a une grande influence sur le coût. D'abord, en dessous de $\alpha = 3$, l'outil dure tellement longtemps que l'influence sur le coût devient négligeable. D'un autre côté, au-delà de $\alpha = 6$, l'outil dure si peu longtemps qu'on ne fait que quelques pièces par outil et le coût unitaire monte en

flèche. Entre temps, on peut remarquer l'influence à la baisse de l'exposant β. Pour le temps d'usinage, on voit sur la Figure 18 qu'il augmente de la même façon. En fait, il est normal que moins l'outil dure longtemps, plus ça coûte cher et plus on passe de temps à changer les outils émoussés.

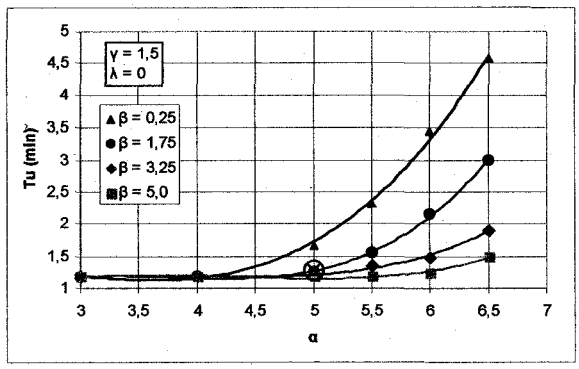

Figure 18 Temps d'usinage Tu selon α

Étant donné que l'impact des paramètres est similaire sur le temps d'usinage et le coût d'usinage, nous n'allons qu'illustrer celui-ci dans les prochains graphiques.

Pour ce qui est de l'exposant de la vitesse d'avance β, on peut voir sur la Figure 19 que l'impact numérique est inverse de celui de α. Ceci s'explique par le fait que la vitesse d'avance par dent s'exprime en fraction de millimètre. L'impact de β est moins important que α. On peut quand même voir un comportement similaire, mais inversé.

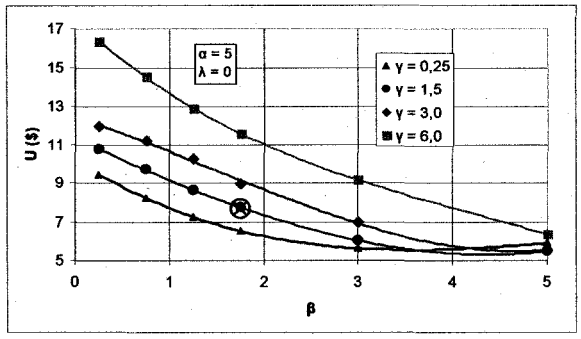

Figure 19 Coût d'usinage U selon β

On peut aussi voir à la Figure 19 que l'exposant de la profondeur de coupe γ a un impact à la hausse sur le coût d'usinage. Étant donné que la profondeur est supérieure à un millimètre, le coût est croissant en fonction de γ. Mais contrairement à l'effet de α et β, on remarque sur la Figure 20 une tendance plus linéaire. L'effet de λ quant à lui, est similaire à celui de γ, ce qui est normal puisque la largeur de coupe s'exprime elle aussi en millimètre et l'ordre de grandeur est similaire en fonction du diamètre de l'outil.

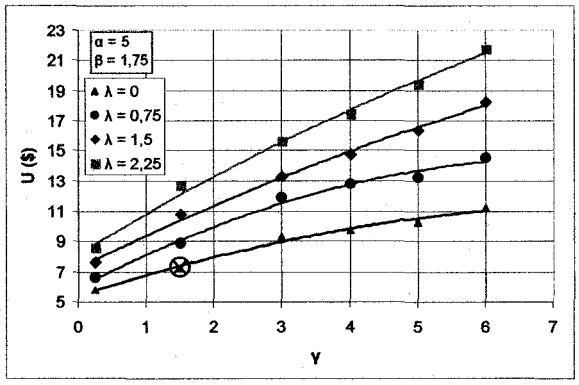

Figure 20 Coût d'usinage U selon γ

Nous avons fait une étude de l'impact du coût des outils sur le coût d'usinage. Il y a deux coûts à considérer pour les outils, le coût d'achat C_P et le coût d'aiguisage C_A. Le fait de varier un de ces coûts en fixant l'autre n'aurait pas beaucoup de sens. En général, le coût d'aiguisage est proportionnel au coût d'achat alors nous avons fait varier les deux coûts en proportion : C_A=24.4 pour C_P =40, C_A.=55 pour C_P =90, etc.

On peut voir sur la Figure 21 que, jusqu'à un coût C_P de 150, le coût d'usinage augmente rapidement parce que le coût des outils devient le coût dominant, ensuite, le coût d'usinage augmente plus linéairement avec le coût des outils.

Il a fallu modifier légèrement notre exemple numérique pour analyser l'impact fini de surface R_a sur le coût et le temps d'usinage. En effet, les deux contraintes du modèle d'optimisation pour le fini de surface n'étaient pas actives avec une limite de 2 µm.

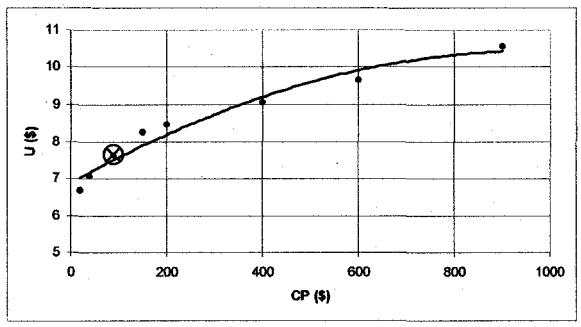

Figure 21 Coût d'usinage U selon C_P

Ceci est facile à comprendre, puisque le paramètre le plus important pour le fini de surface est la vitesse de coupe en finition. Que ce soit pour minimiser le temps ou le coût, l'optimiseur cherche à augmenter la vitesse, ce qui donne un meilleur fini de surface par le fait même. Afin de forcer le modèle à diminuer la vitesse optimale nous avons modifié les exposants de l'équation de Taylor à α=5.6 et β=1.35. De cette façon, la contrainte de fini de surface est activée et on peut en mesurer l'impact.

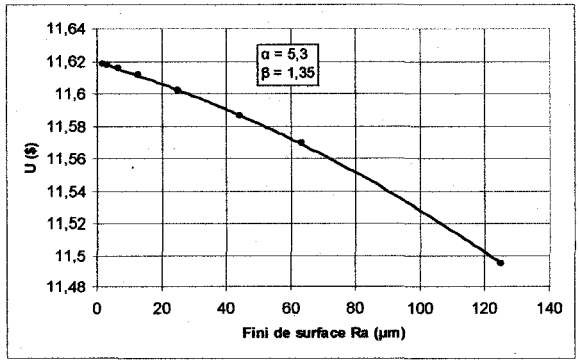

Figure 22 Coût d'usinage U selon Ra

On voit sur la Figure 22 que le coût d'usinage diminue légèrement quand le fini de surface requis est plus grossier. Cette contrainte, dans notre exemple, ne coûte pas trop cher. Il faut donc conclure que le fini de surface requis peut coûter plus cher, mais il faut que cette contrainte soit active. Si la contrainte est déjà inactive, il est normal qu'elle ne coûte rien. Il faut noter que beaucoup de paramètres dans le modèle influencent les contraintes et peuvent en activer une ou plusieurs de façon plus ou moins sévère. L'impact du fini de surface peut donc varier beaucoup d'une application à l'autre.

Avec les graphiques qui suivent, nous allons regarder l'impact de deux caractéristiques qui ne sont pas directement des paramètres d'usinage, mais des concepts importants en gestion de la production : le temps de mise en course et la taille des lots de pièces.

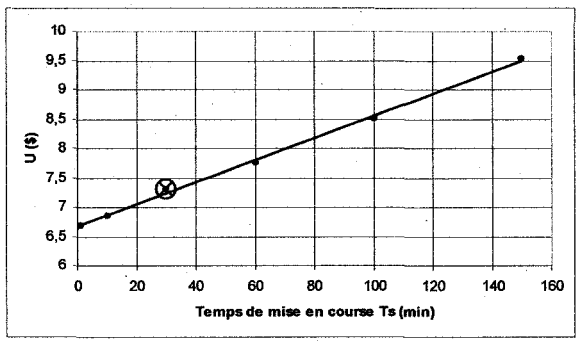

Figure 23 Coût d'usinage U selon Ts

Pour obtenir le graphique de la Figure 23, nous avons fait varier le temps de mise en course. La première donnée correspond à un temps d'une minute et nous montre l'économie qu'un échange de mise en course rapide SMED (Single Minute Exchange of Die) apporte. La relation entre le temps de mise en course et le coût d'usinage est linéaire et une réduction de T_S entraîne une réduction directe des fonctions objectifs.

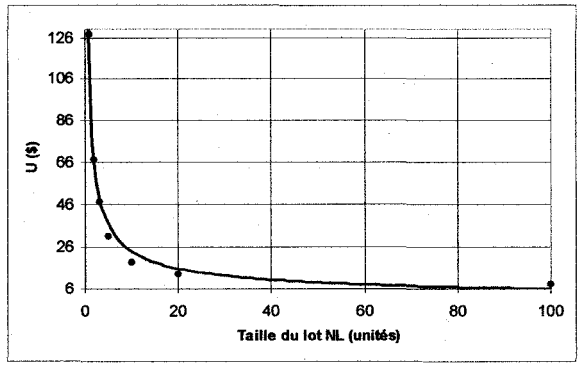

Figure 24 Coût d'usinage U selon N_L

Pour la Figure 24, nous avons étudié l'impact de la taille des lots sur les coûts d'usinage. On peut voir que ceux-ci sont très élevés pour un lot d'une pièce, ce qui correspond à une production à l'unité. Ensuite, les coûts diminuent rapidement à mesure que le temps de mise en course est distribué sur un plus grand nombre de pièces. Notez que les valeurs les plus élevées (à N_L=1) dépendent aussi du temps de mise en course. Dans notre exemple, pour la Figure 24, T_S = 30. Alors, ce temps est directement ajouté au temps d'usinage.

Les graphiques de cette section sont de bons exemples de la quantité d'information et de la variété des analyses que l'on peut faire à partir du modèle que nous avons développé. En fait, tous les paramètres du modèle peuvent être analysés de la même façon.
Il faut tout de même considérer les limites de ce modèle qui ne s'occupe que de l'opération de fraisage en bout elle-même.

Par exemple, on peut voir sur la Figure 24 que le fait d'augmenter la taille du lot de dix à vingt pièces engendre une économie de près de 6 $ l'unité. Il faut par contre considérer l'ensemble des coûts de l'usine avant de prendre une décision de lotissement. Si on économise 6 $ au fraisage en augmentant la taille des lots, il ne faut pas que cette économie à une opération fasse augmenter les coûts de maintien des stocks en-cours de plus de 6 $ dans l'ensemble des opérations, sinon, ça ne vaut pas la peine.

3.4 Analyse de sensibilité

Comme on a vu précédemment, les optimums que l'on va trouver à l'aide de notre modèle et de notre algorithme sont limités dans l'espace de solution par différentes contraintes. Nous allons ici nous concentrer sur les contraintes de force et de puissance maximales parce que ces contraintes concernent directement le choix d'une machine et la conception des montages et gabarits des pièces sur les machines. Nous allons bien sûr utiliser le même exemple numérique de la section 3.2 comme base afin de comparer les résultats. Dans le cas de la minimisation du temps d'usinage, les deux contraintes sont actives. Pour la minimisation du coût d'usinage, seule la contrainte de force maximale est active. Nous allons dans cette section relaxer ces contraintes pour déterminer l'impact que cela aurait sur les fonctions objectifs. Le fait de relaxer les contraintes a une signification très précise dans le système réel. En effet, le fait d'augmenter la puissance maximale signifie que l'on utilise une machine plus puissante pour faire l'opération. La force maximale quant à elle peut être augmentée en utilisant un montage plus solide de la pièce sur la machine. Ceci peut être fait avec un étau plus solide ou en fabriquant un meilleur gabarit. Nous allons relaxer les contraintes de puissances maximales et de forces maximales tant qu'on constate une amélioration des fonctions objectifs, jusqu'à 100 % d'augmentation de leur valeur initiale, c'est-à-dire jusqu'à doubler la puissance et la force admissible. Nous ne croyons pas qu'il faudra aller au-delà parce que cela changerait trop le modèle et les comparaisons deviendraient hasardeuses. En effet, quand on en est à considérer de doubler la force ou la puissance de coupe, il convient de réviser tous les aspects du procédé, incluant les outils de coupe, les matériaux, la forme des bruts, les opérations précédentes et suivantes, etc.

3.4.1 Objectifs et méthodologie de cueillette de données

Afin de nous donner une vue d'ensemble nous avons tracé quelques graphiques en fonction de v_r la vitesse de coupe et f_r la vitesse d'avance, les principales variables qui affectent la puissance et la force de coupe. Notez que les autres variables peuvent aussi affecter la puissance et la force, mais pour tracer ces graphiques nous avons fixé les autres variables à leur valeur optimale de l'exemple de référence. Nous nous intéresserons principalement à la passe d'ébauche dans cette section puisque c'est lors de celle-ci que les contraintes de force et de puissance sont actives. Pour le cas de la force, elle est affectée seulement par la vitesse d'avance et on peut donc tracer le graphique de la Figure 25 en fonction d'une seule variable.

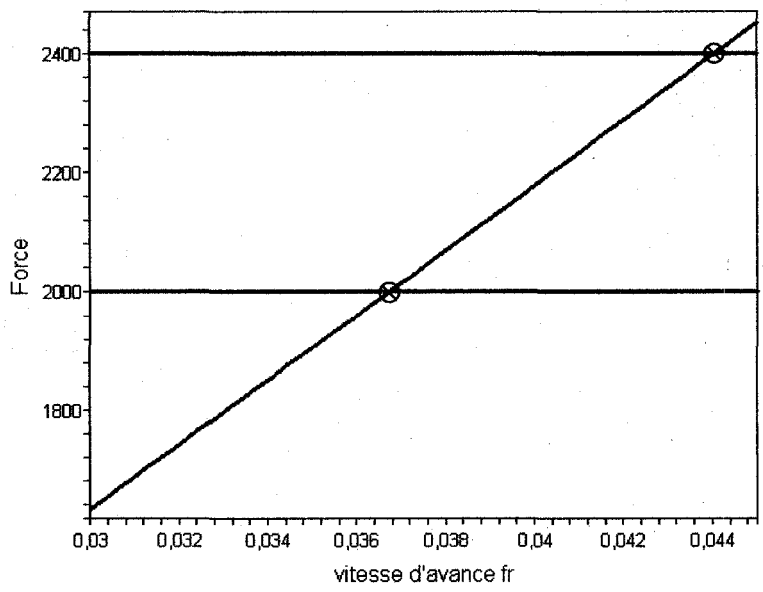

Figure 25 Force en fonction de de la vitesse d'avance en ébauche

On peut remarque sur ce graphique que si on augmente la force maximale de 2000 à 2400 N, on peut augmenter la vitesse d'avance de 0,037 à 0,044 mm/dent, ce qui devrait diminuer les valeurs des fonctions objectifs.

On peut voir évoluer à la Figure 26 la puissance en fonction des vitesses de coupe et d'avance. Nous avons aussi illustré deux exemples de contraintes à 7,5 et 9 kW. Ce graphique en trois dimensions nous offre une bonne illustration générale pour situer le graphique de courbes de niveau de la Figure 27. Ce graphique de courbes de niveau permet de voir plus précisément les valeurs numériques. On voit bien sur cette figure les couples de variables qui sont possibles pour une puissance donnée. On remarque par exemple que si on fait passer la puissance maximale de 7,5 kW à 9 kW, et que la vitesse d'avance est augmentée de 0,037 à 0,044 mm/dent comme on a vu pour la force, la vitesse de coupe est maintenue à 225 m/min.

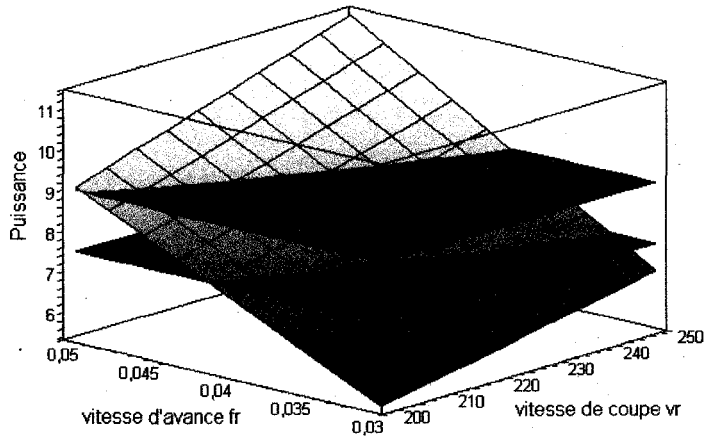

Figure 26 Puissance (3D) en fonction de l'avance et de la vitesse de coupe en ébauche

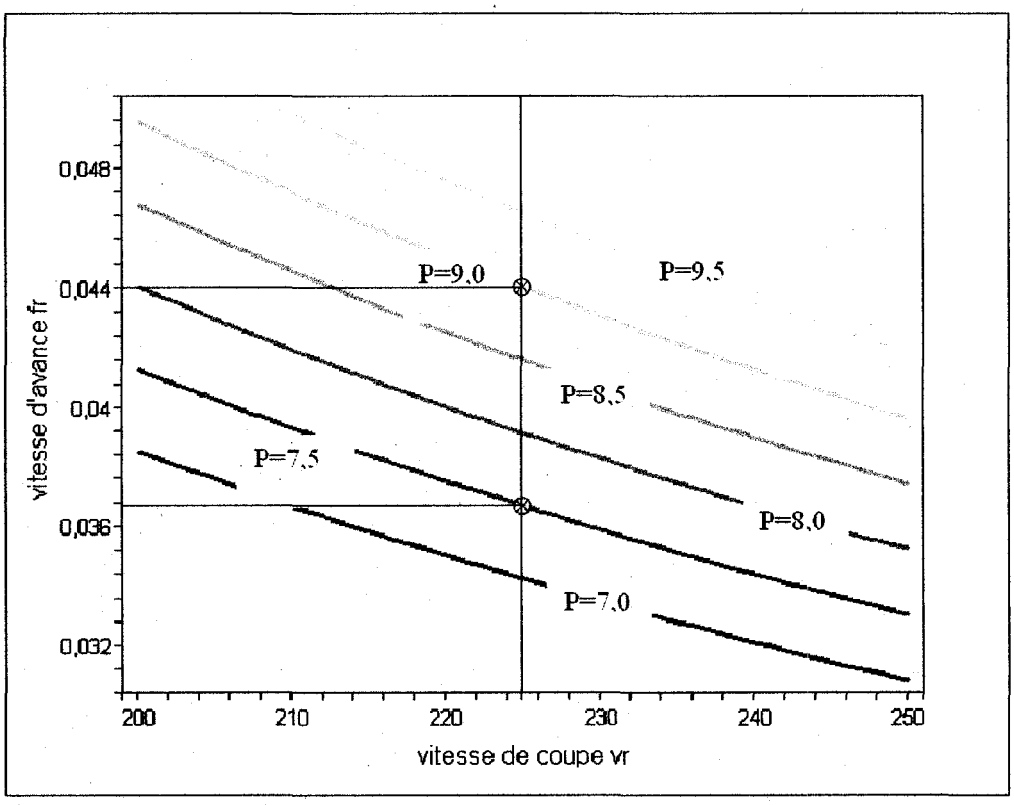

Figure 27 Puissance (courbe de niveau) en fonction de l'avance et de la vitesse de coupe en ébauche

C'est peut-être ce qui va arriver, mais pour en être sûr il faudra faire les essais. Comme l'exposant de la vitesse de coupe est plus élevé que l'exposant de la vitesse d'avance dans l'équation de la durée de vie de l'outil, le logiciel d'optimisation va probablement chercher à augmenter la vitesse d'avance avant la vitesse de coupe.

Évidemment, ces graphiques sont une simplification du problème puisque les autres variables sont fixées. Mais on peut quand même prévoir que les fonctions objectifs pourront être améliorées en relaxant ces contraintes.

En général, quand on résout un problème linéaire, le fait de relaxer une contrainte permet d'améliorer linéairement les optimums jusqu'à l'activation d'une autre contrainte. Mais ce n'est pas ce qui va se passer avec notre modèle. En effet, comme nous sommes dans un espace de solution à dix variables non linéaire, en relaxant les

contraintes nous agrandissons le domaine de solution, ce qui permettra probablement d'améliorer les optimums, mais les résultats ne sont pas aussi prévisibles que dans le cas d'un modèle linéaire simple.

À partir de quelques tests préliminaires, nous avons constaté que des variations de 10 % à 20 % de la puissance ou de la force nous donnent des diminutions de l'ordre de 1 % à 5 % de la valeur des fonctions objectifs. Comme ces différences sont assez petites quoique non négligeables, nous croyons qu'il convient de faire plusieurs essais et d'en faire la moyenne afin de diminuer l'effet des dispersions de résultats, dont nous avons déjà discuté à la section 3.2.2. Afin de déterminer le nombre de réplications suffisantes pour amortir les dispersions nous avons procédé à dix essais identiques, et nous avons tracé les données et les moyennes de ces données sur le graphique de la Figure 28. Nous avons considéré le temps d'usinage pour ces essais, mais le coût d'usinage aurait donné des résultats similaires.

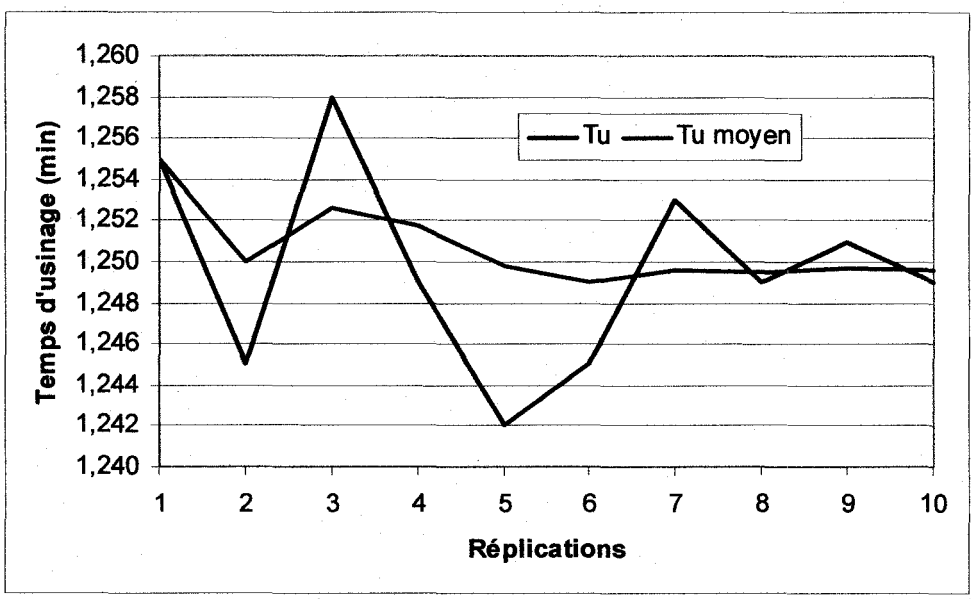

Figure 28 Temps d'usinage, évolution de la moyenne avec les réplications

Les données oscillent entre 1,242 et 1,258 minutes avec une moyenne de 1,250 minutes. On voit bien sur ce graphe que l'effet des pics sur la moyenne est déjà bien amorti après trois réplications et que la moyenne devient très stable après cinq réplications et ensuite les fluctuations sont beaucoup moins importantes. Notons que le plus grand écart à la moyenne est de 0,008 soit 0,67 %, ce qui indique que les résultats sont quand même très précis. Nous pensons donc que la moyenne de cinq réplications nous permettra de générer des données assez stables pour pouvoir les analyser correctement. Chaque donnée des tableaux IV à VI, et chaque point des figures 30 et 31 représentera donc la moyenne de cinq essais identiques.

3.4.2 Résultats et analyses

Pour commencer, nous allons augmenter la puissance maximale. Le Tableau IV résume les résultats de l'optimisation pour le temps et le coût d'usinage. On voit que le logiciel d'optimisation réussi à diminuer le temps d'usinage de 1,26 % à 8,25 kW et de 0,39 % supplémentaire à 9 kW. Le gain tombe à moins de -0,08 % à 9,75 kW. Nous n'avons pas à aller plus loin pour voir que le gain en temps d'usinage est minime par rapport à l'augmentation de la puissance. Ceci s'explique par le fait que la contrainte de force est aussi active pour le temps d'usinage et cet essai semble indiquer que la force maximale retreint plus sévèrement le problème que la puissance maximale.

Tableau IV

Temps et coûts d'usinage selon P_{max}

Pmax	Différence	Tu	Différence	U	Différence
7,5	0,0%	1,250	0,00%	7,228	0,00%
8,25	10,0%	1,234	-1,26%	7,225	-0,05%
9	20,0%	1,229	-1,65%	7,224	-0,05%
9,75	30,0%	1,228	-1,73%	7,227	-0,01%

Pour ce qui est du coût d'usinage, on confirme le fait que la puissance maximale n'est pas une contrainte active puisque le fait de l'augmenter n'apporte aucune diminution du coût.

En constatant que la force maximale semble la plus astreignante des contraintes, on s'attend à des résultats plus intéressants en relaxant celle-ci. Le Tableau V rassemble les données qui ont servi à tracer le graphique de la Figure 29 afin de mieux visualiser l'évolution. On peut tout d'abord remarquer dans le tableau que les premiers résultats indiquent une nette diminution des fonctions objectifs de 1.36 % pour le temps et de 2.48 % pour le coût. Ensuite, l'effet s'estompe d'abord pour le temps et plus loin pour le coût. L'effet de l'augmentation de la force diminue plus rapidement dans le cas du temps d'usinage. C'est dû au fait que la puissance maximale est aussi une contrainte active. On peut conclure de ces résultats que pour le temps d'usinage, à 7,5 kW et 2000 N, la contrainte de force est plus importante que celle de puissance.

Tableau V

Temps et coûts d'usinage selon F_{max}

Fmax	Différence	Tu	Différence	U	Différence
2000	0,0%	1,250	0,00%	7,228	0,00%
2200	10,0%	1,233	-1,36%	7,048	-2,48%
2400	20,0%	1,222	-2,22%	6,886	-4,73%
2600	30,0%	1,213	-2,94%	6,717	-7,07%
2800	40,0%	1,209	-3,25%	6,574	-9,05%
3000	50,0%	1,204	-3,65%	6,493	-10,17%
3200	60,0%	1,198	-4,13%	6,388	-11,62%
3400	70,0%	1,193	-4,51%	6,289	-12,99%
3600	80,0%	1,192	-4,59%	6,180	-14,50%
3800	90,0%	1,191	-4,69%	6,117	-15,37%
4000	100,0%	1,188	-4,91%	6,049	-16,31%

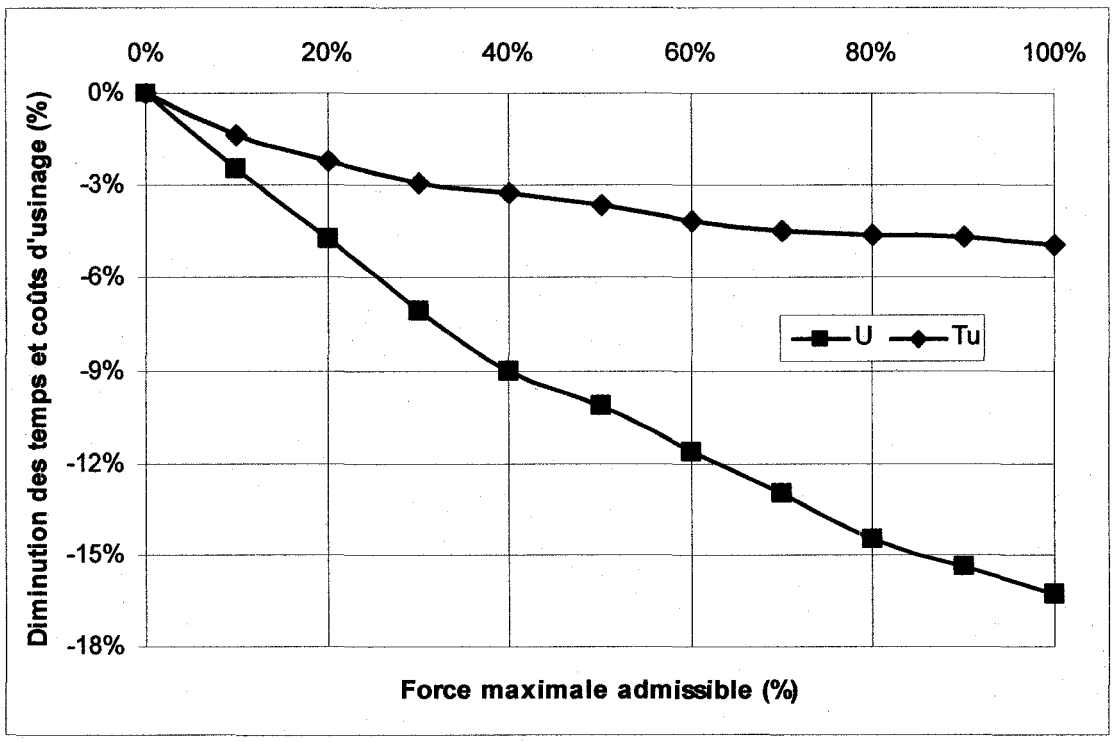

Figure 29 Temps et coût d'usinage selon F_{max}

Quand on commence à augmenter la force, le gain est important au début, mais graduellement, il diminue après avoir atteint le point d'équilibre entre les deux contraintes qui se situe environ à 7,5 kW et 2200 N.

Ensuite, la puissance étant la contrainte principale, le gain relatif entre les points diminue en augmentant la force pour tomber en deçà de -0,50% à 7,5 kW et 2800 N.
Du coté du coût d'usinage, le même phénomène se produit, mais à une échelle (point force, puissance) différente. Au départ, la force est la seule contrainte active alors en l'augmentant, le gain est important jusqu'à l'activation de la contrainte de puissance. On commence à s'approcher de la contrainte de puissance vers 7,5 kW et 3200 N, mais quelques essais supplémentaires nous ont permis de déterminer que la puissance n'est vraiment activée que vers 3800 N. La force demeure donc la principale contrainte puisqu'on constate que même si le gain diminue, il demeure quand même autour de -1 % à la fin des essais, c'est-à-dire à 7,5 kW et 4000 N. Nous avons décidé d'arrêter les essais

au double de la force initiale, mais on peut prévoir qu'éventuellement la force ne sera plus active du tout et le fait de l'augmenter ne donnera plus de gain en coût d'usinage. Afin de voir l'action combinée des contraintes de force et de puissances, nous avons fait un essai en les augmentant toutes les deux en même temps. Pour cet essai, nous avons augmenté la force et la puissance avec le même incrément en pourcentage. Le Tableau VI et la Figure 30 illustrent les résultats.

Tableau VI

Temps et coût d'usinage selon P_{max} et F_{max}

Pmax	Fmax	Différence	Tu	Différence	U	Différence
7,5	2000	0,0%	1,250	0,00%	7,228	0,00%
8,25	2200	10,0%	1,205	-3,55%	7,023	-2,83%
9	2400	20,0%	1,190	-4,76%	6,878	-4,85%
9,75	2600	30,0%	1,162	-7,07%	6,710	-7,17%
10,5	2800	40,0%	1,146	-8,32%	6,592	-8,81%
11,25	3000	50,0%	1,126	-9,92%	6,462	-10,59%

On remarque d'abord que pour le temps d'usinage, les résultats sont très intéressants. Après une diminution initiale de -3,55 %, le gain en temps d'usinage se maintien à environ -1,5 % à chaque point.

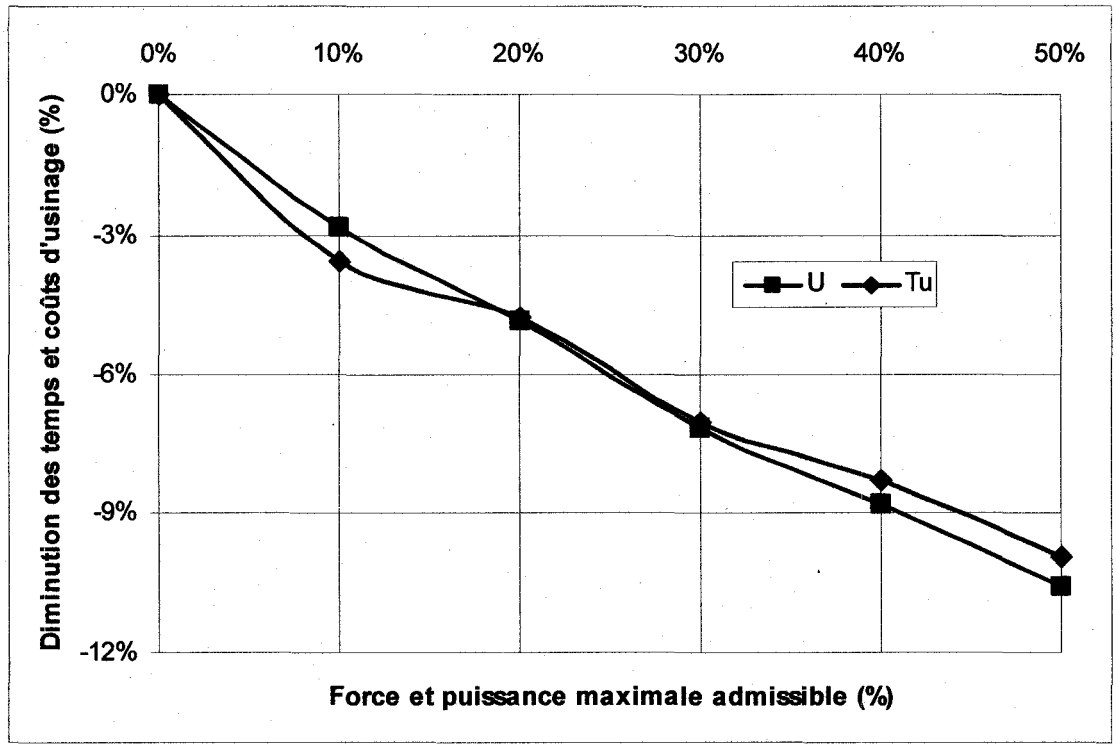

Figure 30 Temps et coût d'usinage selon P_{max} et F_{max}

Ces résultats s'expliquent par le fait que les deux contraintes sont toujours actives en même temps, alors en les relaxant en même temps, elles demeurent toutes les deux actives, mais diminuent beaucoup plus le temps d'usinage. Ces résultats semblent indiquer que pour diminuer le plus efficacement possible le temps d'usinage par la relaxation des contraintes, il faut d'abord relaxer la plus active des contraintes jusqu'à l'activation de l'autre et ensuite les augmenter simultanément en les gardant toutes les deux actives.

Pour ce qui est du coût d'usinage, les résultats sont moins intéressants. En fait, on obtient à peu près les mêmes résultats que pour l'essai précédent. Ce n'est pas surprenant puisque pour le coût d'usinage, la puissance n'est pas une contrainte active alors le fait de l'augmenter ne change pas grand-chose. Seule l'augmentation de la force a un impact sur le coût ce qui fait que les résultats des tableaux V et VI sont similaires pour celui-ci. On ne peut pas vraiment faire le même exercice pour le coût d'usinage

puisque la contrainte de puissance est activée à une force beaucoup trop grande. Il faudrait augmenter la force au-delà de 100 %, ce que nous ne ferons pas pour les raisons expliquées précédemment.

On peut tirer quelques conclusions générales sur les essais que nous avons effectués dans cette section. Mais il faut se rappeler que ces recommandations ne tiennent que pour ce cas particulier. Il conviendrait de faire ce genre d'analyse à chaque cas d'application pour établir des recommandations sur les essais spécifiques à chaque application. Tout d'abord, il faut déterminer si on désire diminuer le temps d'usinage ou le coût d'usinage, parce que cela va déterminer quelle contrainte il faut augmenter afin d'améliorer la fonction objectif. Il faut aussi réaliser qu'en relaxant les contraintes, les gains pour les fonctions objectifs sont limités. Dans les essais que l'on a faits, la plus grande diminution du temps d'usinage est de -9,92 % et la plus grande diminution du coût d'usinage est de -16,31 %. On ne peut donc affecter que légèrement les fonctions objectifs en augmentant les limites de ces contraintes. Comme on l'a déjà mentionné, pour augmenter ces limites en pratique il faut une machine plus puissante dans le cas de la contrainte de puissance et un montage plus solide de la pièce sur la machine dans le cas de la contrainte de force. Donc, si on désire améliorer les temps et les coûts d'usinage en augmentant les limites de ces contraintes il convient tout d'abord de déterminer si c'est possible en pratique. Ensuite il faudrait évaluer combien il en coûterait pour effectuer les modifications de montage et de machine afin de modifier ces contraintes.

Si on décide de diminuer le temps d'usinage au minimum des essais que l'on a fait : jusqu'à -9,92 %, la façon la plus efficace est d'augmenter la force et la puissance de 50 %, c'est-à-dire à 11,25 kW et 3000 N. Nous savons que cela est faisable selon notre modèle, mais il faudrait vérifier si cela est possible en pratique et en évaluant les coûts de ces modifications, une décision pourra être prise de façon éclairée sur la rentabilité de ces modifications.

De la même manière, si on décide de diminuer le coût d'usinage au minimum des essais effectués, soit à -16,31 %, il faut augmenter la force de 100 % à 4000 N. Si cette augmentation n'est pas possible en pratique, à partir de la plus grande augmentation de la force qu'on considère faisable, on pourra déterminer la diminution du coût que l'on pourra atteindre.

Si de plus grandes diminutions du temps ou du coût d'usinage sont requises ou souhaitées, nous pouvons savoir grâce à une analyse comme celle que nous avons faite que cela ne pourra pas être possible seulement en augmentant la puissance ou la force admissible. Il faudra donc chercher à améliorer d'autres caractéristiques du procédé, comme de meilleurs outils de coupe par exemple, ou alors d'ajouter des machines si la capacité de production n'est pas suffisante.

On peut noter finalement que ce type d'analyse de sensibilité pourrait être assez facilement intégré au logiciel en lui permettant de garder dans une liste les individus les plus performants qui dépassent globalement les contraintes d'un certain pourcentage, par exemple de 5 % ou 10 % et regarder comment ce pourcentage de dépassement de contrainte se répartit entre les différentes contraintes. Cette caractéristique pourrait par exemple permettre au logiciel de faire des recommandations aux utilisateurs concernant l'amélioration des fonctions objectifs.

CHAPITRE 4

FABRICATION ET GESTION DE LA PRODUCTION

4.1 Exploitation dans les systèmes de contrôle

Nous allons décrire dans cette section une des applications possibles de notre modèle : une interface entre un système de contrôle et l'opération de fraisage en bout. Il convient tout d'abord de mettre en contexte le concept de stratégie de commande optimale. Les systèmes de contrôle de la production se retrouvent principalement dans les systèmes de production flexibles ou FMS (pour Flexible Manufacturing Systems).

4.1.1 Les systèmes de production flexibles

La flexibilité est la capacité d'un système à varier son taux de production, traiter différents designs d'une famille de pièces, permettre des routages différents à l'intérieur de l'usine et de varier la taille des lots de pièces. On peut inclure différents aspects dans le terme général de flexibilité, en voici quelques-uns.

La flexibilité des machines indique la facilité pour une machine à exécuter des opérations différentes. Elle est caractérisée par le nombre d'opérations différentes qui sont exécutées sur un équipement et par le temps de mise en course nécessaire au changement d'opération, qui doit bien sûr être le plus court possible.

La flexibilité des opérations est la facilité à faire une séquence différente d'opérations afin d'arriver aux mêmes produits finis. Elle est notamment caractérisée par le type de design, le temps pour implanter une séquence d'opérations et la hiérarchie du système de contrôle informatique.

La flexibilité des volumes est la capacité de fabriquer différent volume de la même pièce.

La flexibilité de l'expansion est la facilité de prendre de l'expansion au niveau du plancher d'usine sans avoir à reconfigurer l'aménagement existant. Elle est augmentée par la réduction de l'investissement nécessaire pour accroître la capacité de l'usine.

La flexibilité des cheminements est mesurée par le nombre de chemins efficaces possibles que peut prendre la matière première afin de se transformer en produits finis.

La flexibilité du procédé représente le volume d'une pièce qu'on peut produire dans le système sans avoir à faire de mise en course. Elle regroupe directement la flexibilité des machines, des équipements de manutention, des opérations et des cheminements.

La flexibilité du produit relève de sa conception, les différents produits d'une compagnie doivent être très similaires dans les composantes pour minimiser les niveaux d'inventaire des pièces nécessaires à la production. Les gabarits doivent également être flexibles afin de pouvoir fabriquer plusieurs pièces sur un même gabarit sans devoir le changer.

La flexibilité des programmes est le temps qu'un programme peut s'exécuter sans devoir subir d'intervention externe (humaine).

La flexibilité de la production regroupe les catégories précédentes, car elle est définie comme étant la capacité du système à produire des pièces différentes sans avoir à faire des investissements importants en capitaux. La capacité à faire les changements d'outils (avec les magasins d'outils) est également comprise dans ce type de flexibilité.

La flexibilité du marché est la facilité à s'adapter au changement des conditions du marché. Ceci veut dire le temps pour s'adapter au changement de commandes des clients afin de répondre à leurs besoins.

Afin de bien répondre au marché actuel, tous les aspects de flexibilité doivent être à leur meilleur afin de répondre le plus efficacement possible aux fluctuations du marché, et ce, en minimisant les interventions nécessaires au fonctionnement du système (panne, maintenance, nettoyage, etc.) À cause de ces caractéristiques, il y a une grande quantité d'information à traiter et la gestion d'un tel système est très complexe. L'article d'Adlemo et Andreasson (1996) favorise le contrôle automatisé dans les systèmes de production modernes. Avec des exemples, ils démontrent qu'il existe plusieurs niveaux possibles d'automatisation, en indiquant clairement que le niveau supérieur (automatisation au maximum) est le plus souhaitable dans bien des applications.

4.1.2 La stratégie de commande optimale

Si notre opération de fraisage en bout est automatisée au maximum, notre modèle d'optimisation peut servir d'intermédiaire entre le système de contrôle et l'opération de fraisage. Sans entrer dans le détail du fonctionnement du contrôleur, mentionnons que celui-ci recueille des informations sur la production : niveau des stocks et état des machines, par exemple. D'autres informations peuvent aussi être fournies au contrôleur ainsi que des commandes provenant d'un niveau hiérarchique supérieur. Notez que l'on parle ici de contrôleur d'un point de vue du système. Ce contrôleur peut être un opérateur humain, un système informatique ou une combinaison des deux. Quand le contrôleur a une nouvelle opération à lancer, il la soumet au modèle d'optimisation qui minimise les deux objectifs et lui fournit en retour la fenêtre opérationnelle de l'opération : les temps d'usinages minimums et maximums et les coûts d'usinage minimum et maximum. À partir de ces informations, le contrôleur peut décider en temps réel à quelle vitesse ou à

quel coût produire. Selon cette commande, le modèle d'optimisation minimise l'autre objectif en fonction de la commande. Le modèle d'optimisation fournit en sortie les valeurs des dix paramètres d'usinage pour produire selon la commande de façon optimale.

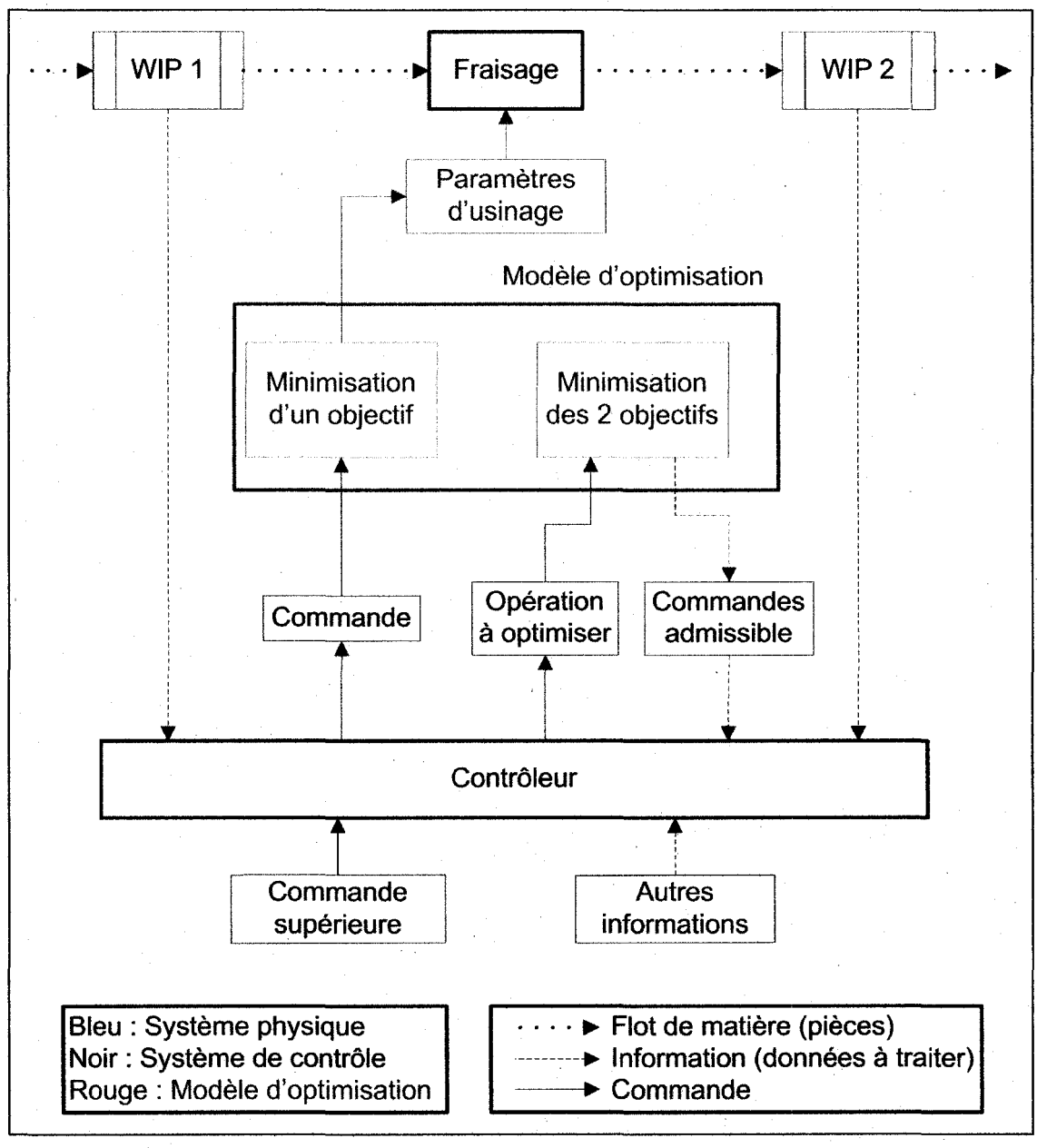

Figure 31 Système de contrôle

La Figure 31 illustre ces interactions entre le système de contrôle, le modèle d'optimisation et l'opération de fraisage dans le système de production.

Par exemple, en prenant les résultats de l'exemple donné au chapitre 3, la plage de temps d'usinage va de 1,24 minute à 1,45 minute à un coût de 9,39 $ à 7,31 $. En tenant compte de toutes sortes d'information, disons que le contrôleur demande de produire à une cadence d'une pièce toutes les 1,35 minutes.

En fournissant cette commande au modèle d'optimisation, celui-ci va calculer le coût d'usinage à 7,44 $. C'est le coût minimum auquel on peut produire les pièces en 1,35 minute par pièce. Si on désire produire à un coût plus bas, il faudra accepter un temps d'usinage plus long. Afin de visualiser le lien entre le temps d'usinage et le coût d'usinage, nous avons tracé le graphique de la Figure 32, en faisant minimiser le coût d'usinage pour différentes valeurs de temps d'usinage.

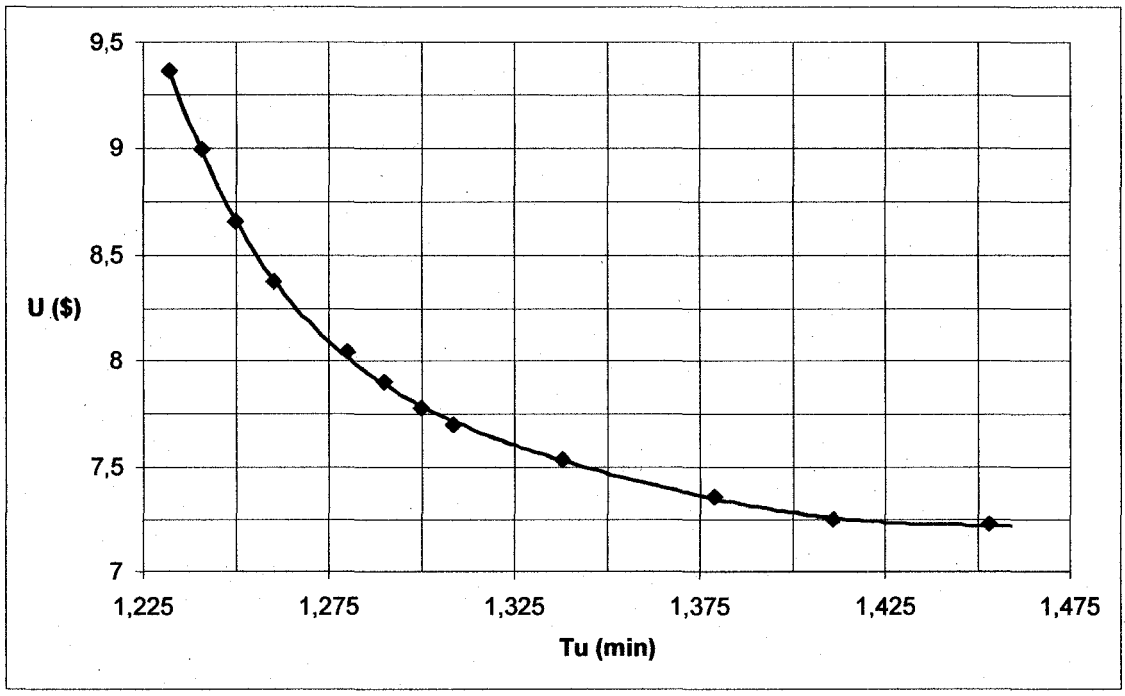

Figure 32 Coût d'usinage minimum pour le temps d'usinage requis

De la même façon, si c'est le coût qui intéresse le contrôleur et qu'il demande de produire à un coût de 8,25 $, le modèle va calculer à un temps d'usinage minimum de 1,27 minute. On voit bien dans ces petits exemples et sur le graphique que la relation n'est pas du tout linéaire entre le temps d'usinage et le coût d'usinage. Aux extrémités de la fenêtre opérationnelle, par exemple au temps de production minimum, une légère augmentation du temps d'usinage de 1,5 % engendre une diminution du coût d'usinage d'environ 8 %. Il convient donc de bien choisir les objectifs que l'on veut optimiser.

4.2 Autres applications

Ayant déjà fait la démonstration de la grande quantité d'information que l'on peut obtenir à partir du modèle que nous avons développé, on peut facilement imaginer d'autres applications industrielles, administratives et scientifiques à toutes ces informations. Nous en proposons quelques brefs exemples dans cette section.

4.2.1 Capacité et faisabilité d'un système

Nous avons déjà parlé de la fenêtre opérationnelle de l'opération. Cette fenêtre nous donne le temps d'usinage minimum auquel une opération peut être complétée au coût maximum et le coût d'usinage minimum possible en un temps maximum. Nous avons considéré précédemment cette fenêtre opérationnelle comme commande admissible pour l'opération à la section précédente. Mais bien avant qu'un système de production avec politique de contrôle soit en fonction, notre modèle d'optimisation peut être utilisé dans la conception et la planification d'un atelier de fabrication.

En effet, à partir des dessins des pièces à produire, des caractéristiques des outils disponibles et des spécifications des machines, on peut utiliser notre modèle afin de prévoir les performances du système en conception. Ces performances sont bien sûr les temps et coûts d'usinages. Premièrement, les coûts d'usinage prévus peuvent être utilisés dans une analyse économique. On pourra prévoir les investissements requis en machine, main d'œuvre et outillage, et les revenus ou profits auxquels on peut s'attendre. Cette analyse permet de déterminer si un projet vaut la peine d'être entrepris. S'il en vaut la peine, on pourra ensuite regarder les temps d'usinage afin de voir si une machine sur un quart de travail permet de rencontrer les délais de livraison. Si ce n'est pas le cas, il faudra déterminer la meilleure option afin de répondre à la demande. Si la machine n'est pas encore achetée, on peut éventuellement en considérer une plus puissante. Sinon, on peut ajouter un quart de travail ou envisager le travail en temps supplémentaire. Il sera

peut-être aussi nécessaire d'acheter une ou plusieurs machines afin d'augmenter la capacité. Toutes ces décisions relèvent bien sûr de la gestion des opérations et nous n'allons pas décrire ici les méthodes et techniques d'analyse spécifique aux sciences de la gestion. Nous pouvons néanmoins affirmer que notre modèle d'optimisation peut fournir beaucoup d'informations qui peuvent aider et soutenir les décisions de conception et de planification des systèmes de production par usinage.

4.2.2 Planification et gestion des stocks de soutien

Dans la gestion des opérations d'usinage, il faut bien sûr prévoir l'achat et la livraison des matières premières et la vente et l'expédition des produits finis. Mais ce n'est pas tout de s'occuper du flot des produits dans la ligne de production. Les opérations d'usinage ont aussi besoin de matériel et de produits de soutien pour bien fonctionner. Il faut entre autres prévoir la disponibilité des stocks d'outil et des fluides de coupe, des pièces de rechange et des lubrifiants pour les machines.

Notre modèle d'optimisation peut fournir de l'information pour aider à la gestion de ces stocks de soutien. Tout d'abord, en fournissant les temps d'usinage, on peut prévoir les temps d'utilisation des machines. Ces temps d'utilisations permettent de prévoir les entretiens préventifs et ainsi connaître les besoins en pièces de rechange et lubrifiants pour les machines. On peut aussi de la même manière prévoir les besoins en fluide de coupe. D'une manière plus directe encore, notre modèle inclut spécifiquement la durée de vie des outils de coupe. On peut donc prévoir le nombre de pièces que l'on peut fabriquer avec un outil et le temps d'usinage total que peut effectuer un outil. Ces informations peuvent donc soutenir la gestion des stocks d'outils afin d'assurer un approvisionnement adéquat sans garder trop de stocks inutilement, ce qui, on le sait, entraîne des coûts supplémentaires.

4.2.3 Extension des modèles de contrôle

Tout un domaine de la recherche sur les systèmes de production s'occupe des stratégies de commande en temps réel. Nous avons déjà décrit de quelle façon notre modèle peut être utilisé pour fournir la fenêtre opérationnelle au système de contrôle et interpréter la commande en minimisant le coût ou le temps en fonction de celle-ci.

Dans un autre ordre d'idées, notre modèle pourrait permettre d'aller plus loin dans l'optimisation de ces modèles de contrôle. Dans la stratégie de commande à seuil critique, afin de déterminer le niveau critique de stock optimal, on minimise le coût total de maintien en inventaire et de rupture des stocks. On pose généralement comme hypothèse que le coût de fabrication est constant peu importe le taux de production. Cette hypothèse simplifie le problème et est légitime quand on ne connaît pas le système de fabrication qui est contrôlé. Or, on a présenté dans ce document un modèle qui permet de bien connaître l'opération de fraisage en bout. On a appris qu'il y a une fenêtre opérationnelle, avec un temps d'usinage minimum et donc un taux de production maximum. On est donc en mesure de vérifier la faisabilité d'un tel système. Mais on a aussi pu constater que le coût d'usinage n'est pas constant à l'intérieur de la fenêtre opérationnelle et qu'il coûte plus cher de produire plus rapidement. Alors, à partir des informations fournies par notre modèle, on pourrait étendre les modèles de contrôle et inclure le coût de production qui, on le sait désormais, n'est pas constant avec le temps d'usinage ou le taux de production dans le cas du fraisage en bout.

CONCLUSION

0Dans ce mémoire, nous avons traité principalement de la modélisation et de l'optimisation de l'opération de fraisage en bout. On a commencé par décrire en détail ce procédé de fabrication par usinage et nous l'avons situé dans son contexte industriel. Nous avons fait ressortir l'importance de modéliser le coût et le temps d'usinage en fonction de dix variables du procédé. Aucun modèle d'une même ampleur n'a été présenté précédemment dans la littérature. Les réalisations de ce travail se résument comme suit :

- Un modèle d'optimisation plus complet, incluant les fonctions objectifs et les contraintes a été élaboré. Ce modèle qui tient compte de l'ébauche et de la finition, contient dix variables de décision et plus de cinquante paramètres. Ce modèle permet d'optimiser les opérations de fraisage selon deux fonctions objectifs : coût d'usinage minimal et/ou temps d'usinage minimal.

- Le modèle a été résolu en utilisant un algorithme génétique (AG) et les résultats comparés avec le solveur commercial LINDO. Vu que ce problème est très difficile à résoudre avec les outils classiques d'optimisation, nous avons proposé un algorithme d'optimisation génétique qui permet une résolution adéquate du problème, et ce, dans un temps de calcul raisonnable.

- Nous avons présenté un exemple numérique de problème que nous avons résolu avec l'algorithme proposé. L'AG a donné un résultat adéquat (environ 1 % de différence) et en un temps plus court (60 % de moins) que le logiciel commercial LINDO.

- Nous avons aussi proposé une analyse de sensibilité pour évaluer les impacts des principales contraintes de force et de puissance maximale sur les fonctions

objectifs. Cette analyse nous a révélé que la relation n'est pas directe entre le relâchement des contraintes et l'amélioration des fonctions objectifs. Une légère amélioration est cependant possible en augmentant la force de coupe admissible et la puissance de la machine.

- Finalement, au chapitre 4, nous avons proposé des applications possibles de notre modèle dans un contexte industriel. On explique spécifiquement le rôle d'intermédiaire entre le contrôleur et la machine que peut jouer notre modèle dans un système de contrôle. On énonce ensuite quelques applications supplémentaires dans la conception des systèmes de production, la gestion des opérations et la recherche scientifique.

On a donc atteint, dans ce mémoire, les objectifs que nous nous étions fixés dans la problématique en posant le modèle mathématique qui représente bien le système d'usinage et en résolvant le problème par la réalisation d'un logiciel qui concrétise l'algorithme d'optimisation génétique proposé.

Le travail n'est bien sûr pas terminé pour pouvoir appliquer le modèle en industrie. Il faudrait développer un logiciel industriel commercialisable afin d'être utilisé. Ce logiciel pourrait contenir plusieurs modules :

- Aide à la décision en Gestion de la production.
- Effet du niveau de qualité et des temps improductifs sur la production.
- Politique de remplacement des outils.
- Gestion des commandes et suivi des pièces.
- Détermination du coût d'usinage et donc, détermination des prix de vente et aide à la préparation des soumissions.

Notre modèle pourrait aussi être développé en interface pour un système de contrôle de la production (plus ou moins automatisé) :

- Permet de déterminer la fenêtre opérationnelle des procédés d'usinage.

- Détermine les paramètres d'usinage afin d'appliquer une commande en minimisant le coût pour un taux de production demandé ou en minimisant le temps d'usinage pour un coût donné.

Pour que notre modèle soit vraiment facile à implanter en industrie, il faudra aussi développer une méthode d'application comprenant un protocole de cueillette de données et le calcul des plans d'expériences pour la durée de vie des outils et le fini de surface en bout. Il faudra aussi s'assurer de l'application des paramètres optimaux et faire le suivi des modifications. Finalement, une mesure des résultats réels des paramètres optimaux et des mesures de performance permettra de confirmer les résultats.

Ce mémoire a inspiré un article scientifique qui a été accepté pour publication dans la revue Production Planning & Control sous le titre : *A generalized model for optimizing end milling operation.*

ANNEXE 1

LE MODÈLE MATHÉMATIQUE

Voici un résumé du modèle mathématique d'optimisation, référez-vous à la section : « liste des abréviations et des sigles » (page ix) pour la description des variables et paramètres.

Minimiser :

$$T_u = \frac{LWH-(W-w_f)Ld_f-HLw_f}{w_r d_r v_{fr}} + \frac{WL}{D_f v_{ff}} + \frac{R}{v_t} + t_L + \frac{t_s}{N_L} + \frac{(LWH-(W-w_f)Ld_f-HLw_f)t_e v_r{}^{\alpha} f_r{}^{\beta} d_r{}^{r} w_r{}^{\lambda}}{kw_r d_r v_{fr}} + \frac{WLt_e v_f{}^{\alpha} f_f{}^{\beta} d_f{}^{r} w_f{}^{\lambda}}{kD_f v_{ff}} \quad \text{(A.1)}$$

Et/ou Minimiser

$$U = C_B + k_0 T_u + \left(\frac{v_r{}^{\alpha} f_r{}^{\beta} d_r{}^{r} w_r{}^{\lambda}\left(C_{pr}+(C_A \cdot N_{Ar})\right)\left(LWH-(W-w_f)Ld_f-HLw_f\right)}{kw_r d_r v_{fr}(N_{Ar}+1)}\right) + \left(\frac{v_f{}^{\alpha} f_f{}^{\beta} d_f{}^{r} w_f{}^{\lambda} WL\left(C_{pf}+(C_A \cdot N_{Af})\right)}{kD_f v_{ff}(N_{Arf}+1)}\right) \quad \text{(A.2)}$$

Et/ou Maximiser

$$P = \frac{60}{T_u} \quad \text{(A.3)}$$

Sujet à :

$$X_0 + X_1 \cdot v_{ff} - X_2 \cdot N_f \cdot v_{ff} + X_3 \cdot N_f \cdot d_f - X_4 \cdot v_{ff} \cdot d_f \leq R_{ae} \quad \text{(A.4)}$$

$$\frac{318 f_f}{4 D_f} \leq R_{ap} \quad \text{(A.5)}$$

$$d_f \cdot w_f \cdot v_{ff} \leq \frac{P_{\max} \eta}{P_s} \quad \text{(A.6)}$$

$$d_r \cdot w_r \cdot v_{fr} \leq \frac{P_{\max} \eta}{P_s} \quad \text{(A.7)}$$

$$\frac{60 \cdot P_r}{v_r} \leq F_{r\max} \quad \text{(A.8)}$$

$$\frac{60 \cdot P_r}{v_f} \leq F_{f\max} \quad \text{(A.9)}$$

$$w_r \cdot N_w + w_f = W \quad \text{(A.10)}$$

$$d_r \cdot N_p + d_f = H \quad \text{(A.11)}$$

$$w_r \leq D_r \quad \text{(A.12)}$$

$$w_f \leq D_f \quad \text{(A.13)}$$

$$d_r \leq h_{r\max} \quad \text{(A.14)}$$

$$d_f \leq h_{f\max} \quad \text{(A.15)}$$

$$v_{fr}, v_{ff} \leq v_{f\max} \quad \text{(A.16)}$$

$$\frac{1000 \cdot v_r}{\pi \cdot D_r} \leq N_{\max} \quad \text{(A.17)}$$

$$\frac{1000 \cdot v_f}{\pi \cdot D_f} \leq N_{\max} \tag{A.18}$$

$$v_r, f_r, d_r, w_r, D_r, v_f, f_f, d_f, w_f, D_f \geq 0 \tag{A.19}$$

ANNEXE 2

LE CODE DU LOGICIEL

Le fichier en-tête MillOptView.h, interface de la classe CMillOptView :

```
// MillOptView.h : interface of the CMillOptView class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_MILLOPTVIEW_H__DA72CA00_C09D_4DBB_99D4_CD5DDE6C4E93__INCLUDED_)
#define AFX_MILLOPTVIEW_H__DA72CA00_C09D_4DBB_99D4_CD5DDE6C4E93__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CMillOptView : public CScrollView
{
protected: // create from serialization only
    CMillOptView();
    DECLARE_DYNCREATE(CMillOptView)

// Attributes
public:
    CMillOptDoc* GetDocument();

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMillOptView)
    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual void OnInitialUpdate(); // called first time after construct
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    //}}AFX_VIRTUAL

// Implementation
public:
    CString SetTrailingZeros(char str[], int digits);

    bool m_bMessage1;
    int m_nTimer;

    virtual ~CMillOptView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CMillOptView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg BOOL OnMouseWheel(UINT nFlags, short zDelta, CPoint pt);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
```

```
};

#ifndef _DEBUG  // debug version in MillOptView.cpp
inline CMillOptDoc* CMillOptView::GetDocument()
   { return (CMillOptDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.//

#endif // !defined(AFX_MILLOPTVIEW_H__DA72CA00_C09D_4DBB_99D4_CD5DDE6C4E93__INCLUDED_)
```

Le fichier code MillOptView.cpp, implémentation de la classe CMillOptView :

```
// MillOptView.cpp : implementation of the CMillOptView class
//

#include "stdafx.h"
#include "MillOpt.h"

#include "MillOptDoc.h"
#include "MillOptView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMillOptView

IMPLEMENT_DYNCREATE(CMillOptView, CScrollView)

BEGIN_MESSAGE_MAP(CMillOptView, CScrollView)
    //{{AFX_MSG_MAP(CMillOptView)
    ON_WM_LBUTTONDOWN()
    ON_WM_RBUTTONDOWN()
    ON_WM_TIMER()
    ON_WM_MOUSEWHEEL()
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMillOptView construction/destruction

CMillOptView::CMillOptView()
{
    // TODO: add construction code here
    m_nTimer=0;
}

CMillOptView::~CMillOptView()
{
}

BOOL CMillOptView::PreCreateWindow(CREATESTRUCT& cs)
{
```

```
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs


    return CScrollView::PreCreateWindow(cs);
}

////////////////////////////////////////////////////////////////////////////
// CMillOptView drawing

void CMillOptView::OnDraw(CDC* pDC)
{
    CMillOptDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here

    CRect rect;
    GetClientRect(&rect);

    //Initialiser ou détruire un timer
    if(m_nTimer==0 && pDoc->m_bTimerOn==true )
    {
        m_nTimer=SetTimer(1,5,NULL);
        m_bMessage1=false;
    }
    else if (pDoc->m_bTimerOn==false && m_nTimer)
    {
        KillTimer(m_nTimer);
        m_nTimer=0;
        m_bMessage1=false;
    }

    if ( pDoc->m_iTypeAff==0) // Afficher l'écran d'introduction
    {
        pDC->SelectStockObject(WHITE_BRUSH);
        pDC->Rectangle(rect);
        pDC->TextOut(10, 10, "Type d'affichage 0 Écran d'introduction");
        if( (fopen("population.csv","w+t")) == NULL )
            exit(0);
        if( (fopen("résultats.csv","w+t")) == NULL )
            exit(0);
    }

    if ( pDoc->m_iTypeAff==1) // Afficher l'écran Type 1
    {
        pDC->SelectStockObject(WHITE_BRUSH);
        pDC->Rectangle(rect);
        pDC->TextOut(10, 10, "Type d'affichage 1 Click Annuler");

    }

    if ( pDoc->m_iTypeAff==2) // Afficher l'écran Type 2 (OnOK)
    {

        pDC->SelectStockObject(WHITE_BRUSH);
        pDC->Rectangle(rect);
        char buffer[12];
        int posY=10;
        double pourcent=0;
        _itoa( pDoc->m_iGen, buffer, 10 );
        pDC->TextOut(10, posY, buffer);
        CString s;
        pDC->TextOut(50, posY, "générations"), posY=posY+20;
        s = pDoc->TempsOpt.Format( "%D jours, %H:%M:%S" );
        pDC->TextOut(10, posY, "Temps écoulé:");
        pDC->TextOut(110, posY, s), posY=posY+30;
        s.Empty();
```

```
// Afficher le résumé des meilleurs résultats
pDC->TextOut(10, posY, "Meilleurs Résultats:"), posY=posY+20;
s="        U              Tu           FitMoy       DeltaFit
      Converge       Lim. U    Lim. Tu    PdsCont";
pDC->TextOut(10, posY, s), posY=posY+20;
s.Empty();
_gcvt( pDoc->m_dOptU[2], 8, buffer );
s+=SetTrailingZeros(buffer, 9);
s+="  ";
_gcvt( pDoc->m_dOptTu[3], 8, buffer );
s+=SetTrailingZeros(buffer, 9);
s+="  ";

_gcvt( pDoc->m_dFitMoy, 8, buffer );
s+=SetTrailingZeros(buffer, 9);
s+="  ";

_gcvt( pDoc->m_dDeltaFit, 8, buffer );
s+=SetTrailingZeros(buffer, 9);
s+="         ";

if (pDoc->m_iObjectifMin==2)
    pourcent=(double)pDoc->m_iGenPc/(double)pDoc->m_iNbGenPc*50;
else
    pourcent=(double)pDoc->m_iGenPc/(double)pDoc->m_iNbGenPc*100;
if (pDoc->m_bMinTuOk==true && pDoc->m_iObjectifMin==2)
    pourcent+=50;
if (pDoc->m_bMinTuOk==true && pDoc->m_iObjectifMin==0)
    pourcent=100;
_itoa( pourcent, buffer, 10 );
s+=buffer;
s+=" %";

s+="          ";
_gcvt( pDoc->m_dMinU, 5, buffer );
s+=SetTrailingZeros(buffer, 6);
s+="  ";
_gcvt( pDoc->m_dMinTu, 5, buffer );
s+=SetTrailingZeros(buffer, 6);

s+="    ";
_gcvt( pDoc->m_dPoidsCont, 4, buffer );
s+=SetTrailingZeros(buffer, 5);

pDC->TextOut(10, posY, s), posY=posY+30;

// Afficher la meilleure solution pour U
pDC->TextOut(10, posY, "Meilleure Solution pour U:"), posY=posY+20;
s=" #ID          fit           U            Tu          Dr
      Df          vr           vf            fr            ff
          Npr          df        Npwr        Wf";
pDC->TextOut(10, posY, s), posY=posY+20;
s.Empty();

_itoa( pDoc->m_dOptU[0], buffer, 10 );
s+=buffer;
s+="   ";
for (int j=1; j<14; j++)
{
    _gcvt( pDoc->m_dOptU[j], 5, buffer );
    if(j!=0)
        s+=SetTrailingZeros(buffer, 6);
    else
        s+=buffer;
    s+="   ";
}
```

```
        pDC->TextOut(10, posY, s), posY=posY+30;

        // Afficher la meilleure solution pour Tu
        pDC->TextOut(10, posY, "Meilleure Solution pour Tu:"), posY=posY+20;
        s=" #ID          fit          U            Tu          Dr
              Df          vr          vf            fr            ff
                  Npr          df        Npwr        Wf";
        pDC->TextOut(10, posY, s), posY=posY+20;
        s.Empty();

        _itoa( pDoc->m_dOptTu[0], buffer, 10 );
        s+=buffer;
        s+="   ";
        for (j=1; j<14; j++)
        {
            _gcvt( pDoc->m_dOptTu[j], 5, buffer );
            if(j!=0)
                s+=SetTrailingZeros(buffer, 6);
            else
                s+=buffer;
            s+="   ";
        }
        pDC->TextOut(10, posY, s), posY=posY+30;

        // Afficher les contraintes pour U
        s=" Contraintes :   Puissance      Force          Fini           Cible";
        pDC->TextOut(10, posY, s), posY=posY+20;
        s="        Pour U :      ";
        for (j=0; j<4; j++)
        {
            _gcvt( pDoc->m_dContAct[1][j], 6, buffer );
            s+=SetTrailingZeros(buffer, 7);
            s+="     ";
        }
        pDC->TextOut(12, posY, s), posY=posY+20;
        // Afficher les contraintes pour Tu
        s="        Pour Tu :      ";
        for (j=0; j<4; j++)
        {
            _gcvt( pDoc->m_dContAct[0][j], 6, buffer );
            s+=SetTrailingZeros(buffer, 7);
            s+="     ";
        }
        pDC->TextOut(10, posY, s), posY=posY+30;


        if (!m_nTimer)
            pDC->TextOut(10, posY, "Optimisation Terminée !");
//          MessageBox("Optimisation Terminée !");
//          m_bMessage1=true;
    }
    if ( pDoc->m_iTypeAff==3) // Afficher l'écran Type 3 (PassW Err)
    {

        pDC->SelectStockObject(WHITE_BRUSH);
        pDC->Rectangle(rect);
        pDC->TextOut(10, 10, "Mot de passe incorrect");
    }
    if ( pDoc->m_iTypeAff==4) // Afficher l'écran Type 4 (Initialisation)
    {

        pDC->SelectStockObject(WHITE_BRUSH);
        pDC->Rectangle(rect);
        char buffer[12];
        int posY=10;
        double pourcent=(pDoc->m_iIndex)/(double)(pDoc->m_iPop)*100;
        CString s;
```

```
        s = pDoc->TempsOpt.Format( "%D jours, %H:%M:%S" );
        pDC->TextOut(10, posY, "Temps écoulé:");
        pDC->TextOut(110, posY, s), posY=posY+30;
        s.Empty();

        s="Initialisation : ";

        _itoa( pourcent, buffer, 10 );
        s+=buffer;
        s+=" %";
        pDC->TextOut(10, posY, s), posY=posY+30;
    }
}

void CMillOptView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();

    CSize sizeTotal;
    // TODO: calculate the total size of this view
    sizeTotal.cx = sizeTotal.cy = 100;
    SetScrollSizes(MM_TEXT, sizeTotal);
}

/////////////////////////////////////////////////////////////////////////////
// CMillOptView printing

BOOL CMillOptView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CMillOptView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CMillOptView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

/////////////////////////////////////////////////////////////////////////////
// CMillOptView diagnostics

#ifdef _DEBUG
void CMillOptView::AssertValid() const
{
    CScrollView::AssertValid();
}

void CMillOptView::Dump(CDumpContext& dc) const
{
    CScrollView::Dump(dc);
}

CMillOptDoc* CMillOptView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMillOptDoc)));
    return (CMillOptDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMillOptView message handlers
```

```
void CMillOptView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    CScrollView::OnLButtonDown(nFlags, point);
}

BOOL CMillOptView::OnMouseWheel(UINT nFlags, short zDelta, CPoint pt)
{
    // TODO: Add your message handler code here and/or call default
    CMillOptDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    double increment=zDelta/120*.01; // zDelta/120 = -1 ou 1
    pDoc->m_dPoidsCont+=increment;
    return CScrollView::OnMouseWheel(nFlags, zDelta, pt);
}

void CMillOptView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CMillOptDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    pDoc->exit=true;

    CScrollView::OnRButtonDown(nFlags, point);
}

void CMillOptView::OnTimer(UINT nIDEvent)
{
    CMillOptDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    if(pDoc->m_bTimerOn==true)
    {
        CTime Temps = CTime::GetCurrentTime();
        pDoc->TempsOpt=Temps-pDoc->TempsDebut ;
        if(pDoc->m_iGen==0)
            pDoc->Initialise();
        else
            pDoc->Optimise();
    }
    InvalidateRect(NULL,FALSE);

    CScrollView::OnTimer(nIDEvent);
}


 CString CMillOptView::SetTrailingZeros(char str[], int digits)
{
    bool FormExp=false;
    for (int i=0; i<11; i++)
    {
        if (str[i]==101)//ASCII 101="e"
            FormExp=true;
    }
    if(FormExp==true)
    {
        double nombre=atof(str);
        sprintf( str , "%f\n", nombre );
    }


    for (i=0; i<11; i++)
    {                   //ASCII 46="." 100="d"
        if (i<digits && (str[i]<45 || str[i]>101))
            str[i]=48; //ASCII 48="0"
        if (i>=digits )
```

```
            str[i]=0;
        }
        return str;
}
```

Le fichier en-tête MillOptDoc.h, interface de la classe CMillOptDoc :

```
// MillOptDoc.h : interface of the CMillOptDoc class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_MILLOPTDOC_H__0BF97D1A_9077_4013_8437_71C3EACD4473__INCLUDED_)
#define AFX_MILLOPTDOC_H__0BF97D1A_9077_4013_8437_71C3EACD4473__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "nr.h"

class CMillOptDoc : public CDocument
{
protected: // create from serialization only
    CMillOptDoc();
    DECLARE_DYNCREATE(CMillOptDoc)

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMillOptDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    //}}AFX_VIRTUAL

// Implementation
public:
    void RaffNum(int obj);
    int m_iContBusted;
    void Initialise();
    void SetLimitInf();
    void SetContAct();
    int m_iObjectifMin;
    void CheckVarMinMax(Vec_IO_SP &individu);
    void Mut_Grad_U(Vec_IO_SP &individu);
    double m_dForcemax;
    double m_dPoidsCont;
    double m_dPoidsContInit;
    int m_iAGisStalled;
    int arrondi (double reel);
    void SetDiscPopInit(int NbInc);
    void Set_minU_minTu(Mat_IO_SP Population);
    double SetFitCont(Vec_IO_SP individu);
    void SetPopFitness(Mat_IO_SP &Population);
    void FichierResult();

    double  m_dFmax;
    int     m_iMutNbGenes;
    double  m_dMutPcEnfants;
```

```
    double  m_dMutPcGene;
    int     m_iModelRPMmax;
    double  m_dModelPmax;
    double  m_dModelRa;
    BOOL    m_bMinTuPourU;
    BOOL    m_bMinUPourTu;
    double  m_dCibleTu;
    double  m_dCibleU;

    double m_dVarMinMax[2][10];
    double m_dContAct[2][4];

    CString m_sPassw;
    int m_iNbGenPc, m_iGenPc;
    double m_dPcAmel;
    double m_dMinU, m_dMinUback;
    double m_dMinTu, m_dMinTuback;
    double m_dMaxFit, m_dMinFit, m_dDeltaFit, m_dMaxFitback, m_dFitMoy,  m_dFitMoyback;

    double m_dOptU[14];
    double m_dOptTu[14];
    double m_dBestFit[14];

    double M, Pi, epsilon;
    bool m_bMinTuOk;

    void Set_U_P_Fit(Vec_IO_SP &individu);
    void InduviduAlea(Vec_IO_SP &individu);
    double alea(int type, double min, double max, double param1);
    void FichierPop(Mat_IO_SP Population);
    void FichierLog();
    CString RemplPoint(CString str);
    void Optimise();

    int m_iIndex;

    CTime TempsDebut;
    CTimeSpan TempsOpt;
    Mat_IO_SP m_Population;
    Mat_IO_SP m_PopulInit;

    FILE *m_FdebugFile;

    int     m_iGenMax;
    int     m_iPop;
    int     m_iPopInit;
    int     m_iTempsMax;

    bool m_bTimerOn;
    int m_iTypeAff;
    int m_iPourc;
    int m_iTemps;
    int m_iGen;

    int idum;
    bool exit;

    virtual ~CMillOptDoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
```

```
    //{{AFX_MSG(CMillOptDoc)
    afx_msg void OnOptimiser();
    afx_msg void OnOptimisationModle();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_MILLOPTDOC_H__0BF97D1A_9077_4013_8437_71C3EACD4473__INCLUDED_)
```

Le fichier code MillOptDoc.cpp, implémentation de la classe CMillOptDoc :

```
// MillOptDoc.cpp : implementation of the CMillOptDoc class
//

#include "stdafx.h"
#include "MillOpt.h"

#include "MillOptDoc.h"

// ----------- *** Rajouté manuellement -Début- *** ---------------
#include "DlgOptimise.h"
#include "dlgModeleMachineOp.h"



// ----------- *** Rajouté manuellement -Fin- *** -----------------


#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMillOptDoc

IMPLEMENT_DYNCREATE(CMillOptDoc, CDocument)

BEGIN_MESSAGE_MAP(CMillOptDoc, CDocument)
    //{{AFX_MSG_MAP(CMillOptDoc)
    ON_COMMAND(ID_OPTIMISATION_OPTIMISER, OnOptimiser)
    ON_COMMAND(ID_OPTIMISATION_MODLE, OnOptimisationModle)
    ON_COMMAND(ID_BUTTON_OPTIMISER, OnOptimiser)
    ON_COMMAND(ID_BUTTON_MODELE, OnOptimisationModle)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMillOptDoc construction/destruction

CMillOptDoc::CMillOptDoc()
{
    // TODO: add one-time construction code here

    m_bMinTuOk=false;
    m_iTypeAff=0;
```

```
    m_iPourc=0;
    m_iTemps=0;
    m_bTimerOn=false;
    exit=false;
    M=1000000000;
    Pi=3.14159265358979;
    epsilon=.0001;
    TempsDebut = CTime::GetCurrentTime();
    srand( (unsigned)time( NULL ) );
    idum=-(rand());

    m_iGenMax=0;
    m_iPop=0;
    m_iPopInit=0;
    m_iTempsMax=0;
    m_iIndex=0;
    m_dMinU=M;
    m_dMinTu=M;
    m_dMinUback=M;
    m_dMinTuback=M;
    m_sPassw="";
    m_iNbGenPc=0;
    m_dPcAmel=0;

    m_dFmax=0;
    m_iMutNbGenes=0;
    m_dMutPcEnfants=0;
    m_dMutPcGene=0;
    m_iModelRPMmax=0;
    m_dModelPmax=0;
    m_dForcemax=0;
    m_dModelRa=0;
    m_iObjectifMin=2;
    m_bMinTuPourU=false;
    m_bMinUPourTu=false;
    m_dCibleTu=0;
    m_dCibleU=0;

    for(int i=0; i<10; i++)
        m_dVarMinMax[0][i]=epsilon, m_dVarMinMax[1][i]=M;
    for(i=0; i<2; i++)
        for(int j=0; j<4; j++)
            m_dContAct[i][j]=0;

    m_dPoidsCont=0;
    m_dPoidsContInit=m_dPoidsCont;

}

CMillOptDoc::~CMillOptDoc()
{
}

BOOL CMillOptDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    // TODO: add reinitialization code here

    m_bMinTuOk=false;
    m_iTypeAff=0;
    m_iPourc=0;
    m_iTemps=0;
    m_bTimerOn=false;
    exit=false;
    TempsDebut = CTime::GetCurrentTime();
    srand( (unsigned)time( NULL ) );
```

```
    idum=-(rand());

    m_iGenMax=9000;
    m_iPopInit=100;
    m_iPop=m_iPopInit;
    m_iTempsMax=160;
    m_iIndex=0;
    m_dMinU=M;
    m_dMinTu=M;
    m_dMinUback=M;
    m_dMinTuback=M;
    m_sPassw="etsmec";
    m_iNbGenPc=200;
    m_dPcAmel=.010;
    m_iGenPc=0;

    m_dFmax=8000;
    m_iMutNbGenes=8;
    m_dMutPcEnfants=75;
    m_dMutPcGene=10;
    m_iModelRPMmax=12000;
    m_dModelPmax=7.5;
    m_dForcemax=2000;
    m_dModelRa=2;
    m_iObjectifMin=2;
    m_bMinTuPourU=false;
    m_bMinUPourTu=false;
    m_dCibleTu=1.4374;
    m_dCibleU=9.3616;

    for(int i=0; i<2; i++)
        for(int j=0; j<4; j++)
            m_dContAct[i][j]=0;

    m_dVarMinMax[0][0]=10, m_dVarMinMax[1][0]=18; // Dr
    m_dVarMinMax[0][1]=10, m_dVarMinMax[1][1]=18; // Df
    m_dVarMinMax[0][2]=50, m_dVarMinMax[1][2]=450; // vr
    m_dVarMinMax[0][3]=50, m_dVarMinMax[1][3]=450; // vf
    m_dVarMinMax[0][4]=0.01, m_dVarMinMax[1][4]=1; // fr
    m_dVarMinMax[0][5]=0.01, m_dVarMinMax[1][5]=0.15; // ff
    m_dVarMinMax[0][6]=1, m_dVarMinMax[1][6]=12; // Npdr
    m_dVarMinMax[0][7]=1.0, m_dVarMinMax[1][7]=3.0; // df
    m_dVarMinMax[0][8]=2, m_dVarMinMax[1][8]=10; // Npwr
    m_dVarMinMax[0][9]=1.0, m_dVarMinMax[1][9]=3.0; // wf


    m_dPoidsContInit=.50;

    // (SDI documents will reuse this document)

    return TRUE;
}


/////////////////////////////////////////////////////////////////////////////
// CMillOptDoc serialization

void CMillOptDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
```

```
    }
}

/////////////////////////////////////////////////////////////////////////////
// CMillOptDoc diagnostics

#ifdef _DEBUG
void CMillOptDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CMillOptDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMillOptDoc commands

void CMillOptDoc::OnOptimisationModle()
{

    CdlgModeleMachineOp Dlg;
    //Transfert des variables Doc -> Dlg
    Dlg.m_dFmax=m_dFmax;
    Dlg.m_iModelRPMmax=m_iModelRPMmax;
    Dlg.m_dModelPmax=m_dModelPmax;
    Dlg.m_dForcemax=m_dForcemax;
    Dlg.m_dModelRa=m_dModelRa;

    Dlg.m_dDrMin=m_dVarMinMax[0][0], Dlg.m_dDrMax=m_dVarMinMax[1][0]; // Dr
    Dlg.m_dDfMin=m_dVarMinMax[0][1], Dlg.m_dDfMax=m_dVarMinMax[1][1]; // Df
    Dlg.m_dVrMin=m_dVarMinMax[0][2], Dlg.m_dVrMax=m_dVarMinMax[1][2]; // vr
    Dlg.m_dVfMin=m_dVarMinMax[0][3], Dlg.m_dVfMax=m_dVarMinMax[1][3]; // vf
    Dlg.m_dFrMin=m_dVarMinMax[0][4], Dlg.m_dFrMax=m_dVarMinMax[1][4]; // fr
    Dlg.m_dFfMin=m_dVarMinMax[0][5], Dlg.m_dFfMax=m_dVarMinMax[1][5]; // ff
    Dlg.m_iNpdMin=m_dVarMinMax[0][6], Dlg.m_iNpdMax=m_dVarMinMax[1][6]; // Npdr
    Dlg.m_ddfMin=m_dVarMinMax[0][7], Dlg.m_ddfMax=m_dVarMinMax[1][7]; // df
    Dlg.m_iNpwMin=m_dVarMinMax[0][8], Dlg.m_iNpwMax=m_dVarMinMax[1][8]; // Npwr
    Dlg.m_dwfMin=m_dVarMinMax[0][9], Dlg.m_dwfMax=m_dVarMinMax[1][9]; // wf


    if (Dlg.DoModal()==IDOK)
    {
        //Transfert des variables Dlg -> Doc
        m_dFmax=Dlg.m_dFmax;
        m_iModelRPMmax=Dlg.m_iModelRPMmax;
        m_dModelPmax=Dlg.m_dModelPmax;
        m_dForcemax=Dlg.m_dForcemax;
        m_dModelRa=Dlg.m_dModelRa;

        m_dVarMinMax[0][0]=Dlg.m_dDrMin, m_dVarMinMax[1][0]=Dlg.m_dDrMax; //Dr
        m_dVarMinMax[0][1]=Dlg.m_dDfMin, m_dVarMinMax[1][1]=Dlg.m_dDfMax; //Df
        m_dVarMinMax[0][2]=Dlg.m_dVrMin, m_dVarMinMax[1][2]=Dlg.m_dVrMax; //vr
        m_dVarMinMax[0][3]=Dlg.m_dVfMin, m_dVarMinMax[1][3]=Dlg.m_dVfMax; //vf
        m_dVarMinMax[0][4]=Dlg.m_dFrMin, m_dVarMinMax[1][4]=Dlg.m_dFrMax; //fr
        m_dVarMinMax[0][5]=Dlg.m_dFfMin, m_dVarMinMax[1][5]=Dlg.m_dFfMax; //ff
        m_dVarMinMax[0][6]=Dlg.m_iNpdMin, m_dVarMinMax[1][6]=Dlg.m_iNpdMax; //Npdr
        m_dVarMinMax[0][7]=Dlg.m_ddfMin, m_dVarMinMax[1][7]=Dlg.m_ddfMax; //df
        m_dVarMinMax[0][8]=Dlg.m_iNpwMin, m_dVarMinMax[1][8]=Dlg.m_iNpwMax; //Npwr
        m_dVarMinMax[0][9]=Dlg.m_dwfMin, m_dVarMinMax[1][9]=Dlg.m_dwfMax; //wf
    }

    UpdateAllViews(NULL);
```

```
} // FIN CMillOptDoc::OnOptimisationModle

void CMillOptDoc::OnOptimiser()
{
    CDlgOptimise Dlg;

    //Transfert des variables Doc -> Dlg
    Dlg.m_iGenMax=m_iGenMax;
    Dlg.m_iPop=m_iPopInit;
    Dlg.m_iTempsMax=m_iTempsMax;
    Dlg.m_sPassw=m_sPassw;
    Dlg.m_iNbGenPc=m_iNbGenPc;
    Dlg.m_dPcAmel=m_dPcAmel;

    Dlg.m_iMutNbGenes=m_iMutNbGenes;
    Dlg.m_dMutPcEnfants=m_dMutPcEnfants;
    Dlg.m_dMutPcGene=m_dMutPcGene;
    Dlg.m_iObjectifMin=m_iObjectifMin;
    Dlg.m_bMinTuPourU=m_bMinTuPourU;
    Dlg.m_bMinUPourTu=m_bMinUPourTu;
    Dlg.m_dCibleTu=m_dCibleTu;
    Dlg.m_dCibleU=m_dCibleU;
    Dlg.m_dPoidsCont=m_dPoidsContInit;

    if (Dlg.DoModal()==IDOK)
    {
        //Transfert des variables Dlg -> Doc
        m_iGenMax=Dlg.m_iGenMax;
        m_iPopInit=Dlg.m_iPop;
        m_iPop=m_iPopInit;
        m_iTempsMax=Dlg.m_iTempsMax;
        m_sPassw=Dlg.m_sPassw;
        m_iNbGenPc=Dlg.m_iNbGenPc;
        m_dPcAmel=Dlg.m_dPcAmel;

        m_iMutNbGenes=Dlg.m_iMutNbGenes;
        m_dMutPcEnfants=Dlg.m_dMutPcEnfants;
        m_dMutPcGene=Dlg.m_dMutPcGene;
        m_iObjectifMin=Dlg.m_iObjectifMin;
        m_bMinTuPourU=Dlg.m_bMinTuPourU;
        m_bMinUPourTu=Dlg.m_bMinUPourTu;
        m_dCibleTu=Dlg.m_dCibleTu;
        m_dCibleU=Dlg.m_dCibleU;
        m_dPoidsContInit=Dlg.m_dPoidsCont;
        m_dPoidsCont=m_dPoidsContInit;

        // Réinitialisation des variables
        m_bTimerOn=true;
        m_iGen=0;
        m_iTemps=0;
        m_iIndex=0;
        m_iGenPc=0;
        exit=false;
        m_bMinTuOk=false;
        for(int i=0; i<14; i++)
            m_dOptU[i]=M, m_dOptTu[i]=M, m_dBestFit[i]=0;
        m_dOptU[1]=0, m_dOptTu[1]=0;
        m_dMinU=M;
        m_dMinTu=M;
        m_dMaxFit=0;
        m_dDeltaFit=M;
        m_dMinFit=M;
        m_dMinUback=M;
        m_dMinTuback=M;
        m_dMaxFitback=0;
        m_dFitMoy=.5;
        m_dFitMoyback=.5;
```

```
        for(i=0; i<2; i++)
            for(int j=0; j<4; j++)
                m_dContAct[i][j]=0;
        m_iAGisStalled=0;
        m_iContBusted=0;

        // Réinitialiser le fichier population
        m_FdebugFile=fopen("population.csv","w+t");
        fclose(m_FdebugFile);
        m_FdebugFile=fopen("résultats.csv","w+t");
        fclose(m_FdebugFile);

        // Dimentionner les matrices de population
        Mat_IO_SP PopTemp(m_iPop,14);
        m_Population=PopTemp;
        m_PopulInit=PopTemp;

    if (m_sPassw=="etsmec")
    {
        // Réinitialiser le temps
        CTime Temps = CTime::GetCurrentTime();
        TempsDebut = CTime::GetCurrentTime();
        TempsOpt=Temps-TempsDebut;
        FichierLog();
    }
    else
        m_iTypeAff=3;
    }
    else
        m_iTypeAff=1;

    UpdateAllViews(NULL);
} // FIN CMillOptDoc::OnOptimiser

void CMillOptDoc::Initialise()
{   // Générer la population initiale
    if(exit==false)
    {
        if (m_iIndex==m_iPop)
        {
            FichierPop(m_PopulInit);
            Set_minU_minTu(m_PopulInit);
            SetPopFitness(m_PopulInit);
            m_Population=m_PopulInit;
            FichierResult();
            m_iGen=1, m_iTypeAff=2;
        }
        else
        {
            Vec_IO_SP IndividuTemp(14);// Vecteur individu temporaire

            // Initialiser la population version 1 (aléatoire avec Fit>cst)
            IndividuTemp[0]=m_iIndex++;
            do
            {
                InduviduAlea(IndividuTemp);
            }   while   (IndividuTemp[1]<.40      ||   IndividuTemp[3]>(m_dMinTu*3)   ||
                  IndividuTemp[2]>(m_dMinU*3));

            // Fixer les bornes initiales
            if (m_dMinTu>IndividuTemp[3])
                m_dMinTu=IndividuTemp[3];
            if (m_dMinU>IndividuTemp[2])
                m_dMinU=IndividuTemp[2];
```

```
            for(int j=0; j<14; j++)
                m_PopulInit[(m_iIndex-1)][j]=IndividuTemp[j];

            m_iTypeAff=4;
        }
    }
    else
        m_bTimerOn=false;

} // FIN CMillOptDoc::Initialise

void CMillOptDoc::Optimise()
{//************* Dédut de l'optimisation ******************

    int jrs=TempsOpt.GetDays();
    int hrs=TempsOpt.GetHours();
    int mins=TempsOpt.GetMinutes();
    int secs=TempsOpt.GetSeconds();

    m_iTemps=((jrs*24+hrs)*60+mins)*60+secs;

    if (m_iObjectifMin==1)
        m_bMinTuOk=true;
    if (m_iObjectifMin==0 && m_bMinTuOk==true)
        exit=true;

    Vec_IO_SP IndividuTemp(14);
    // injecter un individu particulier dans le syst'eme
    if(m_iGen==2)
    {
        IndividuTemp[0]=1000; //Index
        IndividuTemp[1]=0; //Index
        IndividuTemp[2]=M; //Index
        IndividuTemp[3]=M; //Index
        IndividuTemp[4]=14.5; //Dr
        IndividuTemp[5]=18; //Df
        IndividuTemp[6]=225.0f; //Vr
        IndividuTemp[7]=177.2f; //Vf
        IndividuTemp[8]=0.037f; //Fr
        IndividuTemp[9]=0.150f; //Ff
        IndividuTemp[10]=2.0f; //Npdr
        IndividuTemp[11]=3.0f; //df
        IndividuTemp[12]=2.0f; //wr
        IndividuTemp[13]=3.0f; //Npwr
        Set_U_P_Fit(IndividuTemp);
        for(int j=0; j<14; j++)
            m_Population[10][j]=IndividuTemp[j];

        IndividuTemp[0]=1001; //Index
        IndividuTemp[1]=0; //Index
        IndividuTemp[2]=M; //Index
        IndividuTemp[3]=M; //Index
        IndividuTemp[4]=14.5; //Dr
        IndividuTemp[5]=12.4f; //Df
        IndividuTemp[6]=147.7f; //Vr
        IndividuTemp[7]=132.2f; //Vf
        IndividuTemp[8]=0.037f; //Fr
        IndividuTemp[9]=0.150f; //Ff
        IndividuTemp[10]=2.0f; //Npdr
        IndividuTemp[11]=3.0f; //df
        IndividuTemp[12]=2.0f; //wr
        IndividuTemp[13]=3.0f; //Npwr
        Set_U_P_Fit(IndividuTemp);
        for(j=0; j<14; j++)
            m_Population[11][j]=IndividuTemp[j];
    }
```

```
    // Critères d'arrêts relatifs
    if(m_bMinTuOk==true) // Le minimum a baissé pour U
        if (m_dOptU[2]<m_dMinUback)
            m_iGenPc=0;
    if(m_bMinTuOk==false) // Le minimum a baissé pour Tu
        if (m_dOptTu[3]<m_dMinTuback)
            m_iGenPc=0;
    if(m_dDeltaFit>m_dPcAmel)
        m_iGenPc=0; // La population n'est pas encore homogène
    m_iGenPc++;
//  if(m_dFitMoy>.999)
//      m_iGenPc++; // Le fitness moyen est élevé

    if (m_iGenPc==(m_iNbGenPc-1) && m_dMaxFit<.99)
    {   // On est stallé si le maxfit est bas et qu'on ne s'améliore pas
        m_iGenPc--;
        m_iAGisStalled++;
    }
    if (m_iAGisStalled>=5 && m_dPoidsCont<.7)
    {   // Si on est stallé et que la limite inférieure
            est atteinte, augmenter le pdsCont
        if((m_bMinTuOk==false && m_dOptTu[3]==m_dMinTu) ||
                (m_bMinTuOk==true && m_dOptU[2]==m_dMinU))
            m_dPoidsCont+=.01, m_iGenPc=0, m_iAGisStalled=0;
        if((m_bMinTuOk==false && (m_dContAct[0][0]>1.5 || m_dContAct[0][1]>1.5))
            || (m_bMinTuOk==true && (m_dContAct[1][0]>1.5 || m_dContAct[1][1]>1.5)))
            m_dPoidsCont+=.01, m_iGenPc=0, m_iAGisStalled=0;
    }
    if (m_iAGisStalled>10) // AG est vraiment stallé
    {
        m_iAGisStalled=0, m_iGenPc=0;
        if(m_dDeltaFit<.008)
        {
            m_Population=m_PopulInit;
            for (int j=0; j<14; j++)
                m_Population[1][j]=m_dOptU[j], m_Population[2][j]=m_dOptTu[j];
        }
        SetLimitInf();
    }

    if(m_bMinTuOk==false && (m_dContAct[0][0]>1.1 || m_dContAct[0][1]>1.1))
        m_iContBusted++;
    if(m_bMinTuOk==true && (m_dContAct[1][0]>1.1 || m_dContAct[1][1]>1.1))
        m_iContBusted++;
    if(m_iContBusted>10 && m_dPoidsCont<.7)
        m_dPoidsCont+=.002, m_iContBusted=0;

    if (m_iGenPc>=m_iNbGenPc && m_bMinTuOk==false)
    {
        m_bMinTuOk=true, m_iGenPc=0, m_dMaxFit=0, m_dMinFit=M, m_iPop=m_iPopInit;
        m_dPoidsCont=m_dPoidsContInit, m_Population=m_PopulInit;
        for (int j=0; j<14; j++)
            m_Population[1][j]=m_dOptU[j];
    }

    if (m_iGenPc>=m_iNbGenPc && m_bMinTuOk==true )
        exit=true;

    if ( m_iGen < m_iGenMax && m_iTemps < m_iTempsMax && exit == false)
    {
        m_dFitMoyback=m_dFitMoy;   // Pour le critère d'arrêt
        m_dMinUback=m_dOptU[2];
        m_dMinTuback=m_dOptTu[3];

        m_iGen++;

//************* Dédut de la reproduction *************
```

```
// Sélection pour la reproduction
Mat_IO_SP couples(m_iPop,2);

if (m_iGenPc>0)
{
    // Sélection version 1 (complètement alétoire)
    for(int i=0; i<m_iPop; i++)
    {
        int male=alea(0, 0, m_iPop, 0);
        int femelle=alea(0, 0, m_iPop, 0);
        couples[i][0]=male;
        couples[i][1]=femelle;
    }
}
else
{

    // Sélection version 2 (élitisme selon fitness)
    // Calcul du cumul.
    Vec_IO_SP Cumul(m_iPop);
    double cumultotal=0;
    Cumul[0]=m_Population[0][1];
    for(int i=1; i<m_iPop; i++)
        Cumul[i]=Cumul[i-1]+m_Population[i][1], cumultotal=Cumul[i];

// Sélection ver. 2 (assignation sur cumul)
    for(i=0; i<m_iPop; i++)
    {
        double male=alea(0, 0, Cumul[m_iPop-1], 0);
        double femelle;
        int j=0, k=0;
        while (male>Cumul[j])
                j++;
        do
        {
                k=0;
                femelle=alea(0, 0, Cumul[m_iPop-1], 0);
                while (femelle>Cumul[k])
                    k++;
        } while(j==k);

        couples[i][0]=j;
        couples[i][1]=k;

    }
}

// Reproduction version moyenne pondérée aléatoire
Mat_IO_SP enfants(m_iPop,14);
double FitTemp=m_dFitMoy/3;

for(int i=0; i<m_iPop; i++)
{
    double poid=.5;
    enfants[i][0]=++m_iIndex;
    for(int j=4; j<14; j++)
    {
        poid=alea(0, 0, 1, 0);
        enfants[i][j]=(m_Population[(couples[i][0])][j]*poid+
            (m_Population[(couples[i][1])][j]*(1-poid)));
        if (j==10 || j==12)
            enfants[i][j]=arrondi(enfants[i][j]);
        IndividuTemp[j]=enfants[i][j];
    }
    IndividuTemp[1]=0;
    Set_U_P_Fit(IndividuTemp); // Le fitness ne tient compte que des contraintes
```

```
        // Éliminer un enfant qui n'est pas intéressant
            double DebugFit=IndividuTemp[1];
            if (IndividuTemp[1]<(FitTemp) && i>0)
                i--, m_iIndex--, FitTemp*=.995;
        }

//************* Fin de la reproduction *************


//************* Début de la mutation *************
        // Mutation
        double Amplimut=m_dMutPcGene/100;

        for(i=0; i<m_iPop; i++)
        {
            Vec_IO_SP IndividuTemp(14);

            if (alea(0, 0, 100, 0)<m_dMutPcEnfants)
            {
                int GeneMut[10], GeneAlea;
                for (int j=0; j<m_iMutNbGenes; j++)
                {   // si le nombre à muter = 10 on les mute tous
                    if (m_iMutNbGenes==10)
                        GeneMut[j]=(j+4);
                    else
                    {       // sinon, on remplit un vecteur sans doublons
                        bool nouveau=true;
                        do
                        {
                            nouveau=true;
                            GeneAlea=int(alea(0, 4, 14, 0));
                            for (int k=0; k<j; k++)
                            {
                                if(GeneAlea==GeneMut[k])
                                    nouveau=false;
                            }
                        }  while(nouveau==false);
                        GeneMut[j]=GeneAlea;
                    }
                }

                for (j=0; j<m_iMutNbGenes; j++)// on effectue la mutation
                {   // on annule la mutation si une variable est sur un min ou un max
                 if(enfants[i][GeneMut[j]]>(m_dVarMinMax[0][(GeneMut[j]-
                    4)]+(m_dVarMinMax[1][(GeneMut[j]-4)]*.01))                          &&
                    enfants[i][GeneMut[j]]<(m_dVarMinMax[1][(GeneMut[j]-4)]*.99))
                    {
                        if (GeneMut[j]==10 || GeneMut[j]==12)
                        {
                            double mut=(alea(1, -1, 1, 0));
                            if (mut==0)
                                mut=-1;
                            enfants[i][GeneMut[j]]+=mut;
                        }
                    //  else if (GeneMut[j]==11 || GeneMut[j]==13)
                    //      enfants[i][GeneMut[j]]=enfants[i][GeneMut[j]]*
                                alea(0, (1-(Amplimut/2)), (1+(Amplimut*2)), 0);
                        else
                            enfants[i][GeneMut[j]]=enfants[i][GeneMut[j]]*
                                alea(0, (1-Amplimut), (1+Amplimut), 0);
                    }
                }
            }


            for(int j=4; j<14; j++)
            {
```

```
                if (enfants[i][j]<epsilon) // non-négativité
                    enfants[i][j]=epsilon;
            }

//************* Fin de la mutation *************

//************* Début de l'Évaluation *************

            // Évaluation
//          Vec_IO_SP IndividuTemp2(14);
            for(j=0; j<14; j++)
            {
                IndividuTemp[j]=enfants[i][j];
//              IndividuTemp2[j]=enfants[i][j];
            }

            // Vérifier que les valeurs sont dans les min-max
            CheckVarMinMax(IndividuTemp);

            IndividuTemp[1]=0; // Fitness vierge
//          IndividuTemp2[1]=0;

            Set_U_P_Fit(IndividuTemp);
//          double Uavant=IndividuTemp[2], Fitav=IndividuTemp[1];

             // Mutation selon le gradient
/*          if(m_dFitMoy>0.90)
            {
                Mut_Grad_U(IndividuTemp2);
                // Vérifier que les valeurs sont dans les min-max
                CheckVarMinMax(IndividuTemp)

                Set_U_P_Fit(IndividuTemp2);
                double Uapres=IndividuTemp2[2], Fitap=IndividuTemp2[1];

                if(Uavant>Uapres && (Fitav/2)<Fitap)
                {
                    for(j=1; j<14; j++)
                        enfants[i][j]=IndividuTemp2[j];
                }
                else
                {
                    for(j=1; j<14; j++)
                        enfants[i][j]=IndividuTemp[j];
                }
            }
            else
*/          {
                for(j=1; j<14; j++)
                    enfants[i][j]=IndividuTemp[j];
            }
        }
//************* Fin de l'Évaluation *************

//************* Début de l'Élimination et remplacement *************

        int PopSurv=2*m_iPop;
        Mat_IO_SP survivants(PopSurv,14); // Bassin d'Individus survivants potentiels

        for(i=0; i<m_iPop; i++)    // Remettre le fitness des parents à zéro
            m_Population[i][1]=0;
        SetPopFitness(m_Population); // Fitness des parents selon contrainte

        for(i=0; i<m_iPop; i++) // Transférer les Parents et les enfants
        {                       //     dans les survivants (potentiels)
            for(int j=0; j<14; j++)
            {
```

```
                survivants[i][j]=m_Population[i][j];
                survivants[i+m_iPop][j]=enfants[i][j];
            }
        }
        Set_minU_minTu(survivants); // Déterminer les mins sur fitmoyen des contraintes
        SetPopFitness(survivants); // Déterminer le fitness sur population globale

        int survivant=survivants[0][0];
        i=0;
/*      do
        {
            if (survivants[i][1]<1)
                survivant=survivants[i][0];
            i++;
        } while(survivant==survivants[0][0] && i<PopSurv);
*/
        // Choisir le survivant selon fitness
        for(i=0; i<m_iPop; i++)
        {
            double valeur=0;
            for(int k=0; k<PopSurv; k++)
            {
                if(survivants[k][1]>valeur) //  && survivants[k][1]<1)
                {
                    valeur=survivants[k][1];
                    survivant=survivants[k][0];
                }
            }
/*          if (valeur==0)
            {   // Si il y a trop d'individus avec un fitness=1 alors on les accepte
                for(int k=0; k<PopSurv; k++)
                {
                    if(survivants[k][1]>valeur)
                    {
                        valeur=survivants[k][1];
                        survivant=survivants[k][0];
                    }
                }
            }
*/

        // Trouver, transférer et annuler le survivant

            k=0;
            while(survivants[k][0]!=survivant && k<PopSurv)
                k++; // Placer k sur l'index du survivant
            for (int j=0; j<14; j++)
                m_Population[i][j]=survivants[k][j];
            survivants[k][1]=0, survivants[k][2]=M, survivants[k][3]=M;
            m_Population[i][1]=0;
            // Réinitialiser les fitness pour la prochaine évaluation
        }
        SetPopFitness(m_Population); // Selon les contraintes
        Set_minU_minTu(m_Population);
        SetPopFitness(m_Population); // Selon les contraintes et objectif
        Set_minU_minTu(m_Population);// Selon fitness complet

//************** Fin de l'Élimination et remplacement *************

        if(m_iGenPc>=(m_iNbGenPc-1) || m_iGen==m_iGenMax)
            FichierPop(m_Population), FichierResult();

    }
    else
    {
        if (m_iObjectifMin==0 || m_iObjectifMin==2)
        {
```

```
            m_bMinTuOk=false; // Pour calculer contrainte cible tu pour U
            RaffNum(3);
            m_bMinTuOk=true; // Pour calculer contrainte cible U pour tu
        }
        if (m_iObjectifMin==1 || m_iObjectifMin==2)
            RaffNum(2);
        CTime Temps = CTime::GetCurrentTime();
        TempsOpt=Temps-TempsDebut ;

        m_bTimerOn=false;
    }

} // FIN CMillOptDoc::Optimise()
//************* Fin de l'optimisation ******************

void CMillOptDoc::InduviduAlea(Vec_IO_SP &individu)
{
    double W=30, H=25, wfmax=m_dVarMinMax[1][9];
    double dfmax=m_dVarMinMax[1][7], Drmax=m_dVarMinMax[1][0];

    // Arranger les min-max pour les variables qui peuvent
    // avoir des relations (dépendances) dans leur bornes
    double Npwrmin= ceil((W-wfmax)/(Drmax-1));
    double Npdrmin= ceil((H-dfmax)/(Drmax));
    double Dr=((W-wfmax)/Npwrmin)+1;
    if (Dr<((H-dfmax)/Npdrmin))
        Dr=((H-dfmax)/Npdrmin);

    individu[1]=0;     // Fitness vierge à la création
    individu[4]=alea(0, m_dVarMinMax[0][0], m_dVarMinMax[1][0], 0); // Dr;
    individu[5]=alea(0, m_dVarMinMax[0][1], m_dVarMinMax[1][1], 0); // Df
    individu[7]=alea(0, m_dVarMinMax[0][2], m_dVarMinMax[1][2], 0); //vf
    individu[6]=alea(0, m_dVarMinMax[0][3], m_dVarMinMax[1][3], 0); //vr
    individu[8]=alea(0, m_dVarMinMax[0][4], m_dVarMinMax[1][4], 0); //fr
    individu[9]=alea(0, m_dVarMinMax[0][5], m_dVarMinMax[1][5], 0); //ff
    individu[10]=m_dVarMinMax[0][6];
           //Npdralea(1, m_dVarMinMax[0][6], (m_dVarMinMax[1][6]/2), 0); //Npdr
    individu[11]=dfmax; //alea(0, m_dVarMinMax[0][7], dfmax, 0);    //df
    individu[12]=m_dVarMinMax[0][8];
            //Npwralea(1, m_dVarMinMax[0][8], (m_dVarMinMax[1][8]/2), 0); //Npwr
    individu[13]=wfmax; //alea(0, m_dVarMinMax[0][9], wfmax, 0);    //wf

    CheckVarMinMax(individu);
    Set_U_P_Fit(individu);
} // FIN CMillOptDoc::InduviduAlea

void CMillOptDoc::SetDiscPopInit(int NbInc)
{   // Générer la population initiale en discrétisant les variables de décision
    //  IndividuTemp(14);
    int size=pow((float)NbInc, 10.0), index=m_iIndex;
    Mat_IO_SP Temp(size,14);
    Vec_IO_SP IndividuTemp(14);

    for (int a=0; a<NbInc; a++) //Dr
     for (int b=0; b<NbInc; b++) //Df
      for (int c=0; c<NbInc; c++) //vr
       for (int d=0; d<NbInc; d++) //vf
        for (int e=0; e<NbInc; e++) //fr
         for (int f=0; f<NbInc; f++) //ff
          for (int g=0; g<NbInc; g++) //Npr
           for (int h=0; h<NbInc; h++) //df
            for (int i=0; i<NbInc; i++) //Npw
             for (int j=0; j<NbInc; j++) //wf
             {  // Temp[index][x]=alpha*(max-min)/(NbInc-1)+min;
                Temp[index][0]=index;
                Temp[index][4]=a*(m_dVarMinMax[1][0]-m_dVarMinMax[0][0])/
                    (NbInc-1)+m_dVarMinMax[0][0];
```

```
                Temp[index][5]=b*(m_dVarMinMax[1][1]-m_dVarMinMax[0][1])/
                    (NbInc-1)+m_dVarMinMax[0][1];
                Temp[index][6]=c*(m_dVarMinMax[1][2]-m_dVarMinMax[0][2])/
                    (NbInc-1)+m_dVarMinMax[0][2];
                Temp[index][7]=d*(m_dVarMinMax[1][3]-m_dVarMinMax[0][3])/
                    (NbInc-1)+m_dVarMinMax[0][3];
                Temp[index][8]=e*(m_dVarMinMax[1][4]-m_dVarMinMax[0][4])/
                    (NbInc-1)+m_dVarMinMax[0][4]; //fr
                Temp[index][9]=f*(m_dVarMinMax[1][5]-m_dVarMinMax[0][5])/
                    (NbInc-1)+m_dVarMinMax[0][5]; //ff
                double Npr=g*(m_dVarMinMax[1][6]-m_dVarMinMax[0][6])/
                    (NbInc-1)+m_dVarMinMax[0][6];
                int entier=floor(Npr);
                Npr=Npr-entier;
                if (Npr>0.5)
                    Npr=++entier;
                else
                    Npr=entier;
                Temp[index][10]=Npr;
                Temp[index][11]=h*(m_dVarMinMax[1][7]-m_dVarMinMax[0][7])/
                    (NbInc-1)+m_dVarMinMax[0][7]; //df
                double Npw=i*(m_dVarMinMax[1][8]-m_dVarMinMax[0][8])/
                    (NbInc-1)+m_dVarMinMax[0][8];
                entier=floor(Npw);
                Npw=Npw-entier;
                if (Npw>0.5)
                    Npw=++entier;
                else
                    Npw=entier;
                Temp[index][12]=Npw;
                Temp[index][13]=j*(m_dVarMinMax[1][9]-m_dVarMinMax[0][9])/
                    (NbInc-1)+m_dVarMinMax[0][9];
                for(int k=0; k<14; k++)
                        IndividuTemp[k]=Temp[index][k];
                IndividuTemp[1]=0;
                Set_U_P_Fit(IndividuTemp);
                if(IndividuTemp[1]>.2)
                {
                    for(k=0; k<14; k++)
                        Temp[index][k]=IndividuTemp[k];
                    index++;
                }
            }
    for (int i=0; i<(m_iPop/2); i++)
    {
        int survivant=Temp[0][0];
        double maxfit=M;
        for (int j=0; j<(index-1); j++)
        {
            if (Temp[j][2]<maxfit)
                survivant=j, maxfit=Temp[j][2];
        }
        for (int k=0; k<14; k++)
                m_PopulInit[i][k]=Temp[survivant][k];
        Temp[survivant][2]=M;

    }
    m_iIndex=index;

} // FIN CMillOptDoc::SetDiscPopInit

void CMillOptDoc::Mut_Grad_U(Vec_IO_SP &individu)
{
    // Paramètres du problème
    double R=300, vt=36000, tL=0.5, Ts=30, NL=200,
                L=80, W=30, H=25, Ps=0.059157, nu=0.85;
    double alpha=5.0, beta=1.75, gamma=1.5, lambda=0.0, k=5.00e11,
```

```
            te=1, CA=55, CB=0.75, k0=4;
    double Zr=4, Zf=4, Cpr=90, Cpf=Cpr, NAr=7, NAf=4;

    // Transférer les variables de décision
    double Df=individu[5], vf=individu[7], ff=individu[9],
            df=individu[11], wf=individu[13];
    double Dr=individu[4], vr=individu[6], fr=individu[8],
            Npr=individu[10], Npwr=individu[12];
    double dr=(H-df)/Npr, wr=(W-wf)/Npwr;
    individu[10]=dr, individu[12]=wr;
    // Évaluer les expressions intermédiaires
    double vff=1000*Zf*ff*vf/(Pi*Df), vfr=1000*Zr*fr*vr/(Pi*Dr);
    double tcr=(L*W*H-((W-wf)*L*df)-H*L*wf)/(wr*dr*vfr), tcf=(L*W/(Df*vff));
    double t0r=k/pow(vr,alpha)/pow(fr,beta)/pow(dr,gamma)/pow(wr,lambda);
    double t0f=k/pow(vf,alpha)/pow(ff,beta)/pow(df,gamma)/pow(wf,lambda);
    double CHr=(Cpr+CA*NAr)/(t0r*(NAr+1)), CHf=(Cpf+CA*NAf)/(t0f*(NAf+1));

    double grad[14];

    grad[4]=k0*(.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/dr/Zr/fr/vr+.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/dr/Zr/fr/vr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda))+.
3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)/(NAr+1)*(L*W*H-
1.*(W-1.*wf)*L*df-1.*H*L*wf)/wr/dr/Zr/fr/vr;
    grad[5]=0;
    grad[6]=k0*(-.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/dr/Zr/fr/pow(vr,2.0)*Dr-.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/dr/Zr/fr/pow(vr,2.0)*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(w
r,lambda)+.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/dr/Zr/fr/pow(vr,2.0)*Dr*te/k*pow(vr,alpha)*alpha*pow(fr,beta)*pow(dr,gamma)
*pow(wr,lambda))+.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*alpha/pow(vr,2.0)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda
)/(NAr+1)*(L*W*H-1.*(W-1.*wf)*L*df-1.*H*L*wf)/wr/dr/Zr/fr*Dr-.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)/(NAr+1)*(L*W*H-
1.*(W-1.*wf)*L*df-1.*H*L*wf)/wr/dr/Zr/fr/pow(vr,2.0)*Dr;
    grad[7]=k0*(-.3141592654e-2*L*W/Zf/ff/pow(vf,2.0)-.3141592654e-
2*L*W/Zf/ff/pow(vf,2.0)*te/k*pow(vf,alpha)*pow(ff,beta)*pow(df,gamma)*pow(wf,lambda)+.314
1592654e-
2*L*W/Zf/ff/pow(vf,2.0)*te/k*pow(vf,alpha)*alpha*pow(ff,beta)*pow(df,gamma)*pow(wf,lambda
))+.3141592654e-
2*(Cpf+CA*NAf)/k*pow(vf,alpha)*alpha/pow(vf,2.0)*pow(ff,beta)*pow(df,gamma)*pow(wf,lambda
)/(NAf+1)*L*W/Zf/ff-.3141592654e-
2*(Cpf+CA*NAf)/k*pow(vf,alpha)*pow(ff,beta)*pow(df,gamma)*pow(wf,lambda)/(NAf+1)*L*W/Zf/f
f/pow(vf,2.0);
    grad[8]=k0*(-.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/dr/Zr/pow(fr,2.0)/vr*Dr-.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/dr/Zr/pow(fr,2.0)/vr*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(w
r,lambda)+.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/dr/Zr/pow(fr,2.0)/vr*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*beta*pow(dr,gamma)*
pow(wr,lambda))+.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*beta/pow(fr,2.0)*pow(dr,gamma)*pow(wr,lambda)
/(NAr+1)*(L*W*H-1.*(W-1.*wf)*L*df-1.*H*L*wf)/wr/dr/Zr/vr*Dr-.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)/(NAr+1)*(L*W*H-
1.*(W-1.*wf)*L*df-1.*H*L*wf)/wr/dr/Zr/pow(fr,2.0)/vr*Dr;
    grad[9]=k0*(-.3141592654e-2*L*W/Zf/pow(ff,2.0)/vf-.3141592654e-
2*L*W/Zf/pow(ff,2.0)/vf*te/k*pow(vf,alpha)*pow(ff,beta)*pow(df,gamma)*pow(wf,lambda)+.314
1592654e-
2*L*W/Zf/pow(ff,2.0)/vf*te/k*pow(vf,alpha)*pow(ff,beta)*beta*pow(df,gamma)*pow(wf,lambda)
)+.3141592654e-
2*(Cpf+CA*NAf)/k*pow(vf,alpha)*pow(ff,beta)*beta/pow(ff,2.0)*pow(df,gamma)*pow(wf,lambda)
/(NAf+1)*L*W/Zf/vf-.3141592654e-
2*(Cpf+CA*NAf)/k*pow(vf,alpha)*pow(ff,beta)*pow(df,gamma)*pow(wf,lambda)/(NAf+1)*L*W/Zf/p
ow(ff,2.0)/vf;
    grad[10]=k0*(-.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/pow(dr,2.0)/Zr/fr/vr*Dr-.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/pow(dr,2.0)/Zr/fr/vr*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(w
```

```
r,lambda)+.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/wr/pow(dr,2.0)/Zr/fr/vr*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*gamma
*pow(wr,lambda))+.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*gamma/pow(dr,2.0)*pow(wr,lambda
)/(NAr+1)*(L*W*H-1.*(W-1.*wf)*L*df-1.*H*L*wf)/wr/Zr/fr/vr*Dr-.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)/(NAr+1)*(L*W*H-
1.*(W-1.*wf)*L*df-1.*H*L*wf)/wr/pow(dr,2.0)/Zr/fr/vr*Dr;
    grad[11]=k0*(-.3141592654e-2*(W-1.*wf)*L/wr/dr/Zr/fr/vr*Dr-.3141592654e-2*(W-
1.*wf)*L/wr/dr/Zr/fr/vr*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)+.
3141592654e-
2*L*W/Zf/ff/vf*te/k*pow(vf,alpha)*pow(ff,beta)*pow(df,gamma)*gamma/df*pow(wf,lambda))-
.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)/(NAr+1)*(W-
1.*wf)*L/wr/dr/Zr/fr/vr*Dr+.3141592654e-
2*(Cpf+CA*NAf)/k*pow(vf,alpha)*pow(ff,beta)*pow(df,gamma)*gamma/df*pow(wf,lambda)/(NAf+1)
*L*W/Zf/ff/vf;
    grad[12]=k0*(-.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/pow(wr,2.0)/dr/Zr/fr/vr*Dr-.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/pow(wr,2.0)/dr/Zr/fr/vr*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(w
r,lambda)+.3141592654e-2*(L*W*H-1.*(W-1.*wf)*L*df-
1.*H*L*wf)/pow(wr,2.0)/dr/Zr/fr/vr*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(w
r,lambda)*lambda)+.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)*lambda/pow(wr,2.
0)/(NAr+1)*(L*W*H-1.*(W-1.*wf)*L*df-1.*H*L*wf)/dr/Zr/fr/vr*Dr-.3141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)/(NAr+1)*(L*W*H-
1.*(W-1.*wf)*L*df-1.*H*L*wf)/pow(wr,2.0)/dr/Zr/fr/vr*Dr;
    grad[13]=k0*(.3141592654e-2*(1.*L*df-1.*H*L)/wr/dr/Zr/fr/vr*Dr+.3141592654e-
2*(1.*L*df-
1.*H*L)/wr/dr/Zr/fr/vr*Dr*te/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)+.3
141592654e-
2*L*W/Zf/ff/vf*te/k*pow(vf,alpha)*pow(ff,beta)*pow(df,gamma)*pow(wf,lambda)*lambda/wf)+.3
141592654e-
2*(Cpr+CA*NAr)/k*pow(vr,alpha)*pow(fr,beta)*pow(dr,gamma)*pow(wr,lambda)/(NAr+1)*(1.*L*df
-1.*H*L)/wr/dr/Zr/fr/vr*Dr+.3141592654e-
2*(Cpf+CA*NAf)/k*pow(vf,alpha)*pow(ff,beta)*pow(df,gamma)*pow(wf,lambda)*lambda/wf/(NAf+1
)*L*W/Zf/ff/vf;

    for (int j=4; j<14; j++)
    {
        if(grad[j]>0)
            grad[j]=-1;
        else
            grad[j]=1;
    }

    double Amplimut=m_dMutPcGene/100, ampligrad=0;

    for (j=4; j<14; j++)
    {
        ampligrad=grad[j]*Amplimut;
        individu[j]=individu[j]*alea(0, 0, (1+Amplimut), 0);
    }

    individu[5]=Df;

    Npr=ceil((H-individu[11])/individu[10]), Npwr=ceil((W-individu[13])/individu[12]);
    if (individu[4]<individu[10])
        individu[4]=individu[10];
    if (individu[4]<individu[12])
        individu[4]=individu[12];

    individu[10]=Npr;
    individu[12]=Npwr;
} // FIN CMillOptDoc::Mut_Grad_U

void CMillOptDoc::Set_U_P_Fit(Vec_IO_SP &individu)
{
```

```
    // Paramètres du problème
    double  R=300,  vt=36000,  tL=0.5,  Ts=30,  NL=200,  L=80,  W=30,  H=25,  Ps=0.059157,
nu=0.85;
    double alpha=5.0, beta=1.75, gamma=1.5, lambda=0.0, k=5.00e11, te=1, CA=55, CB=0.75,
k0=4;
    double Zr=4, Zf=4, Cpr=90, Cpf=Cpr, NAr=7, NAf=4;

    // Transférer les variables de décision
    double     Df=individu[5],     vf=individu[7],     ff=individu[9],     df=individu[11],
wf=individu[13];
    double     Dr=individu[4],     vr=individu[6],     fr=individu[8],     Npr=individu[10],
Npwr=individu[12];
    double dr=(H-df)/Npr, wr=(W-wf)/Npwr;

    // Évaluer les expressions intermédiaires
    double vff=1000*Zf*ff*vf/(Pi*Df), vfr=1000*Zr*fr*vr/(Pi*Dr);
    double tcr=(L*W*H-((W-wf)*L*df)-H*L*wf)/(wr*dr*vfr), tcf=(L*W/(Df*vff));
    double expol=pow(wf,lambda);
    double t0r=k/pow(vr,alpha)/pow(fr,beta)/pow(dr,gamma)/pow(wr,lambda);
    double t0f=k/pow(vf,alpha)/pow(ff,beta)/pow(df,gamma)/pow(wf,lambda);
    double CHr=(Cpr+CA*NAr)/(t0r*(NAr+1)), CHf=(Cpf+CA*NAf)/(t0f*(NAf+1));

    // Évaluer les valeurs des fonctions objectifs
    double Tu=tcr+tcf+R/vt+tL+Ts/NL+tcr*te/t0r+tcf*te/t0f;
    double U=CB+k0*Tu+CHr*tcr+CHf*tcf;

    individu[2]=U;
    individu[3]=Tu;

    if(individu[1]==0) // Si le fitnes est vierge, il ne tient compte que des contraintes
        individu[1]=SetFitCont(individu);

} // FIN CMillOptDoc::Set_U_P_Fit

double CMillOptDoc::SetFitCont(Vec_IO_SP individu)
{
    // Paramètres du problème
    double  R=300,  vt=36000,  tL=0.5,  Ts=30,  NL=200,  L=80,  W=30,  H=25,  Ps=0.059157,
nu=0.85;
    double alpha=5.0, beta=1.75, gamma=1.5, lambda=0.0, k=5.00E+11, te=1, CA=55, CB=0.75,
k0=4;
    double Zr=4, Zf=4, Cpr=90, Cpf=Cpr, NAr=7, NAf=4;

    // Transférer les variables de décision
    double     Df=individu[5],     vf=individu[7],     ff=individu[9],     df=individu[11],
wf=individu[13];
    double     Dr=individu[4],     vr=individu[6],     fr=individu[8],     Npr=individu[10],
Npwr=individu[12];
    double dr=(H-df)/Npr, wr=(W-wf)/Npwr;
    double U=individu[2], P=individu[3];

    // Évaluer les expressions intermédiaires
    double vff=1000*Zf*ff*vf/(Pi*Df), vfr=1000*Zr*fr*vr/(Pi*Dr);

    double C1, C2, C3e, C3p, C4, C5, C6, C7, C8, C9, FitCont=1;
    double penalite1=1 ,penalite2=1, penalite3=1, penalite4=1, penalite5=1, penalite6=1;
    double    penalite7=1,    penalite8=1,    penalite9=1,    penalite10=1,    penalite11=1,
penalite12=1;

    // Contraintes puissances
    C1=Ps*dr*wr*vfr/(nu*1000); //62.48354411*dr*fr*vr/1000;
    C2=Ps*df*wf*vff/(nu*1000); //62.48354411*df*ff*vf/1000;

    // Contrainte fini Ra
    C3e=22.9468+45.68164596*ff*vf-5.683028067*pow(vf,2.0)/Df*ff+8.457882161*vf/Df*df;
    C3p=318*pow(ff,2.0)/(4*Df);
```

```
    // Contraintes Nmax
    C4=1000*vr/(Pi*Dr);
    C5=1000*vf/(Pi*Df);

    // Contraintes Feed mm/min
    C6=4*fr*((1000*vr)/(Pi*Dr));
    C7=4*ff*((1000*vf)/(Pi*Df));

    // Contrainte 5 (Force de coupe)
    // Équations à vérifier
    C8=60*C1/vr*1000;
    C9=60*C2/vf*1000;

    // Vérifier si les contraintes sont respectées et pénaliser sinon
    //  Puissance
    if (C1>m_dModelPmax)
        penalite1=m_dModelPmax/C1;
//      penalite1=sqrt(m_dModelPmax/C1);
    if (C2>m_dModelPmax)
        penalite2=m_dModelPmax/C2;
//      penalite2=sqrt(m_dModelPmax/C2);

    // Fini de surface
    if (C3e<0)
        penalite3=m_dModelRa/(m_dModelRa-C3e);
    if (C3e>m_dModelRa)
        penalite3=m_dModelRa/C3e;
    if (C3p>m_dModelRa)
        penalite3*=m_dModelRa/C3p;

    // RPM de la broche
    if (C4>m_iModelRPMmax)
        penalite4=m_iModelRPMmax/C4;
    if (C5>m_iModelRPMmax)
        penalite5=m_iModelRPMmax/C5;
    // Vitesse d'avance en mm/min
    if (C6>m_dFmax)
        penalite6=m_dFmax/C6;
    if (C7>m_dFmax)
        penalite7=m_dFmax/C7;

    // Force
    if (C8>m_dForcemax) // Vérifier ordre de grandeur, valeur et formule
        penalite8=m_dForcemax/C8; //Fcr
//      penalite8=sqrt(m_dForcemax/C8);   //Fcr
    if (C9>m_dForcemax) // Vérifier ordre de grandeur, valeur et formule
        penalite9=m_dForcemax/C9; //Fcf
//      penalite9=sqrt(m_dForcemax/C9);   //Fcf

    // Cible selon l'autre objectif
    if (m_iGen>0)
    {

        if (m_bMinTuOk==false && m_bMinTuPourU==TRUE)
    //      penalite11=pow((1-fabs(m_dCibleU-U)),2.0);
            penalite11=1-fabs(m_dCibleU-U);

        if (m_bMinTuOk==true && m_bMinUPourTu==TRUE)
            penalite11=1-fabs(m_dCibleTu-P);

        if (penalite11<m_dPoidsCont)
            penalite11=m_dPoidsCont;
    }

    if (dr>Dr) // Largeur et profondeur de passes respectent l<outil
        penalite12=Dr/dr;
    if (wr>(Dr-1))
```

```
        penalite12=penalite12*(Dr-1)/wr;


    if(m_iGenPc>=(m_iNbGenPc-1)) // COnditon bidon pour débugger
        m_iGen=m_iGen;
/*
    // Si la contrainte de force est active on pénalise aussi ceux qui sont en dessous
    if(((m_dContAct[0][1]>.95   &&   m_bMinTuOk==false)   ||   (m_dContAct[1][1]>.95   &&
m_bMinTuOk==true)) && penalite8==1)
        penalite8=1-(fabs(m_dForcemax-C8)/m_dForcemax);
    // Si la contrainte de puissance est active
    if(((m_dContAct[0][0]>.95   &&   m_bMinTuOk==false)   ||   (m_dContAct[1][0]>.95   &&
m_bMinTuOk==true)) && penalite1==1)
        penalite1=1-(fabs(m_dModelPmax-C1)/m_dModelPmax);
*/
    FitCont=penalite1*penalite2*penalite3*penalite4*penalite5*penalite6*penalite7*penalit
e8*penalite9*penalite10*penalite11*penalite12;

    if (FitCont>1)
        FitCont=1;
    if (FitCont<0)
        FitCont=0;

    return FitCont;
} // FIN CMillOptDoc::SetFitCont

void CMillOptDoc::SetPopFitness(Mat_IO_SP &Population)
{   // Déterminer le fitness selon les contraintes et U ou Tu
    Vec_IO_SP Individu(14);
    double FitCont=1, FitObj=1, Fitness=1;
    for(int i=0; i<m_iPop; i++)
    {
        for (int j=0; j<14; j++) //Transférer l'individu de la population
            Individu[j]=Population[i][j];
        FitCont=SetFitCont(Individu);
        if (Individu[1]!=0)
        {
            if(m_bMinTuOk==true && m_dMinU!=M)
            {
                FitObj=m_dMinU/Individu[2];
                if (FitObj<0)
                    FitObj=0;
                if (FitObj>1)
                    FitObj=1;
                Fitness=FitCont*(m_dPoidsCont)+FitObj*(1-m_dPoidsCont);
            }
            else if(m_dMinTu!=M)
            {
                FitObj=m_dMinTu/Individu[3];
                if (FitObj<0)
                    FitObj=0;
                if (FitObj>1)
                    FitObj=1;
                Fitness=FitCont*m_dPoidsCont+FitObj*(1-m_dPoidsCont);
            }
        }
        else
            Fitness=FitCont;

        if (Fitness<0)
            Fitness=0;
        Population[i][1]=Fitness;
    }
} // FIN CMillOptDoc::SetPopFitness

void CMillOptDoc::Set_minU_minTu(Mat_IO_SP Population)
{
```

```
    int taillePop=Population.nrows();
    // Déterminer le fitness moyen et max
    m_dMaxFit=0, m_dMinFit=M;
    for (int i=0; i<taillePop; i++)
    {
        m_dFitMoy+=Population[i][1];
        if (Population[i][1]>m_dMaxFit)
            m_dMaxFit=Population[i][1];
        if (Population[i][1]<m_dMinFit)
            m_dMinFit=Population[i][1];
    }
    m_dFitMoy/=(taillePop+1);
    m_dDeltaFit=m_dMaxFit-m_dMinFit;

    // Remonter la limite inférieure si on commence à converger
    if (m_iGenPc>=(.5*m_iNbGenPc) || m_dDeltaFit<(0.75*m_dPcAmel))
        SetLimitInf();

    if(m_bMinTuOk==false)
        if(m_dContAct[0][0]>1.5 || m_dContAct[0][1]>1.5)
           SetLimitInf();
    if(m_bMinTuOk==true)
        if(m_dContAct[1][0]>1.5 || m_dContAct[1][1]>1.5)
            SetLimitInf();

    // Identifier le meilleur individu Selon Fitness
    for( i=0; i<(taillePop); i++)
    {
        if (m_dMaxFit<=Population[i][1] && Population[i][1]<1)
        {
            if(m_bMinTuOk==true)
            {
                for (int j=0; j<14; j++)
                    m_dOptU[j]=Population[i][j];
            }
            else
            {
                for (int j=0; j<14; j++)
                    m_dOptTu[j]=Population[i][j];
            }
        }
        if (taillePop==m_iPop) //  Mise à jour des limites inférieures
        {
            if (m_dOptU[2]>=Population[i][2] && m_bMinTuOk==false
                    && Population[i][1]>.90)
            {   // ramasser une bonne solution pour U en passant
                for (int j=0; j<14; j++)
                    m_dOptU[j]=Population[i][j];
            }
        }
    }
    if (m_dOptU[1]>0 || m_dOptTu[1]>0)
        SetContAct();

} // FIN CMillOptDoc::Set_minU_minTu

void CMillOptDoc::CheckVarMinMax(Vec_IO_SP &individu)
{
    // Arranger les min-max pour les variables qui peuvent
    // avoir des relations (dépendances) dans leur bornes
    double W=30, H=25, wfmax=m_dVarMinMax[1][9];
    double dfmax=m_dVarMinMax[1][7], Drmax=m_dVarMinMax[1][0];
    double Npwrmin= ceil((W-wfmax)/(Drmax-1));
    double Npdrmin= ceil((H-dfmax)/(Drmax));
    if (individu[10]<Npdrmin)
        individu[10]=Npdrmin;
    if (individu[12]<Npwrmin)
```

```
        individu[12]=Npwrmin;

    for(int i=0; i<10; i++)
    {
      if (individu[(i+4)]<(m_dVarMinMax[0][i]+(m_dVarMinMax[1][i]*.01)))
          individu[(i+4)]=m_dVarMinMax[0][i];
      if (individu[(i+4)]>(m_dVarMinMax[1][i]*.99))
          individu[(i+4)]=m_dVarMinMax[1][i];
    }
} // FIN CMillOptDoc::CheckVarMinMax

void CMillOptDoc::SetContAct()
{
    // Déterminer les principales contraintes actives pour les optimums
//************************************************

    // Paramètres du problème
    double H=25, Zr=4, Ps=0.059157, W=30, nu=0.85;
    // Transfert des variables
    double Dr, vr, fr, Npr, Npwr, Df, vf, ff, df, wf;

    for(int i=0; i<2; i++)
    {
        Dr=0, Df=0;
        if (i==0 && m_dOptTu[1]>0)
        {// Transférer les variables de décision pour m_dOptTu
            Dr=m_dOptTu[4], vr=m_dOptTu[6], fr=m_dOptTu[8],
                Npr=m_dOptTu[10], Npwr=m_dOptTu[12];
            Df=m_dOptTu[5], vf=m_dOptTu[7], ff=m_dOptTu[9],
                df=m_dOptTu[11], wf=m_dOptTu[13];
            m_dContAct[i][3]=m_dOptTu[2]/m_dCibleU;
        }
        else if (i==1 && m_dOptU[1]>0)
        {// Transférer les variables de décision pour m_dOptU
            Dr=m_dOptU[4], vr=m_dOptU[6], fr=m_dOptU[8],
                Npr=m_dOptU[10], Npwr=m_dOptU[12];
            Df=m_dOptU[5], vf=m_dOptU[7], ff=m_dOptU[9], df=m_dOptU[11], wf=m_dOptU[13];
            m_dContAct[i][3]=m_dOptU[3]/m_dCibleTu;
        }
        if (Dr!=0 && Df!=0)
        {
            // Évaluer les expressions intermédiaires
            double dr=(H-df)/Npr, wr=(W-wf)/Npwr;
            double vfr=1000*Zr*fr*vr/(Pi*Dr);
            double Pmaxr, Rae, Fcr;

            // Puissance
            Pmaxr=Ps*dr*wr*vfr/(nu*1000);
            m_dContAct[i][0]=Pmaxr/m_dModelPmax;

            // Force de coupe
            Fcr=60*Pmaxr/vr*1000;
            m_dContAct[i][1]=Fcr/m_dForcemax; //Fcr

            // Fini de surface Rae
             Rae=22.9468+45.68164596*ff*vf-5.683028067*pow(vf,2.0)/Df*ff+
                8.457882161*vf/Df*df;
            m_dContAct[i][2]=Rae/m_dModelRa;

        }
    }

    if(m_iGenPc>10) // COnditon bidon pour débugger
        m_iGenPc=m_iGenPc;


//************************************************
```

```
} // FIN CMillOptDoc::SetContAct

void CMillOptDoc::SetLimitInf()
{
    if (m_bMinTuOk==true)
        m_dMinU=m_dOptU[2];
    else
        m_dMinTu=m_dOptTu[3];

    for(int i=0; i<m_iPop; i++)
    {
        if (m_dMinTu>m_Population[i][3] && m_bMinTuOk==false )
            m_dMinTu=m_Population[i][3];
        if (m_dMinU>m_Population[i][2] && m_bMinTuOk==true )
            m_dMinU=m_Population[i][2];
    }

    if (m_iGen==m_iGen)    // Bidon
        m_iGen=m_iGen;
} // FIN CMillOptDoc::SetLimitInf

void CMillOptDoc::RaffNum(int obj)
{   // obj: 3 pour Tu, 2 pour U
    Vec_IO_SP temp(14);

    double pas, bestsols[2][14], bestTu, bestU;
    int NbPasses=4, index=10000, i_pas=0;
    bool solbest=false;
    for (int i=0; i<14; i++) // Transférer les meilleures solutions
        bestsols[0][i]=m_dOptTu[i], bestsols[1][i]=m_dOptU[i], temp[i]=M;
    do
    {
    pas=.02, bestTu=bestsols[0][3], bestU=bestsols[1][2];
    i_pas=(m_dVarMinMax[1][6]-m_dVarMinMax[0][6])/2;
    if(((m_dVarMinMax[1][8]-m_dVarMinMax[0][8])/2)>i_pas)
        i_pas=((m_dVarMinMax[1][8]-m_dVarMinMax[0][8])/2);

    temp[0]=index, temp[1]=0;

    for (int p=0; p<NbPasses; p++)
    {
        solbest=false;
        for (int a=-1; a<2; a++) //Dr
        {
         temp[4]=bestsols[(3-obj)][4]*(1+a*pas);
         for (int b=-1; b<2; b++) //
         {
          temp[5]=bestsols[(3-obj)][5]*(1+b*pas);
          for (int c=-1; c<2; c++) //
          {
           temp[6]=bestsols[(3-obj)][6]*(1+c*pas);
           for (int d=-1; d<2; d++) //
           {
            temp[7]=bestsols[(3-obj)][7]*(1+d*pas);
            for (int e=-1; e<2; e++) //
            {
             temp[8]=bestsols[(3-obj)][8]*(1+e*pas);
             for (int f=-1; f<2; f++) //
             {
              temp[9]=bestsols[(3-obj)][9]*(1+f*pas);
              for (int g=-1; g<2; g++) //
              {
               temp[10]=bestsols[(3-obj)][10]+(g*i_pas);
               for (int h=-1; h<2; h++) //
               {
                temp[11]=bestsols[(3-obj)][11]*(1+h*pas);
                for (int i=-1; i<2; i++) //
```

```
                {
                 temp[12]=bestsols[(3-obj)][12]*(i*i_pas);
                 for (int j=-1; j<2; j++) //
                 {
                  temp[13]=bestsols[(3-obj)][13]*(1+j*pas);
                  temp[1]=0;
                  CheckVarMinMax(temp); // Vérifier les limites
                  Set_U_P_Fit(temp); // Le fitness ne tient compte que des contraintes
                  if(temp[obj]<bestsols[(3-obj)][obj])
                    if( (temp[1]>.99 && obj==3) || (temp[1]>.99 && obj==2) )
                    {
                      temp[0]=index++, solbest=true;
                      for (int k=0; k<14; k++) // Transférer les meilleures solutions
                        bestsols[3-obj][k]=temp[k];
                    }
                 } // fin j
                } // fin i
               } // fin h
              } // fin g
             } // fin f
            } // fin e
           } // fin d
          } // fin c
         } // fin b
        } // fin a
        if(solbest==false)
        {
          pas*=.20;
          if(i_pas>1)
            i_pas--;
        }
        else if(p>0)
          p--;
    } // fin p
    } while (bestTu!=bestsols[0][3] || bestU!=bestsols[1][2]);
    for (i=0; i<14; i++) // Transférer les meilleures solutions
        m_dOptTu[i]=bestsols[0][i], m_dOptU[i]=bestsols[1][i];

    SetContAct();

} // FIN CMillOptDoc::RaffNum(int obj)

//************* Routines utilitaires (Début)******************

DP NR::ran1(int &idum)
/*  "Minimal" random number generator of Park and Miller with Bays-Durham shuffle and
added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of the
endpoint values). Call with idum a negative integer to initialize; thereafter, do not
alter idum between successive deviates in a sequence. RNMX should approximate the largest
floating value that is less than 1.   */
{
    const int IA=16807,IM=2147483647,IQ=127773,IR=2836,NTAB=32;
    const int NDIV=(1+(IM-1)/NTAB);
    const DP EPS=3.0e-16,AM=1.0/IM,RNMX=(1.0-EPS);
    static int iy=0;
    static Vec_INT iv(NTAB);
    int j,k;
    DP temp;

    if (idum <= 0 || !iy)              //  Initialize.
        {
            if (-idum < 1)
                idum=1;                //  Be sure to prevent idum = 0.
            else
                idum = -idum;
            for (j=NTAB+7; j>=0; j--) //  Load the shuffle table (after 8 warm-ups).
            {
```

```
                k=idum/IQ;
                idum=IA*(idum-k*IQ)-IR*k;
                if (idum < 0)
                    idum += IM;
                if (j < NTAB)
                    iv[j] = idum;
            }
            iy=iv[0];
        }

    k=idum/IQ;                          //  Start here when not initializing.
    idum=IA*(idum-k*IQ)-IR*k;           //  Compute idum=(IA*idum) h IM withoutover
    if (idum < 0)
        idum += IM;                     //  flows by Schrage's method.
    j=iy/NDIV;                          //  Will be in the range 0..NTAB-1.
    iy=iv[j] ;                          //  Output previously stored value and refill the
    iv[j] = idum;                       //  shuffle table.
    if ((temp=AM*iy) > RNMX)
        return RNMX;                    //  Because  users  don't  expect  endpoint  values.
else return temp;
    else
        return temp;
} // FIN NR::ran1

CString CMillOptDoc::RemplPoint(CString str)
{   // Remplacer les points par des virgules dans une CString
    // Pour compatibilité avec le format métrique dans Excel
    int L=0;
    L=str.GetLength();
    char c;
    for (int i=0; i<L; i++)
    {
        c=str.GetAt(i);
        if(c==46)
        {
            c=44;
            str.SetAt(i, c);
        }
    }
    return str;
} // FIN RemplPoint

void CMillOptDoc::FichierLog()
{
    // Mise à jour du fichier Log
    int NbRun;
    char buffer[15];
    m_FdebugFile=fopen("debug.txt","rt");
    fgets( buffer, 15, m_FdebugFile );
    NbRun=atoi(buffer);
    fclose(m_FdebugFile);

    m_FdebugFile=fopen("debug.txt","w+t");
    _itoa(++NbRun, buffer, 10);
    fputs( buffer, m_FdebugFile );
    fclose(m_FdebugFile);
} // FIN CMillOptDoc::FichierLog

void CMillOptDoc::FichierResult()
{   // Afficher le résumé des meilleurs résultats
    CString s;
    char Temp[20];
    m_FdebugFile=fopen("résultats.csv","a+t");

    // Mettre à jour le temps d'optimisation
    int jrs=TempsOpt.GetDays();
    int hrs=TempsOpt.GetHours();
```

```
    int mins=TempsOpt.GetMinutes();
    int secs=TempsOpt.GetSeconds();
    m_iTemps=((jrs*24+hrs)*60+mins)*60+secs;

    // En-tête
    s="Génération:;";
    _gcvt( m_iGen, 4, Temp );
    s+=Temp;
    s+=";Temps:;";
    _gcvt( m_iTemps, 12, Temp );
    s+=Temp;
    s+="\n";
    s=RemplPoint(s);
    fputs( s, m_FdebugFile );

    // afficher le meilleur pour min U
    s=";Min U:;";
    for (int i=0; i<14; i++)
    {
        _gcvt( m_dOptU[i], 12, Temp );
        s+=Temp;
        s+=";";
    }
    s+="\n";
    s=RemplPoint(s);
    fputs( s, m_FdebugFile );

// afficher le meilleur pour min Tu
    s=";Min Tu:;";
    for (i=0; i<14; i++)
    {
        _gcvt( m_dOptTu[i], 12, Temp );
        s+=Temp;
        s+=";";
    }

    s+="\n\n";
    s=RemplPoint(s);
    fputs( s, m_FdebugFile );

    /*  s+=";Fit Moy:;";
    _gcvt( m_dFitMoy, 12, Temp );
    s+=Temp;

    s+=";Gen conv:;";
    _gcvt( m_iGenPc, 12, Temp );
    s+=Temp;
*/

    fclose(m_FdebugFile);
} // FIN CMillOptDoc::FichierResult

void CMillOptDoc::FichierPop(Mat_IO_SP Population)
{   // Écrire la population dans un fichier
    FILE *FichierPop;
    CString s;
    char Temp[24];
    int a=m_iPop;
    int b=m_Population.ncols();
    double TempMinU=m_dMinU,TempMinTu=m_dOptTu[3], TempMaxFit=m_dFitMoy;

    FichierPop=fopen("population.csv","a+t");

    s="Génération:";
    s+=";";
    _gcvt( m_iGen, 5, Temp );
    s+=Temp;
```

```
    s+="Min U:";
    s+=";";
    _gcvt( TempMinU, 8, Temp );
    s+=Temp;
    s+=";";
    s+="Min Tu:";
    s+=";";
    _gcvt( TempMinTu, 8, Temp );
    s+=Temp;
    s+=";";
    s+="Fit Moy:";
    s+=";";
    _gcvt( TempMaxFit, 8, Temp );
    s+=Temp;
    s+="\n";
    fputs( s, m_FdebugFile );
    s.Empty();

    for(int i=0; i<a; i++)
    {
        for(int j=0; j<b; j++)
        {
            _gcvt( Population[i][j], 16, Temp );
            s+=Temp;
            s+=";";
        }
        s+="\n";
        s=RemplPoint(s);
        fputs( s, m_FdebugFile );
        s.Empty();
    }
    s="\n";
    fputs( s, m_FdebugFile );
    fclose(FichierPop);
} // FIN CMillOptDoc::FichierPop

double CMillOptDoc::alea(int type, double min, double max, double param1)
{   // Pilote de génération aléatoire
    double nalea=0, plage=max-min;

    if(type==0) // Type 0, réel de distribution uniforme entre min et max
        nalea=NR::ran1(idum)*plage+min;
    if(type==1) // Type 1, entier de distribution uniforme entre min et max
        nalea=arrondi(NR::ran1(idum)*plage+min);
    return nalea;
} // FIN CMillOptDoc::alea

int CMillOptDoc::arrondi(double reel)
{
    int entier=floor(reel);
    reel-=entier;
    if (reel>=0.5)
        entier++;
    return entier;
} // FIN CMillOptDoc::arrondi

//************* Routines utilitaires (Fin)******************
```

BIBLIOGRAPHIE

Adlemo, A., Andreasson, S. A. (1996). Balanced Automation in Flexible Manufacturing System. *Studies in informatics and control,* 5(2).

Agapiou, J.S. (1992). Optimization of machining operations based on a combined criterion, part 1. The use of combined objectives in single-pass operations. *Journal of Engineering for Industry, Transactions of the ASME,* 114(4), 500-507.

Agapiou, J.S. (1992). Optimization of machining operations based on a combined criterion, part 2. Multipass operations. . *Journal of Engineering for Industry, Transactions of the ASME,* 114(4), 508-513.

Akella, R., Kumar, P. R. (1986). Optimal Control of Production Rate in a Failure Prone Manufacturing System. *IEEE Transactions on Automatic Control,* 31(2), 116-126.

Al-Ahmari, A.M.A. (2001). Mathematical Model for Determining Machining Parameters in Multipass Turning Operations with Constraints. *International Journal of Production Research,* 39(15), 3367-3376.

Al-Ahmari, A.M.A. (2002). Computer aided Optimization of Scheduling and Machining Parameters in Job Shop Manufacturing Systems. *Production planning & control,* 13(4), 401-406.

Creese, Robert C. (1999). *Introduction to Manufacturing Processes and Materials.* New York : Marcel Dekker.

Darwin, Charles. (1859) *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* London : J. Murray.

Ermer, D. S. (1997). A Century of Optimizing Machining Operations. *Journal of Manufacturing Science and Engineering,* 119, 817-822.

Ermer, D. S., Araj, S. N. (1983). Optimum Selection of Machining Conditions for Peripheral Milling with Practical Constraints. *Manufacturing Engineering Transactions,* 453-460

Gilbert, W. W. (1950). *The Economics of Machining, Machining Theory and Practice.* Cleveland : ASM, H. Ernst ed.

Groover, Mikell, P. (2002). *Fundamentals of Modern Manufacturing.* (2e éd.). New York : John Wiley & Sons.

Kenne, J. P., & Gharbi, A. (2004). Stohastic Optimal Production Control Problem with Corrective Maintenance. *Computers and Industrial Engineering*, 46(4), 865-875.

Lou, M. S., Chen, J. C., Caleb, M. L. (1999). Surface Roughness Prediction Technique For CNC End-Milling. *Journal of Industrial Technology,* 15(1).

Machinability Data Center. (1980). *Machining Data Handbook.* vol. 2. (3e éd.). Cincinnati, Ohio : Metcut Research Associates.

Malakooti, B., Deviprasad, J. (1989). An Interactive Multiple Criteria Approach for Parameter Selection in Metal Cutting. *Operations Research*, 37(5), 805-818.

Munoz-Escalona, P., Cassier, Z. (1998). Influence of the Critical Cutting Speed on the Surface Finish of Turned Steel. *Wear*, 218(1), 103-109.

Parent, L., Songmene, V., Kenné J. P. (2007). A generalized model for optimizing end milling operation. *Production Planning & Control,* (Article ID : TPPC 229154, à paraître en 2007)

Press, William H. (2002). *Numerical recipes in C++ : the art of scientific computing.* (2e éd.). Cambridge, Angleterre : Cambridge University Press.

Taylor, F.W. (1907). On the First Art of Cutting Metals, *Transaction of ASME*, 28, 31.
Tolouei-Rad, M.; Bidhendi, I.M. (1997). On the Optimization of Machining Parameters for Milling Operations. *International Journal of Machine Tools & Manufacture*, 37(1), 1-16

White, B., Houshyar, A. (1992). Quality and Optimum Parameter Selection in Metal Cutting. *Computers in industry,* 20(1), 87-98.

Wu, S. M., Ermer, D. S. (1966). Maximum Profit Rate as the Criterion in the Determination of the Optimum Cutting Conditions. *ASME Journal of Engineering for Industry*, 88, 435-442.