ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Projet de Semestre

Optimisation de structures métalliques

Département de mathématiques
Chaire de Recherche Opérationnelle
Professeur Th. M. Liebling

Paroz Sandrine
Mathématiques - 7ème semestre

Responsables :
J.-F. Hêche, L. Pournin

Hiver 2000-2001

Table des matières

Remarque préliminaire

Dans le cadre de ce projet de semestre, j'ai dû réaliser une présentation HTML du problème. Ce qui se trouve sur cette page web correspond aux chapitres 1 et 2, ainsi qu'à la section 5.2 de ce présent rapport.

Chapitre 1

Introduction

L'optimisation structurelle permet d'obtenir les structures les plus performantes que ce soit un pont, un toît ou une navette spatiale. Elle se trouve au carrefour de plusieurs disciplines : les mathématiques, la physique et l'ingéniérie civile. Dans notre cas particulier, nous allons nous intéresser à la conception et à la stabilité de treillis articulés.

Ces treillis sont des constructions formées de barres articulées dont les extrémités sont reliées entre elles par des noeuds. Pour éviter la formation de couples, il faut que les axes des membrures se coupent en un seul point. De plus, il faut que les forces extérieures (appliquées à cette structure) soient appliquées uniquement aux noeuds, car les barres des treillis supportent très mal les charges latérales.

Nous allons étudier l'aspect mathématique du problème, plus précisément, nous allons utiliser la recherche opérationnelle afin de décider quelle est la structure de poids minimum supportant une charge extérieure donnée.
Pour faire cela, nous introduisons un critère d'optimisation : le poids de la structure que nous allons chercher à minimiser. Ce critère nous permettra de choisir parmi toutes les barres possibles, celles qui constitueront notre structure.
Pour simplifier la modélisation, nous supposons que :

1. Le poids d'une barre est directement proportionnel à son volume.
2. La densité du matériau est la même pour toutes les barres.
3. La section d'une barre est proportionnelle à l'effort dans cette barre.
4. La constante de proportionalité entre la section et l'effort d'une barre est la même pour toutes les barres et ne dépend pas du fait que celles-ci soient en compression ou en traction. On fixe cette constante égale à 1.

Avec les deux premières hypothèses, nous remarquons que minimiser le poids de la structure revient à minimiser son volume.

Chapitre 2

Théorie

2.1 Conditions d'équilibre

On considère les treillis de la façon suivante : il n'y a pas de couples extérieurs appliqués aux noeuds et les membrures ne sont soumises qu'à des efforts axiaux. Ceci fait que chaque barre n'est soumise qu'à deux forces extérieures, une à chaque extrémité. Ces deux forces sont de même direction, de même module mais de sens opposé. Chaque barre exerce aussi des forces sur les deux noeuds à ses extrémités : ce sont des forces de même module mais de sens opposé.
Si on veut qu'une structure soit en équilibre, il faut que les forces exercées sur chaque noeud s'équilibrent. Ces forces sont d'une part, les réactions des membrures incidentes aux noeuds et d'autre part, les charges ou réactions extérieures. On peut alors définir un système d'équations d'équilibre quasi-statique de la structure.

2.2 Matrice d'incidence

Chaque sommet i de la structure se trouve à une position déterminée par le vecteur $\mathbf{p_i}$. Si j est un sommet adjacent à i alors :

$$\mathbf{u_{ij}} = \frac{\mathbf{p_j} - \mathbf{p_i}}{\|\mathbf{p_j} - \mathbf{p_i}\|}$$

est le vecteur unitaire le long de l'arête $\{i, j\}$. On peut remarquer que $\mathbf{u_{ji}} = -\mathbf{u_{ij}}$

On note :
x_{ij} : l'effort qu'exerce une barre $\{i, j\}$ sur ses extrémités (attention $x_{ij} = x_{ji}$).
$\mathbf{b_i}$: le vecteur des forces extérieures appliquées au sommet i.

On peut alors écrire l'équilibre des forces en chaque noeud de la manière suivante :

$$\sum_{j:\{i,j\}\in E} \mathbf{u_{ij}} x_{ij} = -\mathbf{b_i} \qquad \forall i \in V \tag{2.1}$$

où Vest l'ensemble des sommets (noeuds) et E l'ensemble des arêtes (barres) du graphe associé à la structure.
Sous forme matricielle, ce système s'écrit de la manière suivante :

$$\mathbf{Ax} = -\mathbf{b} \tag{2.2}$$

où :
x : est le vecteur des efforts x_{ij} dans chaque barre
b : est le vecteur des charges extérieures b_i appliquées aux noeuds de la structure
A : est la matrice contenant les vecteurs unitaires $\mathbf{u_{ij}}$.

Voici un exemple de structure :

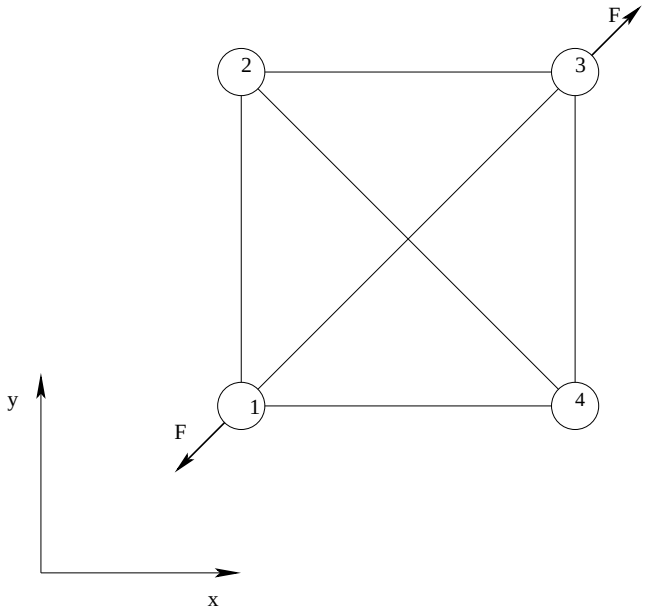

FIG. 2.1 – *Exemple de treillis soumis à deux forces extérieures de même module* **F**

Cette structure nous donne le système d'équations suivant :

$$\begin{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & & & \\ \begin{pmatrix} 0 \\ -1 \end{pmatrix} & & & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{pmatrix} & \\ & \begin{pmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{pmatrix} & & \begin{pmatrix} -1 \\ 0 \end{pmatrix} & & \begin{pmatrix} 0 \\ -1 \end{pmatrix} \\ & & \begin{pmatrix} -1 \\ 0 \end{pmatrix} & & \begin{pmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} \begin{pmatrix} x_{12} \\ x_{13} \\ x_{14} \\ x_{23} \\ x_{24} \\ x_{34} \end{pmatrix} = \begin{pmatrix} \frac{F}{\sqrt{2}} \\ \frac{F}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{-F}{\sqrt{2}} \\ \frac{-F}{\sqrt{2}} \\ 0 \\ 0 \end{pmatrix}$$

Dans la matrice **A** (8x6) ci-dessus, les éléments manquants sont nuls. Les parenthèses autour des vecteurs unitaires ne sont là que pour la compréhension du système d'équations. Cela permet de montrer que dans (2.1) il faut tenir compte des équations sur **x** et sur **y**. Pour résoudre le système il ne faut pas tenir compte de ces parenthèses.

Nous pouvons remarquer que si l'on considère la matrice **A** comme une matrice de vecteurs à deux dimensions, celle-ci n'a que deux entrées non nulles par colonne et la somme de ces deux entrées est toujours nulle (les deux entrées sont des vecteurs unitaires de sens opposé). On peut comparer **A** avec la matrice d'incidence "sommets-arcs" d'un réseau (les lignes de la matrice correspondent aux sommets et les colonnes correspondent aux arêtes). Dans une matrice "sommets-arcs" chaque colonne a deux entrées non nulles : l'une vaut -1 et l'autre $+1$.

Par analogie à ces matrices d'incidence "sommets-arcs", on appelle **A** la *matrice d'incidence*.

2.3 Ancrage des structures

Les ouvrages que nous étudions sont ancrés au sol. Or le système (2.2) ne s'applique efficacement qu'aux structures pouvant se déplacer librement (dans le plan ou dans l'espace). Pour résoudre ce problème, il suffit d'éliminer de (2.2) les équations pour les points d'ancrage. En effet, grâce au "principe d'action-réaction", la force exercée par le sol (la Terre) équilibrera le bilan des forces à ces points d'ancrage.

2.4 Définition du programme linéaire

2.4.1 Programme déjà existant

Avec ce que nous avons vu précédemment, notons :

$l_{ij} = \|p_j - p_i\|$: la longueur de la barre $\{i, j\}$.
x_{ij} : l'effort de la barre $\{i, j\}$.
V : l'ensemble des noeuds de la structure.
E : les barres potentielles de la structure.
$\mathbf{b_i}$: la charge extérieure appliquée au noeud i.

La fonction objectif de notre problème est égale à la somme, sur toutes les barres possibles, de leur volume $l_{ij}|x_{ij}|$.
Nous pouvons donc écrire notre problème d'optimisation de la façon suivante :

$$\begin{aligned} \text{Minimiser} \quad & z = \sum_{\{i,j\}\in E} l_{ij}|x_{ij}| \\ \text{s.c.} \quad & \sum_{j:\{i,j\}\in E} \mathbf{u_{ij}} x_{ij} = -\mathbf{b_i} \quad \forall i \in V \\ & x_{ij} \in \mathbb{R} \qquad\qquad\qquad \forall \{i, j\} \in E \end{aligned} \tag{2.3}$$

La formulation (2.3) ne définit pas un programme linéaire ; en effet, on a dans la fonction objectif la valeur absolue de chaque variable. Pour obtenir un programme linéaire, on remplace pour chaque $\{i, j\}$ la variable x_{ij} par la différence de deux variables non négatives :

$$x_{ij} = x_{ij}^+ - x_{ij}^-, \qquad x_{ij}^+, x_{ij}^- \geq 0$$

où :

x_{ij}^+ représente la force de traction exercée sur la barre $\{i, j\}$.
x_{ij}^- représente la force de compression exercée sur la barre $\{i, j\}$.

Avec cette notation et pour autant qu'au plus une des deux variables x_{ij}^+ ou x_{ij}^- soit non nulle, alors nous avons :

$$|x_{ij}| = x_{ij}^+ + x_{ij}^-$$

Finalement, nous pouvons écrire le programme linéaire suivant :

$$\begin{aligned} &\text{Minimiser} \quad z = \sum_{\{i,j\}\in E} l_{ij}(x_{ij}^{+} + x_{ij}^{-}) \\ &\text{s.c} \qquad \sum_{j:\{i,j\}\in E} \mathbf{u_{ij}} \left(x_{ij}^{+} - x_{ij}^{-}\right) = -\mathbf{b_i} \quad \forall i \in V \\ &\qquad x_{ij}^{+}, x_{ij}^{-} \geq 0 \qquad\qquad \forall \{i,j\} \in E \end{aligned} \tag{2.4}$$

2.4.2 Amélioration du programme linéaire

Le principal désavantage de ce problème d'optimisation est le fait qu'il ne tient pas compte du poids propre de la barre.

Par conséquent, pour obtenir un modèle plus proche de la réalité, il est important d'inclure le poids propre de la barre, ce qui nous donne le problème suivant :

$$\begin{aligned} &\text{Minimiser} \quad z = \sum_{\{i,j\}\in E} l_{ij}|x_{ij}| \\ &\text{s.c.} \qquad \sum_{j:\{i,j\}\in E} \mathbf{u_{ij}} x_{ij} = -\mathbf{b_i} - \sum_{j:\{i,j\}\in E} \frac{l_{ij}}{2}|x_{ij}|\mathbf{g} \quad \forall i \in V \\ &\qquad x_{ij} \in \mathbb{R} \qquad\qquad \forall \{i,j\} \in E \end{aligned} \tag{2.5}$$

où :

g est le vecteur d'accélération de pesanteur

$\frac{l_{ij}}{2}|x_{ij}|\mathbf{g}$ est la moitié du poids propre de la barre $\{i,j\}$. En effet, au lieu d'appliquer le poids d'une barre à son centre de gravité, nous appliquons, pour simplifier, la moitié de son poids propre à chacune de ses deux extrémités.

A nouveau, la formulation (2.5) ne définit pas un programme linéaire, on la remplace donc avec :

$$x_{ij} = x_{ij}^{+} - x_{ij}^{-}, \qquad x_{ij}^{+}, x_{ij}^{-} \geq 0$$

et

$$|x_{ij}| = x_{ij}^{+} + x_{ij}^{-}$$

Finalement, nous pouvons écrire le programme linéaire suivant :

$$\begin{aligned}
&\text{Minimiser} \quad z = \sum_{\{i,j\} \in E} l_{ij}(x_{ij}^{+} + x_{ij}^{-}) \\
&\text{s.c} \qquad \sum_{j:\{i,j\} \in E} \left[(\mathbf{u_{ij}} + \frac{l_{ij}}{2}\mathbf{g})x_{ij}^{+} + (-\mathbf{u_{ij}} + \frac{l_{ij}}{2}\mathbf{g})x_{ij}^{-} \right] = -\mathbf{b_i} \quad \forall i \in V \\
&\qquad\quad x_{ij}^{+}, x_{ij}^{-} \geq 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \forall \{i,j\} \in E
\end{aligned} \tag{2.6}$$

Nous pouvons décomposer les contraintes en les projetant sur $\mathbf{x}$ et $\mathbf{y}$. Soit les notations suivantes :

u_{ij}^{x} : la position de $\mathbf{u_{ij}}$ sur $\mathbf{x}$.
u_{ij}^{y} : la position de $\mathbf{u_{ij}}$ sur $\mathbf{y}$.
b_i^{x} : la composante des forces extérieures (au noeud i) sur $\mathbf{x}$.
b_i^{y} : la composante des forces extérieures (au noeud i) sur $\mathbf{y}$.
g : la norme de l'accélération de pesanteur : $g = \|\mathbf{g}\|$

On obtient le PL suivant : (en sachant que le système d'axes de Java est orienté négativement, i.e. **x** est dirigé vers la droite et **y** est dirigé vers le bas) :

$$
\begin{aligned}
\text{Minimiser} \quad & z = \sum_{\{i,j\}\in E} l_{ij}(x_{ij}^{+} + x_{ij}^{-}) \\
\text{s.c} \quad & \sum_{j:\{i,j\}\in E} \left[(u_{ij}^{x} x_{ij}^{+} - u_{ij}^{x} x_{ij}^{-}\right] = -b_i^{x} && \forall i \in V \\
& \sum_{j:\{i,j\}\in E} \left[(u_{ij}^{y} + g\frac{l_{ij}}{2})x_{ij}^{+} + (-u_{ij}^{y} + g\frac{l_{ij}}{2})x_{ij}^{-}\right] = -b_i^{y} && \forall i \in V \\
& x_{ij}^{+}, x_{ij}^{-} \geq 0 && \forall \{i,j\} \in E
\end{aligned}
\tag{2.7}
$$

Comme nous allons le voir plus tard, cette modélisation pose problème, car l'optimisation avec ce modèle (2.7) n'aboutit presque jamais à une solution.

Dans l'applet Java qui a été élaborée, on appelle *voisinage* l'ensemble des barres possibles depuis un point. La taille k du voisinage est le nombre de couches "autorisées" à partir d'un point. Dans les figures suivantes on peut voir les barres potentielles pour un voisinage de taille $k = 1$ (figure 2.2, page 12) et un voisinage de taille $k = 3$ (figure 2.3, page 12) . Il faut prendre en compte que les points sortant du quadrillage retenu ne sont pas valables.

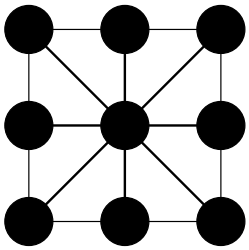

FIG. 2.2 – *Voisinage de taille 1.*

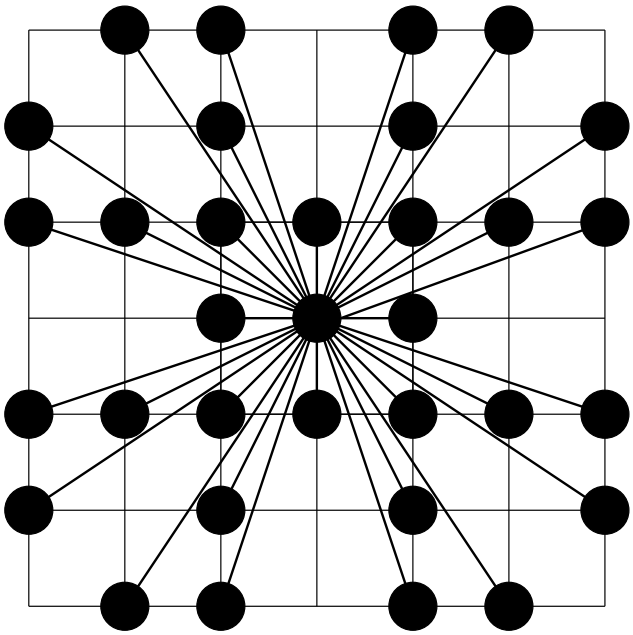

FIG. 2.3 – *Voisinage de taille 3.*

Chapitre 3

Développement

3.1 Première étape

Tout d'abord, je me suis familiarisée avec le langage Java (avec [4], [2] et [3]) que je ne connaissait pas auparavant puis j'ai étudié le programme TCL déjà existant. Pour mes débuts avec Java, j'ai testé le programme **HelloWorld.java** que l'on trouve dans presque tous les livres traitant de Java ainsi que sur de nombreuses pages webs. Ce petit programme existe en version application et en version applet. Après avoir compris la différence entre ces deux versions, j'ai pu commencer la programmation mon applet. Jean-François Hêche m'a donné le code à insérer dans un fichier **.html** permettant de visualiser les applets autant bien sur Netscape que sur Explorer.
Lorsque je programmais, j'ai utilisé **appletviewer** pour visualiser la compilation de mon code (sauf à la fin parce que j'avais besoin d'être sur une page web). En effet, avec un navigateur Internet, il faut, à chaque changement effectué dans le programme, fermer et rouvrir le navigateur pour être sûr que celui-ci prenne en compte les modifications.

3.2 Développement de la structure du programme

La première étape dans le développement du programme a été celle de créer le dialogue **NewDialog.java**, permettant de fixer le nombre de points horizontaux et verticaux ainsi que la taille du voisinage. En même temps, il s'agissait de développer la classe principale du programme : **OptStruct.java**. Cette classe permet d'initialiser l'applet et contient le menu principal de cette dernière. De plus, elle fait, en quelque sorte, le lien entre les différentes classes car certaines ont des constructeurs qui ont comme paramètre un objet de type `Optstruct`.

Pour créer les différents dialogues présents dans ce programme, j'ai utilisé la classe `javax.swing`. En effet, en Java, il est possible de faire cela aussi bien avec la classe `java.awt` qu'avec `javax.swing`. La différence entre ces deux classes est que `javax.swing` devrait tourner de la même façon sur toutes les plateformes ; de plus, Swing est censé améliorer l'aspect, le fonctionnement et le contrôle des interfaces par l'utilisateur.

L'étape suivante a été celle de créer le panel **PanelGraphique.java**, où est dessiné le canevas, les points d'ancrages, les forces et la solution. Au début, j'avais créé la grille avec des rayons fixés. Ainsi, si le canevas était composé de trente points horizontaux et vingt verticaux ou deux horizontaux et deux verticaux, le rayon ne changeait pas. De plus, la force était représentée par un trait fin trop peu visible. J'ai ensuite changé cela, en faisant en sorte que le rayon et l'espacement des points du canevas diffèrent selon le nombre de points dans le canevas. Le changement concernant la représentation de la force s'est fait plus tard. A la fin, Jean-François Hêche a nettement amélioré tous les graphismes se trouvant dans ce programme, notamment le maillage de départ.

En parallèle à la construction du canevas, j'ai commencé l'élaboration de la classe **Structure.java** et j'ai créé la classe **Force.java**. En effet, une force est définie par quatre composantes : sa coordonnée en **x**, sa coordonnée en **y**, sa valeur en **x** et sa valeur en **y**. Or il n'existe pas d'objet Java de telle sorte. La classe créée prend comme paramètres deux points : l'un pour les coordonnées et l'autre pour les composantes de la force.

J'ai passé un certain temps pour comprendre le principe de l'utilisation des classes, notamment le fait qu'il faille les écrire dans des fichiers **.java** séparés. En effet, au début, j'avais mis les différentes classes dans un seul fichier. Il n'y avait pas d'erreur à la compilation mais de nombreuses erreurs à l'exécution.

3.3 Fonction objectif et contraintes

Avant de développer ces méthodes, il s'est agi d'améliorer le modèle théorique. En effet, le programme **TCL** ne tenait pas compte du poids propre de la barre. Après discussion avec Lionel Pournin, le modèle (2.5) de la théorie a été établi. Comme indiqué précédemment, celui-ci tient compte de la moitié du poids propre appliqué à chaque extrémité d'une barre. En revanche, pour pouvoir tester mon programme au début (comparer les solutions avec celles de **TCL**) et pour pouvoir laisser le choix à l'utilisateur de tenir compte du poids de la barre, j'ai créé une boîte de dialogue permettant de choisir la valeur de l'accélération de pesanteur ; une valeur nulle de g correspondant à ne pas tenir compte du poids propre de la barre. Ce dialogue a, par la suite, été abandonné au profit de radio-boutons dans la barre de menu de l'applet.
Ensuite, le développement des méthodes permettant d'écrire la fonction objectif et les contraintes m'a pris beaucoup de temps. En effet, il fallait d'abord trouver les algorithmes permettant de les créer et de les implémenter de façon rigoureuse et efficace.
Une fois ces méthodes écrites, j'ai commencé d'étudier **lp_solve** version Java. Effectivement, nous avions décidé d'utiliser ce programme pour résoudre le PL car celui-ci est écrit en Java et se trouve sous la forme d'un package. On peut donc l'inclure dans son propre programme en faisant une importation, (de plus, s'il avait été efficace, on n'aurait pas eu besoin de limiter l'utilisation de l'applet à un certain groupe de personnes, au contraire de CPLEX qui est sous licence). Ce programme, à la base, est un programme écrit en **C**, très

mal commenté, mais pour lequel il existe une documentation extérieure pour la version **C** (voir page web [8]). Celle-ci est proche de la version **Java** dans le sens que le nom des méthodes existant en **Java** est le même que celles existant en **C** (il y a par contre plus de méthodes pour la version **C**.

Cette phase m'a aussi permis d'écrire les petits dialogues indiquant s'il y a une solution ou non.

3.4 Premiers essais

Lors des premiers tests de mon programme (avec **lp_solve** version Java), il s'est avéré qu'il fonctionnait pour une grille 2x2 (i.e. deux points horizontaux et deux points verticaux), 2x3 et 3x2, mais qu'à partir d'une grille 3x3 il ne trouvait pas la solution. Pour savoir si le problème venait de mon programme ou de **lp_solve**, j'ai d'abord contrôlé s'il y avait le bon nombre de variables dans le programme que je demandais à **lp_solve** d'optimiser. Après vérification, j'ai modifié et employé la méthode `void write_LP(lprec lp, PrintStream output)` de **solve.java** de telle sorte que lorsqu'on optimise une structure, un fichier **prgjava.lp** soit créé dans lequel se trouve le programme linéaire que **lp_solve** essaie de résoudre. Ce fichier a ensuite été employé avec **CPLEX** afin de savoir si celui-ci trouvait une solution. J'ai testé plusieurs dispositions et quand **lp_solve** ne trouvait pas de solution je testais le PL avec **CPLEX**, qui trouvait par contre une solution. Ainsi, on a pu en conclure qu'il y avait certainement un problème de conditionnement dans la version Java de **lp_solve**. Jean-François Hêche m'a suggéré d'employer la méthode `void auto_scale(lprec lp)` de **solve.java** pour essayer de résoudre ce problème. Ceci n'a pas amélioré la résolution des PL. Jean-François Hêche m'a alors proposé de tester différentes structures que la version Java de **lp_solve** n'arrivait pas à résoudre avec la version C pour voir où se situe le problème. Cette version trouve des solutions pour des petits et moyens systèmes. En revanche, pour certains grands PL (avec beaucoup de variables), il y a des problèmes numériques empêchant la résolution. Nous avons décidé d'abandonner totalement la version Java de **lp_solve**, puisque la version C fonctionnait plutôt bien.

3.5 Utilisation de sockets

Après toutes ces vérifications, nous avons choisi de faire une version *duale*, c'est-à-dire de laisser le choix à l'utilisateur d'optimiser avec **lp_solve version C** ou avec **CPLEX**, tout en sachant que seuls les utilisateurs provenant de l'EPFL pourront utiliser **CPLEX**, parce que celui-ci est un programme sous licence, et en ayant conscience que **lp_solve** a des problèmes numériques pour des grands systèmes. De cette façon on pourra quand même mettre l'applet sur le web.
Cette façon de faire complique le programme. En effet, quand un utilisateur veut optimiser, il faut ouvrir un socket qui fait la connexion avec le serveur se trouvant à l'EPFL, et ensuite il faut renvoyer la solution à l'ordinateur de l'utilisateur. Les deux graphiques suivants

montrent la demande de connexion de l'utilisateur au serveur (figure 3.1) et la connexion entre le serveur et l'utilisateur(figure 3.2) :

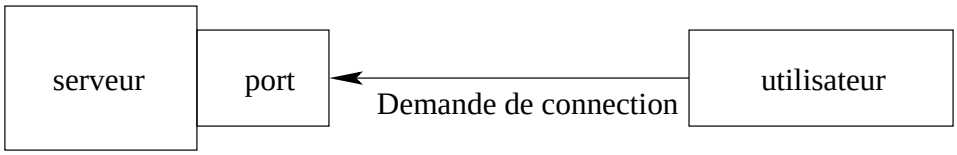

FIG. 3.1 – *Demande de connection de l'utilisateur au serveur*

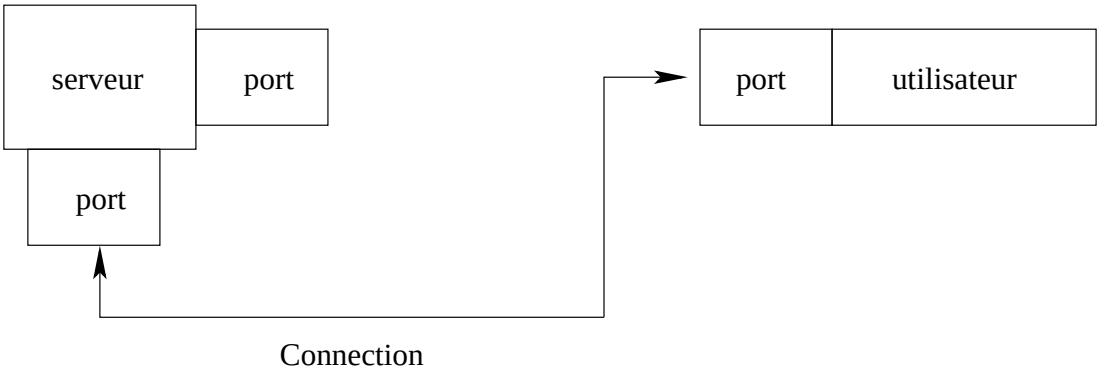

FIG. 3.2 – *Connection*

3.6 Version finale

A la fin, Jean-François Hêche a écrit les fichiers des serveurs et a modifié certaines de mes classes et de mes méthodes afin d'avoir une meilleure présentation graphique. J'ai encore dû écrire la méthode lisant la solution que renvoie **CPLEX** ou **lp_solve**, ainsi que modifier la méthode dessinant la solution.
Après corrections des nombreuses erreurs encore présentes, j'ai enfin pu voir sur l'écran l'affichage d'une solution dans le cas où l'on ne tient pas compte de la gravité. De plus, j'ai modifié l'épaisseur des lignes représentant les forces ainsi que les barres des structures pour qu'elle soit en rapport avec la valeur de la force ou de la variable.

La version avec gravité ne fonctionne toujours pas. Il se peut que le modèle mathémathique soit faux et qu'il faille tenir compte des sections dans le programme linéaire que l'on cherche

à optimiser.

On peut remarquer que parfois **CPLEX** et **lp_solve** ne trouvent pas la la même structure (n'emploient pas les mêmes barres) bien que la valeur de la fonction objectif soit la même (ils trouvent le même volume minimal). Ceci montre que quelquefois, il y a plusieurs solutions possibles comme l'illustre les figures suivantes dans un maillage 7x7 (figure 3.3, ainsi 3.4 et 3.5 page 18)

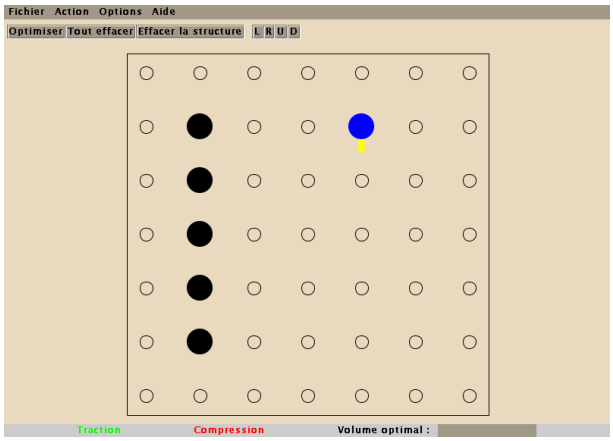

FIG. 3.3 – *Définition des points d'ancrage et de force.*

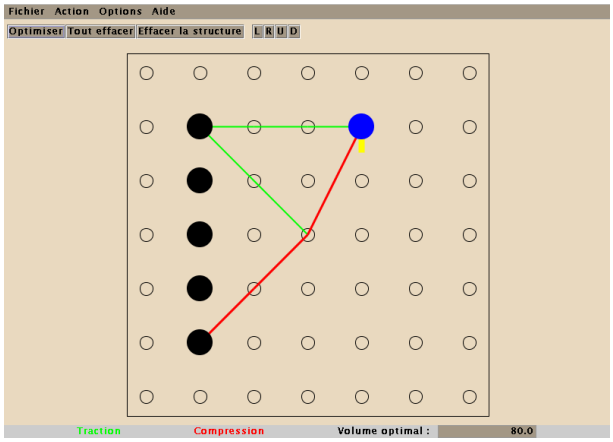

FIG. 3.4 – *Résultat de l'optimisation avec* **lp_solve**

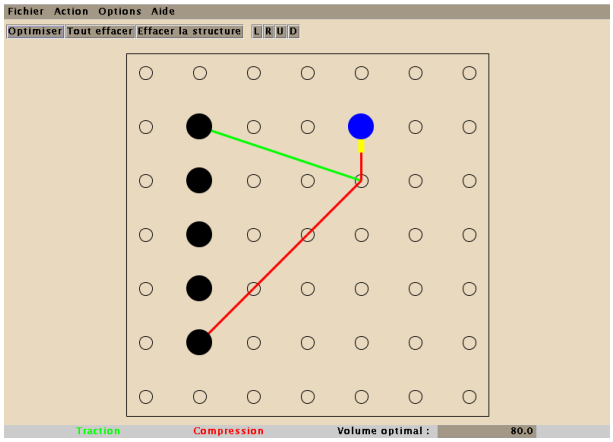

FIG. 3.5 – *Résultat de l'optimisation avec* **CPLEX**

Les deux moteurs de calculs obtiennent le même volume optimal qui est 80.

Chapitre 4

Programme en java

4.1 Généralités

Le programme Java est constitué de 10 classes :

- Force.java
- Forcedialog.java
- MoveDialog.java
- NewDialog.java
- OptStruct.java
- Options.java
- PanelGraphique.java
- PasdesolDialog.java
- ResistDialog.java
- Structure.java

Toutes ces classes se trouvent dans le package `epfl.roso.applet.optstruct`

La classe principale qui dépend de JApplet est **OptStruct.java**.
De plus, nous utilisons pour l'optimisation la version **C** de **lp_solve** ou **CPLEX**.

En cours d'élaboration, j'avais créé d'autres classes qui ont disparu dans la version finale de l'applet. En effet, j'avais par exemple créé des dialogues pour le choix de la gravité ou du format d'optimisation ; ceux-ci ont disparu et ont été remplacés par des radio-boutons dans le menu de l'applet.
J'avais aussi créé une boîte de dialogue m'indiquant s'il y avait une solution et qui me servait uniquement quand je travaillais chez moi le week-end. En effet, je travaillais avec **Forte** et je n'arrivais pas à obtenir les output (qu'on a dans un Terminal avec Unix ou Linux).

4.2 Force.java

Cette classe a été créée pour pouvoir déterminer un objet Java nommé *force*. Une *force* est constituée de deux points, le premier représentant les coordonnées du point sur lequel la force s'applique et le deuxième les composantes de la force.

Cette classe comporte quatre méthodes :

`void setCoordForces(Point p)` qui permet de fixer les coordonnées d'une force.
`Point getCoordForces()` qui retourne les coordonnées d'une force.
`void setValForces(Point p)` qui permet de fixer les composantes d'une forces.
`Point getValForces()` qui retourne les composantes d'une force.

4.3 Forcedialog.java

Cette classe est un frame qui apparaît lorsque l'utilisateur veut ajouter une force et qui permet d'entrer au clavier les valeurs désirées pour la force cliquée.

Les valeurs *Composante horizontale* et *Composante verticale* sont captées au moyen de JTextFields.

Le constructeur de cette classe prend comme paramètre `OptStruct`.

4.4 MoveDialog.java

Cette classe est un frame qui apparaît lorsque l'utilisateur clique sur *Action* puis *déplacer*. Lorsque l'on clique sur une des directions, tous les forces et ancrages se déplacent dans la direction choisie. Si un point sort du canevas alors il disparaît définitivement.
Ce frame fait le même travail que les boutons se trouvant sur la barre d'outils de l'applet : L R U D. De plus, sa présentation graphique n'est pas bonne.
Le constructeur de cette classe prend comme paramètre `OptStruct`.

4.5 NewDialog.java

Cette classe est un frame qui apparaît lorsque l'utilisateur clique sur *Fichier* puis *Nouveau* dans le menu principal. Ce dialogue permet de définir le nombre de points horizontaux et

verticaux du maillage au moyen de JSliders et la grandeur du voisinage au moyen d'un JTextField.

Quand l'utilisateur clique sur *OK*, les différentes valeurs choisies sont transmises à la classe **Structure.java**. Le constructeur de cette classe prend comme paramètre `OptStruct`.

4.6 OptStruct.java

C'est la classe qui fait le lien avec les autres classes et qui définit la structure de départ de l'applet.
Le menu est créé avec la commande JMenuBar et la barre d'outils avec JToolbar.
C'est dans cette classe que l'applet est initialisée.
On y trouve les différentes méthodes liées au menu, comme celle permettant d'effacer tout, d'effacer la solution ou d'optimiser. La plupart de ces méthodes appellent des méthodes faisant partie de **Structure.java**.
C'est aussi dans cette classe que l'on ouvre le socket permettant de faire la connexion avec le serveur du DMA et que l'on appelle la fonction permettant de générer le PL. De même on y trouve aussi la méthode permettant de lire la solution renvoyée par le moteur de calculs et celle fermant le socket une fois l'optimisation terminée et la solution lue. Après avoir été lue la solution est envoyée dans **Structure.java** d'où elle pourra être utilisée.

4.7 PanelGraphique.java

C'est la classe qui s'occupe de tous les éléments graphiques, c'est-à-dire elle dessine le canevas, les points d'ancrage, les points de force, les vecteurs des forces et la solution.
On trouve aussi dans ce fichier un listener, appelé **CanevasListener**, qui permet au programme de réagir aux clics de souris sur le canvas. En effet, un clic du bouton de gauche de la souris sur un point du canevas crée un point d'ancrage à cet endroit, tandis qu'un clic droit y crée une force.

Pour le dessin des noeuds du canevas, on emploie des cercles. Pour le dessin des vecteurs des forces, on utilise simplement des lignes partant du centre des disques. En effet, il n'existe pas de fonction prédéfinie en Java permettant de dessiner une flèche. Ces lignes sont dessinées avec la classe Graphics2D existant en Java et qui permet par exemple de définir l'épaisseur des lignes. J'ai fait varier cette épaisseur en faisant un rapport de la norme de la force considérée avec la norme de la force maximale. Le dessin de la solution se fait de la même façon. Pour dessiner la solution, on appelle la méthode `getVariable(int i)` de la classe **Structure.java**, car c'est là qu'est stockée la valeur de chaque variable après résolution.

Ce canevas est ensuite inclus dans le ContentPane de la classe **OptStruct.java**.

Le constructeur de cette classe prend comme paramètre `OptStruct`.

4.8 PasdesolDialog.java

Cette classe est un frame qui apparaît lorsqu'il n'y a pas de solution pour la structure proposée par l'utilisateur. Il se referme en cliquant sur *OK*.

4.9 ResistDialog.java

Cette classe est aussi un frame qui permet de fixer le pourcentage de traction et de compression pour les barres. Pour le moment aucune action n'est liée à ce dialogue.

4.10 Structure.java

On peut dire que c'est la classe la plus importante du projet. En effet, c'est dans celle-ci que presque tout est défini ou stocké, comme par exemple la fonction objectif, les contraintes ou encore la solution du problème. La plupart des méthodes présentes dans cette classe sont utilisées par d'autres classes.

On y trouve :

- Les méthodes permettant de fixer le nombre de points horizontaux et verticaux du canevas : `void setXSize(int s)` et `void setYSize(int s)`.
- Les méthodes retournant le nombre de points horizontaux et verticaux du canevas : `int getXSize()` et `int getYSize()`.
- Les méthodes fixant et retournant la valeur du maillage défini : `void setGridSize(int x,int y)`, `void setGridSize(Point s)` et `Point getGridSize()`.
- Différentes méthodes concernant les points d'ancrage. Certaines permettent de vérifier s'il existe déjà un point d'ancrage à un point donné : `existAnchor(int i,int j)` et `existAnchor(Point p)`, d'autres permettent de supprimer un point d'ancrage existant : `deleteAnchor(int i,int j)` et `deleteAnchor(Point p)`, d'autres encore

d'ajouter un point d'ancrage : `addAnchor(int i,int j)`, et `addAnchor(Point p)`, la dernière permet d'obtenir un itérateur du vecteur des points d'ancrage : `Iterator getAnchorIter()`. Il existe aussi une méthode retournant le nombre de points d'ancrage : `int numberOfAnchors()`.

- Les mêmes méthodes qu'au point précédent mais concernant les forces : `boolean existeForce(int i,int j)`, `boolean existeForce(Point p)`, `void deleteForce(int i,int j)`, `void deletForce (Point p)`, `void addForce (int i,int j,int k,int l)`, `void addForce(Force f)`, `Iterator getForcesIter()`,`int numberOfForces()`.

- Une méthode permettant de translater les points d'ancrage et les forces : `void translate(Point d)`.

- Une méthode permettant d'effacer tout ce qui se trouve sur le canevas : `void removeAll()`.

- Les méthodes permettant de fixer les valeurs de compression et de traction : `void setValeurCompression(int c)` et `void setValeurTraction(int t)`.

- Les méthodes retournant les valeurs de compression et de traction : `int getValeur-Compression()` et `int getValeurTraction()`.

- Les méthodes permettant de fixer et retourner la valeur de la gravité : `void setGravity (double t)` et `double getGravity()`.

- Les méthodes permettant de fixer la valeur k du voisinage et de retourner cette valeur : `void setNeighborhoodSize(int s)` et `int getNeighborhoodSize()`.

- La méthode permettant de fixer les vecteurs de voisinage possibles selon la valeur fixée k : `void computeNeighborhood()`. Pour faire cela, on crée une double boucle sur i et j et on regarde si i et j sont premiers entre eux. S'ils le sont, alors on ajoute le vecteur (i,j) à la liste des vecteurs. Pour savoir si deux nombres sont premiers entre eux il existe une méthode dans **java.math** :`BigInteger gcd(BigInteger val)`. Cette méthode emploie des BigInteger comme paramètres.

- La méthode donnant le numéro d'une barre si on connaît les coordonnées du début et de la fin de la barre : `int numeroVariable(int a,int b,int c,int d)`.

- Les méthodes donnant les coordonnées du début ou de la fin de la barre si on connaît le numéro de la barre : `Point coorddebutbarre(int num)` et `Point coordfinbarre (int num)`. Ces méthodes sont utilisées dans **PanelGraphique.java** dans la méthode dessinant la solution.

- La méthode permettant de donner le nombre de barres, c'est-à-dire le nombre de variables qui peuvent être considérées : `int getNbVar()`.

– La méthode permettant de fixer la fonction objectif du programme linéaire : `void setFonctionObjectif(PrintWriter out,String format)`. Pour faire cela on emploie l'algorithme suivant :

On parcourt tous les points du maillage.
On parcourt tous les voisins.
On calcule la valeur du voisin courant.
Si le voisin courant est en-dessus ou directement à droite :
On remplit la table de hachage.
On écrit la fonction objectif.

Cette méthode prend comme paramètre le format d'optimisation, car selon celui-ci la fonction objectif ne doit pas être écrite de la même façon. Par exemple, pour lp_solve on doit mettre un point-virgule à la fin de la fonction objectif et pour CPLEX on ne doit pas le faire.

– La méthode permettant de fixer les contraintes : `void setContraintes(PrintWriter out,String format)`. Cette méthode emploie l'algorithme suivant :

On parcourt tous les points du maillage.
On initialise le vecteur contraintesX et le vecteur contraintesY.
On calcule la valeur du point courant.
On regarde si le point courant est une ancre.
Si le point courant est une ancre : on passe au point suivant.
Sinon :
On parcourt tous les voisins.
On calcule la valeur du voisin courant.
On calcule les contraintes.
On cherche le numéro de la variable.
On remplit contraintesX, contraintesY.
On écrit la contrainte.

– La méthode permettant de générer le PL : `void printLP(PrintWriter out,String format)`. Cette méthode appelle les fonctions `setFonctionObjectif (PrintWriter out,String format)` et `setContraintes(PrintWriter out,String format)`.

– Les méthodes permettant de fixer et retourner la valeur de la solution de la fonction objectif : `void setValeurFctObj(double v)` et `double getValeurFctObj()`. Cette valeur est fixée et est utilisée dans **OptStruct.java**.

– Les méthodes fixant le vecteur des solutions des variables : `setVecteurVariables(int i,double v)` et retournant la valeur de la variable i : `double getVariable(int i)`. Ces valeurs sont fixées dans **OptStruct.java** et sont utilisées dans **PanelGraphique.java** pour le dessin de la solution.

- Des méthodes employant des booléens qui indiquent si la structure a été optimisée : `void setOpt(boolean j)` et `boolean getOpt()`.

- Et enfin des méthodes retournant la valeur de la variable maximale : `double getVariableMax()` et la force maximale (en norme) : `double getForceMax()`. Ces deux méthodes sont utilisées dans la classe **PanelGraphique.java** pour déterminer l'épaisseur des traits.

J'ai décrit en détail toutes ces méthodes car elles sont employées dans d'autres classes de l'applet et il est important de savoir ce qu'elles font.

Chapitre 5

Mode d'emploi

5.1 Installer Java chez soi

On peut télécharger le JDK (Java Development Kit) sur la page web [9]. La version que j'ai utilisée pour mon programme est la 1.3 . La documentation relative est disponible également sur cette page. Suivant le système d'exploitation que vous utilisez, vous serez orienté vers une des différentes pages permettant le téléchargement. Ensuite, il faut configurer le PATH en indiquant le chemin d'accès au kit.

5.2 Mode d'emploi de l'applet

Cette applet permet de déterminer une structure de poids (volume) minimum supportant une charge donnée.

Le canevas (quadrillage) présent au lancement de l'applet est formé de 15 points horizontaux et 5 verticaux et est muni d'un voisinage de taille 3.

Pour changer la taille du maillage ou du voisinage cliquer sur *Fichier*, puis *Nouveau* ; choisir les valeurs désirées et cliquer sur *OK*.
Lorsque vous avez la bonne taille du canevas, un clic du bouton gauche de la souris sur un noeud donne un point d'ancrage (représenté par un point noir) , un clic du bouton droit de la souris sur un noeud donne un point de force (représenté par un point bleu) - il vous faut alors entrer les composantes de la force dans la boîte de dialogue qui apparaît.
Un clic sur un point d'ancrage ou point de force existant supprime celui-ci. Vous pouvez effacer tout ce qui se trouve sur le canevas en cliquant sur *Tout effacer* et effacer le résultat de l'optimisation en cliquant sur *Effacer la structure*.

Pour visualiser les directions des forces, il faut aller dans *Options* et activer *Afficher les forces*.

Pour choisir la valeur de la gravité, cliquer sur *Options* puis *Gravité* ensuite cocher la case désirée. Actuellement, il vous faut choisir *Pas de gravité* pour obtenir une solution à votre structure.

Pour choisir le format d'optimisation cliquer sur *Options* et *Moteur de calculs* puis activer le format désiré. Attention, seulement les utilisateurs provenant de l'EPFL peuvent choisir d'optimiser avec **CPLEX**. Il faut aussi savoir que **lp_solve** rencontre des problèmes numériques pour des grandes structures, ce qui a pour conséquence le fait qu'il ne trouve pas de solutions dans ces cas-là. Le format d'optimisation par défaut est **lp_solve**.

Pour déplacer tous les points (forces et ancrages) situés sur le maillage, cliquer sur la lettre de la barre d'outils correspondant à la direction désirée ; L : gauche, R : droite, U : haut, D : bas.

Après avoir choisi vos forces et ancrages, cliquer sur *Optimiser* pour activer la recherche d'une solution.

L'option *Options* - *Résistance* n'est pas encore active, donc un changement des valeurs ne se répercutera pas sur les résultats d'optimisation.

Chapitre 6

Conclusion

L'applet actuelle fonctionne sans problème avec **lp_solve** et **CPLEX** pour autant que l'on ne tienne pas compte de la gravité. Pour le moment, cette applet est comparable au programme **Optstruct.tcl** vu que celui-ci ne tient pas compte de la gravité et optimise les PL avec **CPLEX**. En revanche, le champ d'action de l'applet est nettement plus vaste parce qu'elle va être mise sur le web et qu'ainsi tout le monde y aura accès et pourra tester des structures ; en effet **lp_solve** (qui est le moteur de calculs accessible pour tout le monde) permet quand même d'optimiser des structures de taille moyenne.
La représentation graphique actuelle est bonne grâce aux modifications apportées par Jean-François Hêche.

Il est clair qu'il y a de nombreuses améliorations possibles.
Premièrement, il faudrait trouver une meilleure modélisation, qui tienne compte du poids propre de la structure et qui permette d'obtenir une solution lors de l'optimisation. Il faudrait pour cela réfléchir à la possiblité de faire intervenir les sections dans le programme linéaire.
Deuxièmement, il faudrait encore activer l'option *Résistance*.
On pourrait aussi améliorer le graphisme de certaines boîtes de dialogues comme **NewDialog.java** ou **MoveDialog.java**.
De plus, actuellement l'applet ne prend en compte que des valeurs entières pour les forces. Si l'utilisateur rentre des valeurs réelles, celles-ci ne sont pas acceptées (cela provoque une exception du type : «NumberFormatException»). Il faudrait donc changer cela afin que l'on puisse entrer des valeurs réelles.

Ce projet m'a permis d'apprendre les bases du langage Java et de mettre en pratique les concepts de la programmation orientée objet. En effet, les problèmes que j'ai dû résoudre m'ont permis d'avoir une vision étendue des possibilités offertes par ce langage qu'il me sera certainement possible de mettre à profit dans d'autres situations.
Comme Internet prend de plus en plus de place dans notre société, il est indispensable de connaître un langage permettant de mettre des programmes sur le web. De plus, grâce à la création de la page web, j'ai pu acquérir quelques connaissances du langage HTML.

La réalisation de ce travail m'a beaucoup intéressée et j'ai passé beaucoup de temps à résoudre les différents obstacles qui se sont dressés tout au long de la conception du projet. Il faut dire qu'ayant en fin de compte peu de connaissances de la programmation à ce niveau, je me suis parfois énervée car il me semblait que je n'avançais pas. Je suis très contente d'avoir enfin réussi à obtenir une applet qui fonctionne.

J'ai choisi ce projet parce qu'il était utilisable pour d'autres personnes et parce qu'il me permettait de faire quelque chose de concret. De plus, il réunissait plusieurs domaines tels que la recherche opérationnelle et l'informatique qui sont des sujets qui m'intéressent particulièrement.

Bibliographie

[1] Jean-François Hêche, Calcul à la rupture et optimisation structurelle, Recherche opérationnelle pour ingénieurs, Section génie civil 2e année, Etude de cas N2.

[2] Rogers Cadenhead, JAVA2 Plate-forme, Le tout en poche, CampusPress, 1999.

[3] Brit Schröter, Java 2, Micro Application, 2000, Paris.

[4] http ://www.eteks.com/.

[5] http ://sunline.epfl.ch/for_EPFL/javatut_13Oct2000/.

[6] http ://sunline.epfl.ch/Java/jdk1.3/docs/api/index.html.

[7] http ://www.cs.wustl.edu/ javagrp/help/LinearProgramming.html.

[8] http ://elib.zib.de/pub/Packages/mathprog/linprog/lp-solve/harmut_documentation

[9] http ://java.sun.com/j2se/ ?frontpage-javaplatform.

[10] Krishna Sankar, et al., Bibliothques de classes Java2, CampusPress France, 1999.

[11] Christian Gruber, Willy Benoit, Mécanique générale, PPUR, 1998.

[12] http ://www.princeton.edu/∼rvdb/.

Table des figures