Développement d'un robot à fils contractibles

Félix Chénier

Août 2004

Étude faite dans le cadre d'une bourse CRSNG
à l'École Polytechnique de Montréal

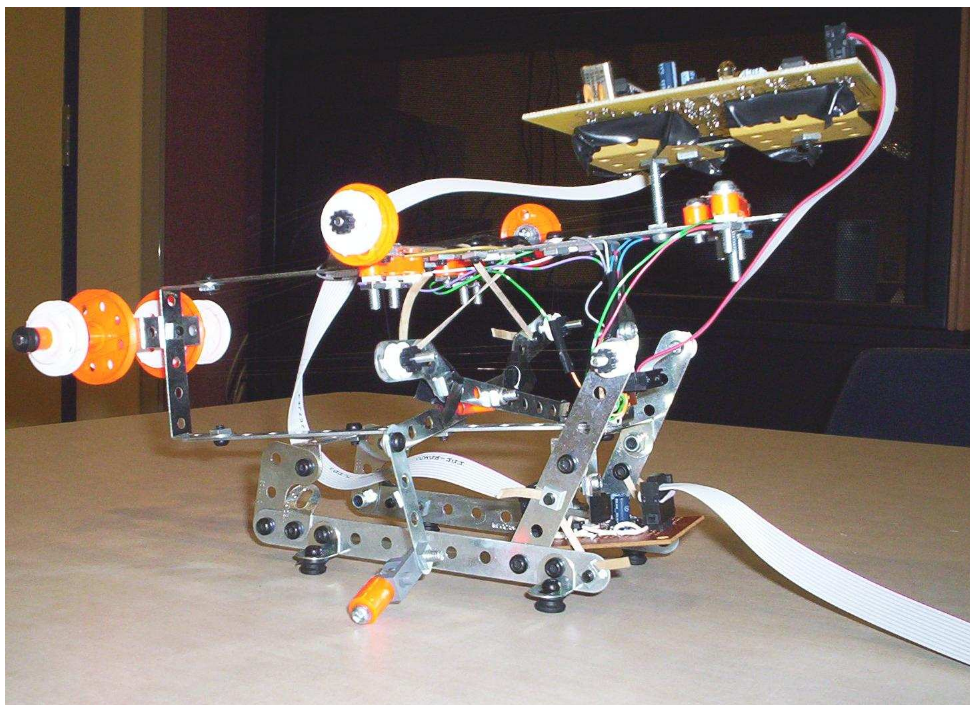
Résumé

Les matériaux à mémoire de phase ne sont pas nouveaux, mais leur utilisation générale est actuellement plutôt limitée. La recherche effectuée pour ce document montre qu'il est en fait possible d'utiliser des fils contractibles (un alliage de Nickel et de Titanium nommé *Nitinol*) pour articuler un objet physique tel un robot.

Un des problèmes majeurs constaté est la vitesse faible à laquelle le fil peut se détendre à l'air libre. Ce problème est atténué en alimentant le fil avec un signal modulé en largeur d'impulsion (PWM[1]). De plus, cette technique permet un contrôle précis de la longueur du fil. La longueur n'est plus limitée à l'état "contracté" ou "détendu".

Ainsi, deux ébauches de développement d'un asservissement en longueur sont suggérées : un contrôleur PID, et un réseau de neurones utilisant la méthode *p-delta*.

Finalement, utilisant un micro-contrôleur, un accéléromètre et des pièces électroniques communes, ce document montre comment on peut appliquer les fils de *Nitinol* à des montages simples et utiles. Le terme de cette recherche conduit à la création d'un robot marcheur dont les étapes de fabrication sont toutes indiquées.



[1] De l'anglais : ***P**ulse **W**idth **M**odulation*

Table des matières

Introduction 1

I Conception et fabrication du système de base 1

1 Fonctionnement des fils contractibles 1
- 1.1 Terminologie . . . 2

2 Choix de grosseur et manipulation du fil 2
- 2.1 Grosseur des fils . . . 2
- 2.2 Attache du muscle . . . 3

3 Modèles préliminaires 4
- 3.1 Modèle à déclenchement . . . 4
  - 3.1.1 But . . . 4
  - 3.1.2 Principe . . . 4
  - 3.1.3 Réalisation . . . 5
  - 3.1.4 Avantages . . . 5
  - 3.1.5 Inconvénients . . . 5
- 3.2 Testeur de robustesse . . . 6
  - 3.2.1 But . . . 6
  - 3.2.2 Principe et réalisation . . . 6
  - 3.2.3 Circuit *driver* . . . 7
  - 3.2.4 Résultats . . . 7
- 3.3 Modèle antagoniste . . . 8
  - 3.3.1 But . . . 8
  - 3.3.2 Principe et réalisation . . . 8
  - 3.3.3 Résultats . . . 8
- 3.4 Bras multi-niveaux . . . 9
  - 3.4.1 But . . . 9
  - 3.4.2 Problème avec la saturation en longueur . . . 9
  - 3.4.3 Contrôle précis de la température du muscle . . . 9
  - 3.4.4 Réalisation du circuit à PWM . . . 10
  - 3.4.5 Montage physique . . . 10
  - 3.4.6 Programmation du micro-contrôleur . . . 11
  - 3.4.7 Programmation de *Matlab* . . . 11
  - 3.4.8 Expérimentation et résultats . . . 12

4 Modèle final 13
- 4.1 Capteur d'inclinaison . . . 13
  - 4.1.1 Type de capteur . . . 13
  - 4.1.2 Terminologie . . . 14
  - 4.1.3 Branchement de l'accéléromètre . . . 14
  - 4.1.4 Montage de l'accéléromètre . . . 14
  - 4.1.5 Programmation du capteur d'inclinaison . . . 15
  - 4.1.6 Programmation de la communication de l'inclinaison . . . 15
- 4.2 Protecteur contre la surchauffe . . . 16
  - 4.2.1 Principe . . . 16
  - 4.2.2 Programmation du protecteur . . . 17
  - 4.2.3 Communication du bit de protection . . . 18

4.3 Schéma-bloc du modèle final 18

II Asservissement du système 19

5 Type d'asservissement 19
5.1 Caractéristiques du système 19

6 Contrôleur PID 19

7 Principe de la méthode *p-delta* sur un perceptron parallèle 20
7.1 Perceptron 20
7.2 Perceptron parallèle 20
7.3 Algorithme d'apprentissage *p-delta* 21
7.4 Résultats préliminaires sur *Matlab* 22

8 Implantation de la méthode *p-delta* sur le système 23
8.1 Principe 23
8.2 Résultats 23

III Fabrication du robot 24

9 Modèle physique 24
9.1 Principe de motion 24
9.2 Dimensions 26
9.3 Montage 26

10 Circuits électroniques 26
10.1 Circuit *driver* 26
10.1.1 Spécification du circuit *driver* 28
10.2 Montage sur circuit imprimé 28
10.3 Circuit de conversion de l'UART 29
10.4 Traitement de l'inclinaison 31

11 Programmation 31
11.1 Communication des informations 31
11.2 Calcul de l'inclinaison sur deux axes 32

12 Résultat final 32

Conclusion et améliorations 33

Annexe 34

A Notes techniques sur l'utilisation du robot 34
A.1 Alimentation 34
A.2 Connecteurs 35
A.3 Manipulation du robot 35

B Code en C du micro-contrôleur 35

C Fichiers *Matlab* pour la communication 45
C.1 Initialisation du port série 45
C.2 Envoyer un octet 45
C.3 Assigner un niveau à un muscle 45
C.4 Attendre n secondes 46
C.5 Marcher 46

D Matériel utilisé 47
D.1 Expérimentation 47
D.2 Rédaction de ce document 47

E Références 47

F Remerciements 47

Table des figures

1 Courbe d'hystérésis (température/longueur) 2
2 Fixation du muscle sur le montage 3
3 Principe du modèle à déclenchement 4
4 Réalisation du modèle à déclenchement 5
5 Testeur de robustesse 6
6 Propositions de circuit *driver* 7
7 Modèle antagoniste 8
8 Illustration du retard causé par la saturation en longueur 9
9 Séquence des informations de l'ordinateur au muscle 10
10 Réalisation du bras multi-niveaux 11
11 Accéléromètre MXD2020ML 14
12 Schéma électrique du circuit entourant l'accéléromètre 14
13 Montage de l'accéléromètre 15
14 Calcul d'une intégrale discrète pour protéger les muscles 17
15 Méthode de mise à jour de l'intégrale 17
16 Schéma-bloc représentant le système final à asservir. 19
17 Fonctionnement d'un perceptron parallèle 20
18 Approximation d'un plan par la méthode *p-delta* 23
19 Modélisation 3D du principe de motion du robot 25
20 Raccordement des membres à l'aide de deux fils par articulation 26
21 Photo du robot 27
22 Schéma électrique du *driver* de courant utilisé pour les muscles du robot 28
23 Circuit imprimé pour le contrôle du robot 29
24 Convertisseur UART - 0/5V 30
25 Photo du convertisseur UART - 0/5V 31
26 Diagramme d'états pour le calcul des deux inclinaisons 32
27 Schéma du système du robot 33
28 Photo de l'aspect final du robot 34
29 Connecteurs 35

Liste des tableaux

1 Avantages et inconvénients par rapport au diamètre des fils 2
2 Voltage entre les sorties des inverseurs en fonction des bits d'entrée 7
3 Illustration de la lecture de l'inclinaison à partir de la gravité. 13
4 Entrées et sorties du réseau de neurones 23
5 Spécifications sur les symboles du circuit imprimé 30
6 Description des symboles de la figure 24 31
7 Commandes reconnues par le programme 31
8 Significations des pins des connecteurs de la figure 29 36
9 Numéros et fonctions des muscles 37

Introduction

Les matériaux à mémoire de phase ne sont pas nouveaux, mais leur utilisation générale est actuellement plutôt limitée. On voit de plus en plus, sur internet, de nouveaux procédés et modèles de plus en plus puissants, mais les applications restent rares.

Le sujet de cette recherche se concentre sur l'utilisation d'un type de matériau à mémoire de forme, le *Nitinol*. En premier lieu, on tente de voir comment on peut utiliser efficacement ce matériau sous la forme d'un fil contractible. Ainsi, des circuits électriques et des méthodes d'alimentation et d'application physique sont explorés.

Par après, on tente de trouver un moyen pour asservir un fil contractible en longueur, utilisant les méthodes trouvées lors de la première partie. Deux approches sont tentées : un contrôleur PID et un réseau de neurones.

Finalement, on utilise les notions intéressantes trouvées lors de cette recherche pour créer un robot articulé par des fils de *Nitinol*. La grande majorité des informations nécessaires pour la fabrication d'un tel robot est présente dans ce document.

Première partie
Conception et fabrication du système de base

Avant de créer le robot, il est important de se familiariser avec les différentes parties qui le composeront. Les fils contractibles étant quand même nouveaux, il n'y a pas encore eu beaucoup d'utilisations pratiques et il importe de voir comment les utiliser efficacement avant même de penser à faire un robot.

De plus, comme on veut contrôler le robot par des moyens simples (via *Matlab*), on doit développer un système qui sera capable de communiquer efficacement avec l'ordinateur, que ce soit pour lui dicter ses commandes (niveau de contraction désiré pour chaque muscle) ou bien pour recevoir ses informations (inclinaison, état du robot).

Cette partie du document permet donc de mettre en place tous les éléments dont aura besoin le robot, en ayant comme objectif un système simple sur lequel celui-ci se basera éventuellement.

1 Fonctionnement des fils contractibles

Les fils utilisés sont des fils de Nitinol provenant de la compagnie *Mondo-Tronics*[2]. Ces fils sont vendus sous le nom de Muscle Wires. Bien que le fonctionnement de tels fils soit très bien expliqué dans le livre *Muscle Wires Project Book* livré avec le kit de départ, voici un court résumé du principe.

Le Nitinol est un alliage spécial de Nickel et de Titanium à mémoire de forme. Lorsqu'on le réchauffe jusqu'à une température de seuil, il se met à raccourcir d'environ 3 à 5% (pratiquement deux fois plus s'il est neuf, mais ça revient à ce pourcentage après seulement quelques cycles de contraction/décontraction). Inversement, lorsqu'on le fait redescendre à une température de seuil et qu'on lui applique une certaine force en tension, l'alliage reprend sa forme normale.

[2]Site Internet : www.mondo-tronics.com

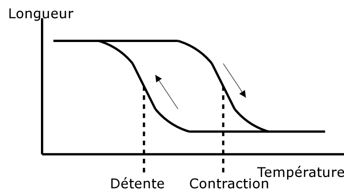


FIG. 1 – Courbe d'hystérésis (température/longueur)

On constate sur la figure 1 qu'il existe un phénomène d'hystérésis vis-à-vis la déformation : on doit revenir à une température légèrement plus basse que la température de contraction pour que le fil se détende. Ceci contribue, en plus de la saturation en longueur, à la non-linéarité de la longueur du muscle par rapport à la température.

Le fil de Nitinol est un conducteur relativement bon, mais possède tout de même une résistance interne non négligeable. Ceci nous permet de réchauffer facilement le fil en faisant passer du courant à travers celui-ci. Les valeurs de courant recommandées changent d'un type de fil à l'autre (et d'une grosseur à l'autre).

1.1 Terminologie

Afin d'alléger le texte, nous utiliserons lors de cette première partie le terme *muscle* pour désigner le fil de nitinol. Ce terme n'est pas rigoureux dans le sens où un muscle est généralement composé de plusieurs fibres. Par contre, comme tous les modèles ont été testés avec seulement un fil à la fois, il n'y a pas risque de confusion

2 Choix de grosseur et manipulation du fil

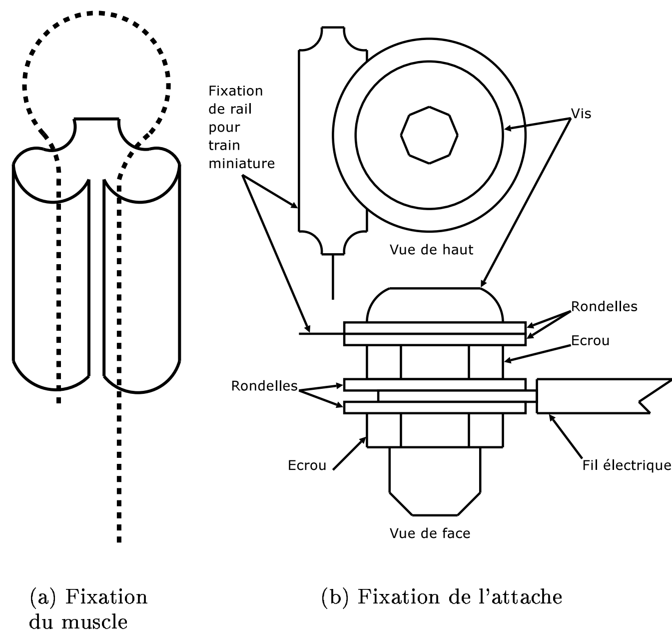
2.1 Grosseur des fils

Le kit de départ contient trois grosseurs de fils du type LT[3] (*Low Temperature*), ayant respectivement un diamètre de 50, 100 et 150μm. Chacun des diamètres a ses avantages, mais le diamètre de 100μm s'avère plus intéressant.

Diamètre	Avantages	Inconvénients
50μm	Détente la plus rapide	Difficile à manipuler ; Le frottement statique du montage est important par rapport à la force développée par le fil.
100μm	Relativement aisé à manipuler ; Détente assez rapide ; Assez solide pour le tester adéquatement sans le mettre en parallèle avec d'autres fils.	Pas aussi rapide que le 50μm
150μm	Très solide ; Facile à manipuler.	Très lent lors de la détente

TAB. 1 – Avantages et inconvénients par rapport au diamètre des fils

[3]Le type LT ou HT correspond à la température autour de laquelle le muscle se contracte ou se détend. Pour LT, cette température se situe autour de 70°C ; pour HT, elle se situe autour de 90°C.



(a) Fixation du muscle

(b) Fixation de l'attache

FIG. 2 – Fixation du muscle sur le montage

Les montages dans cette partie utilisent, pour la plupart, des fils de type LT ayant un diamètre de 100μm. Par contre, le robot utilise des fils de type HT[4], qui se détendent plus vite étant donnée la plus grande différence entre la température ambiante et la température de détente du muscle. Quoi qu'un peu plus chers, ces fils sont très intéressants puisqu'ils rendent le système beaucoup plus rapide à température ambiante.

2.2 Attache du muscle

Un bon moyen pour attacher efficacement et durablement le muscle est de le serrer dans une attache de rail de chemin de fer miniature, ou toute pièce de métal pouvant serrer convenablement le fil. Pour qu'il soit solide, il suffit de lui faire faire une boucle autour d'une des extrémités et de le faire revenir dans l'attache, comme le montre la figure 2(a). Ensuite, on écrase le tout à l'aide d'une pince.

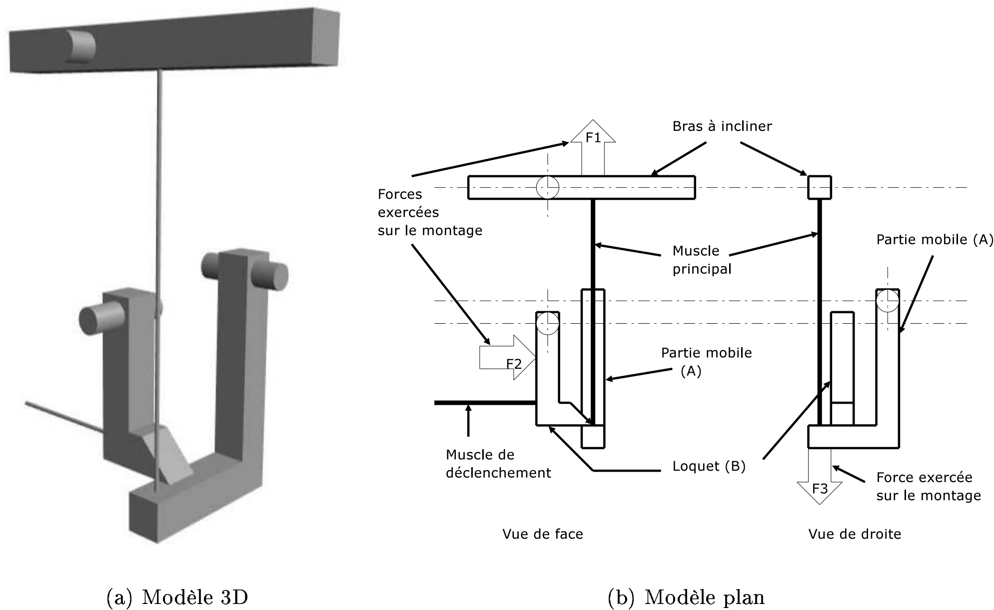
Pour le connecter au montage (structure et fil électrique), la manière qui a été la plus utilisée lors de l'expérimentation est la suivante (présentée à la figure 2(b)) :

On sert ensemble à l'aide d'un écrou deux rondelles de métal entres lesquelles on retient l'attache préparée juste avant. Ensuite, on ajoute deux autres rondelles entre lesquelles on sert la partie dénudée du fil électrique qui amènera le courant. Finalement, on peut se servir du bout de la vis pour fixer le tout sur un montage, à l'aide d'un troisième écrou.

Les montages dont il est question dans cet article ont été montés avec des *Meccano*[5], ce qui a rendu facile cette façon de procéder.

[4] *idem*

[5] Site internet : www.meccano.com



(a) Modèle 3D

(b) Modèle plan

FIG. 3 – Principe du modèle à déclenchement

3 Modèles préliminaires

3.1 Modèle à déclenchement

3.1.1 But

Le modèle à déclenchement est conçu pour faire le plus d'oscillations possible par période de temps. On s'est aperçu que la période de refroidissement est généralement plus lente que la période de chauffage. En fait, le chauffage est très rapide, on n'a qu'à donner plus de courant que nécessaire[6]. Par contre, on ne peut pas faire grand chose pour le refroidissement. Les tests ont eu lieu à environ 20°C dans un environnement aéré. L'ajout d'un ventilateur (ou simplement souffler sur le fil) a un effet notable, mais c'est quand même lent. C'est cette lenteur qu'on tente de régler à l'aide de ce modèle.

3.1.2 Principe

Une solution au problème constaté de lenteur de refroidissement consiste à n'utiliser que le cycle de contraction des muscles. Il faut donc un système de déclenchement qui relâchera la prise que le muscle a sur la pièce mobile lorsque cette pièce devra revenir à son état initial. La figure 3 illustre le système de déclenchement au repos. Si on veut abaisser rapidement le bras, on contracte le muscle principal. Ensuite, si on désire relever rapidement le bras jusqu'à sa position initiale, on contracte le muscle de déclenchement. Celui-ci tire sur un loquet qui empêche la partie mobile A de bouger. Le loquet retiré, le muscle n'est plus retenu (en supposant que la force F1 est plus grande que F3) ; le bras peut donc revenir instantanément à sa position initiale. Pendant la détente du muscle, la partie mobile A redescend et va se ré-enclencher sous le loquet.

Pour faire plusieurs cycles à l'intérieur du temps requis pour un cycle de contraction et détente, on n'a qu'à mettre plusieurs systèmes de ce type en parallèle sur le même bras, en les actionnant chacun leur tour.

[6]Ceci peut avoir un effet néfaste sur la durée de vie du muscle si l'abus de courant est maintenu.

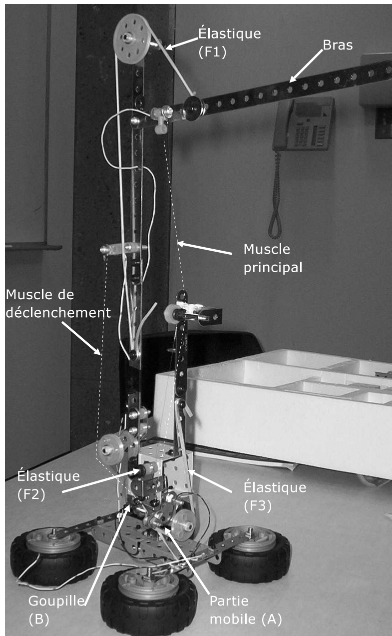


FIG. 4 – Réalisation du modèle à déclenchement

3.1.3 Réalisation

Comme tous les autres modèles, ce montage est réalisé avec des *Meccano*. Les muscles utilisés sont des *Muscle Wires* LT avec 150μm de diamètre (ce diamètre a été choisi avant tout pour sa facilité de manipulation, qui convient bien à un premier essai). Comme il est difficile de voir les muscles (en particulier sur une photo), leur trajectoire a été marquée, sur la figure 3.1.3, par de fines lignes blanches. Les forces à exercer sur le montage sont réalisées par des élastiques. Par ailleurs, l'ajustement de la longueur du muscle principal est réalisé par une poulie située à mi-chemin entre ses deux extrémités. L'ajustement du muscle de déclenchement, lui, est réalisé en déplaçant le bras qui tient son extrémité supérieure. L'expérimentation a montré que ce système fonctionne relativement bien, mais comporte plus d'inconvénients que d'avantages.

3.1.4 Avantages

- Le système fonctionne en pratique, et on pourrait théoriquement lui ajouter d'autres systèmes de déclenchement pour lui faire faire des cycles plus courts.

3.1.5 Inconvénients

- Ce système ne permet pas (ou très difficilement) d'assigner une valeur autre que *long* ou *court* au muscle. Un des moyens permettant d'avoir une courbe de longueur pseudo-continue serait de mettre plusieurs de ces systèmes en série. Or, comme la mécanique est relativement complexe, il est pratiquement impensable de reproduire ce modèle sur toute la longueur du dispositif actionné par ces muscles.
- Il est difficile de calibrer le système pour avoir des performances optimales. En effet, la distance entre le bras à son état initial (muscle principal détendu) et la partie mobile A enclenchée sous le loquet doit correspondre exactement à la longueur du muscle au repos. Autrement, si cette distance était plus longue, la partie A n'irait jamais s'enclencher, et si la distance était plus courte, une bonne partie de la contraction du muscle serait perdue dans le mouvement de la

partie A. Or, on sait qu'à mesure que les muscles vieillissent, leur différence de longueur entre l'état contracté et l'état détendu diminue. On aurait donc toujours une calibration mécanique à faire, ce qui est hors de question dans un système comprenant plusieurs muscles.
- L'absence d'un bon contrôleur de courant affecte la durée de vie des muscles. Ici, la contraction des muscles est faite manuellement, soit en appliquant directement une tension au muscle. Pour avoir une réaction rapide, il faut donner une tension fournissant plus que le courant recommandé. Ceci nous contraint à appliquer des tensions de courte durée. Par contre, il est difficile, suivant cette méthode, d'évaluer à quel point la durée d'application de la tension est nuisible ou pas pour le muscle. Toujours est-il qu'après une expérimentation de quelques cycles sur ce modèle, les muscles ont perdu une bonne part de leurs propriétés et la différence entre l'état contracté et l'état détendu était devenue trop faible pour pouvoir actionner le loquet.

3.2 Testeur de robustesse

3.2.1 But

Comme on l'a constaté dans le modèle à déclenchement, il ne sert à rien de faire un système complexe si la durée de vie d'un muscle est limitée. Le testeur de robustesse est donc un dispositif qui tente d'établir la pertinence d'utiliser des *Muscle Wires* dans un système demandant beaucoup d'oscillations. Il se doit donc de respecter exactement toutes les forces et courants recommandés[7].

3.2.2 Principe et réalisation

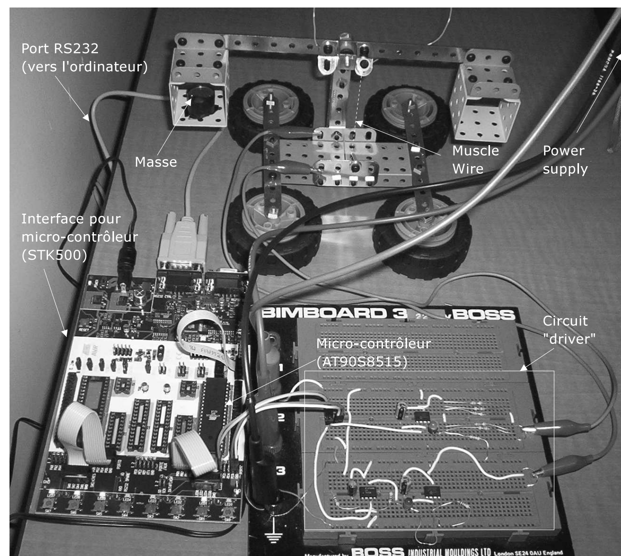


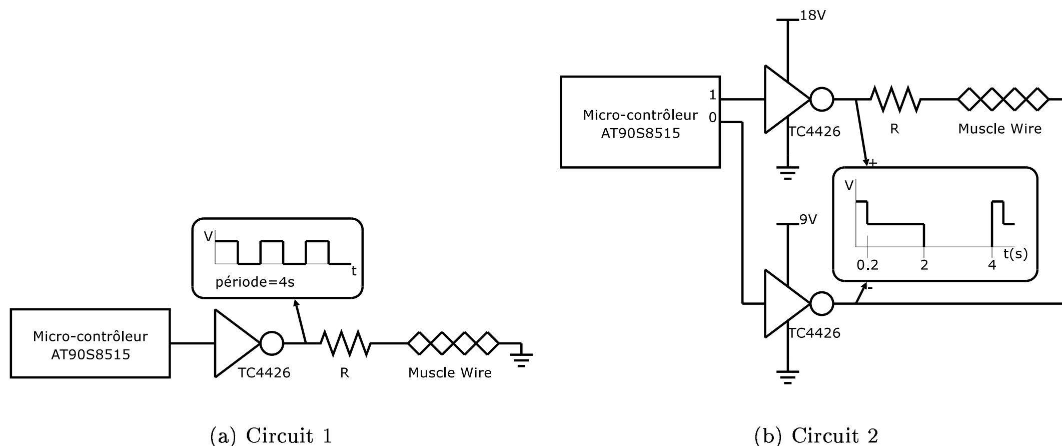
FIG. 5 – Testeur de robustesse

Comme le système doit être très précis, il est hors de question d'utiliser des élastiques comme force de rappel. La technique imaginée ici est plutôt la création d'une balance sur laquelle on installe une masse fixe. Ainsi, la force de rappel est très facile à calculer, très précise et ne varie pratiquement pas lorsque le balancier change d'angle (car l'angle reste faible). La masse et les distances sont calibrées pour avoir une force constante de 150g (0,015N) en tension sur un muscle de 100μm de diamètre. Dans le cas testé, 8 pièces de 1¢(2,5g chacune) ont été installées sur un levier de ratio 1 : 8, pour une force constante de 160g.

[7]Les valeurs recommandées se retrouvent dans le livre *Muscle Wires Project Book* (1), fourni avec le kit de départ.

3.2.3 Circuit *driver*

Pour ce qui est du circuit *driver*, deux approches sont tentées.



(a) Circuit 1

(b) Circuit 2

FIG. 6 – Propositions de circuit *driver*

La première (figure 6(a)) envoie simplement le courant recommandé pendant 2 secondes, puis arrête 2 secondes. Ici, la tension positive à la sortie de l'inverseur étant de 12V et la résistance du muscle étant 11Ω, la résistance à mettre pour atteindre 180mA est de $R = 56\Omega$.

La deuxième approche tente d'améliorer le temps de contraction du muscle en envoyant un courant deux fois plus élevé que le courant recommandé, mais seulement pendant deux dixièmes de seconde. Ainsi, il n'y a pas de risque de surchauffer le muscle. Pour le reste de la période, on envoie simplement le courant recommandé, afin de garder le muscle en tension. Le circuit de la figure 6(b) montre comment on peut s'y prendre pour envoyer 180mA ou 360mA au muscle. Comme on veut un courant d'environ 180mA avec 9V, on trouve $R = 37\Omega$. Avec 18V, on aura évidemment 360mA.

La table 2 illustre le voltage à la sortie de l'inverseur et le courant en fonction de la sortie du micro-contrôleur pour le deuxième circuit.

bit 1	bit 0	Voltage (V)	Courant (mA)
0	0	9	180
0	1	18	360
1	0	-9	-180
1	1	0	0

TAB. 2 – Voltage entre les sorties des inverseurs en fonction des bits d'entrée

3.2.4 Résultats

Le montage a premièrement effectué 5000 cycles à 2 secondes de demi-période à l'aide du premier *driver*. Comme les performances du muscle n'ont pratiquement pas changé (à part la perte normale de performance lors des premiers cycles), une autre série de 5000 cycles a été effectuée, cette fois-ci avec le deuxième *driver*. Là non plus, le muscle ne s'est pas dégradé.

Par contre, le mouvement est beaucoup plus rapide et constant avec le deuxième *driver*. On verra que ce résultat intéressant a inspiré beaucoup le levier qui servira à l'asservissement avec *Matlab*.

3.3 Modèle antagoniste

3.3.1 But

Nos membres sont soumis à des forces de sens contraires par des muscles antagonistes. Par exemple, le biceps fléchit l'avant-bras alors que le triceps y effectue l'extension. Il serait intéressant de recréer ce principe à l'aide de muscles artificiels. C'est le but du modèle antagoniste.

3.3.2 Principe et réalisation

Le *Muscle Wire* n'exerce pas directement une force sur quelque chose. Il exerce plutôt un mouvement. En fait, on ne pourrait pas simplement attacher deux de ces fils sur un bras et en fixer l'extrémité sur deux parties fixes opposées, car on pourrait ainsi briser les fils (si les deux sont en contraction en même temps). L'idée est donc de transformer ces déplacements en forces qui, elles, peuvent s'affronter sans risque de bris. Il s'agit donc de prendre un ressort et de le mettre en série avec le muscle. On aura donc $\Delta F = K\Delta x$ où ΔF est la différence de force, Δx est la différence de longueur du muscle et K est la constante de rappel du ressort.

Le modèle physique a été réalisé avec des élastiques. Bien que non-linéaire, la relation entre la force et la longueur d'un élastique peut être approximée par une droite, dans la mesure où le déplacement est faible, ce qui est notre cas.

Le premier circuit *driver* du testeur de robustesse[8]est utilisé pour contrôler les muscles. Le but n'étant pas un système rapide, mais bien un système à muscles antagonistes fonctionnel, ce circuit est le plus simple à convenir à nos besoins. Le montage consiste en deux fixations qui tiennent un élastique, un muscle, une jonction isolante, un autre muscle et un autre élastique, tous en série (voir la figure 3.3.2). Lorsqu'on contracte le muscle 1, on s'attend à ce que la jonction bouge vers la gauche ; on s'attend au contraire pour le muscle 2.

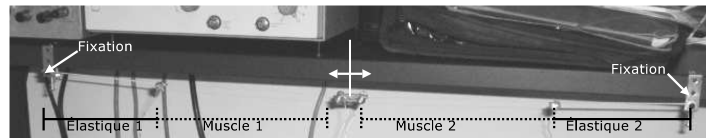


FIG. 7 – Modèle antagoniste

3.3.3 Résultats

Les résultats de cette expérimentation ont été médiocres. Il est clair que le déplacement de l'extrémité du muscle est converti en force. Par contre, rien n'indique que cette force doit fournir un travail. La jonction entre les deux muscles ne bouge donc pratiquement pas, en particulier à cause du trop faible mouvement des muscles.

On ne doit pas non plus oublier que la jonction ne pourra se déplacer que d'un maximum de la moitié du déplacement d'un muscle, car la force développée par une contraction se distribue également dans les deux élastiques.

Une étude pourrait être faite sur la possibilité d'utiliser de grandes quantité de *Muscle Wires* pour parvenir à un mouvement utilisable. Par contre, comme le prix de tels fils est encore assez élevé et qu'il est assez aisé d'utiliser d'autres méthodes pour un bon contrôle des bras d'un système, nous oublierons ce principe pour l'étude actuelle.

[8]Voir la figure 6(a)

3.4 Bras multi-niveaux

3.4.1 But

Ce modèle tente de compenser pour l'échec du modèle antagoniste, tout en ajoutant une précision qu'on ne pouvait obtenir avec les modèles précédents. On utilise en fait la modulation en largeur d'impulsion (PWM[9]) pour définir précisément la longueur que doit avoir le muscle. De cette manière, on peut utiliser une force de rappel constante au lieu d'un muscle antagoniste, et s'arranger pour que le muscle puisse faire passer le bras d'une façon continue de α à β, où α et β sont les angles limites permis par la longueur et la position du muscle. Les deux prochaines sous-sections expliquent le raisonnement qui conduit vers une modulation en largeur d'impulsion.

3.4.2 Problème avec la saturation en longueur

Le premier problème à résoudre est le retard entre le temps où on envoie ou cesse d'envoyer du courant et le temps où le muscle commence à réagir. Ce problème est illustré à la figure 8, qui montre la courbe de contraction/détente sur un cycle. On tente alors de contracter le muscle à partir du repos complet, à partir de la technique utilisée jusqu'ici[10]. On donne alors du courant (1), ce qui fait augmenter progressivement la température jusqu'à ce que le muscle commence à réagir (2). Ce n'est qu'au point 3 que le muscle a fini de se contracter ; on le laisse alors tendu un certain temps (4). Maintenant, on veut le détendre : on arrête donc de donner du courant (5). La température descend donc progressivement jusqu'à ce que le muscle commence à réagir de nouveau (6). Au point 7, le muscle est complètement détendu. On arrête l'expérimentation, et le muscle revient à la température de la pièce, au point 8.

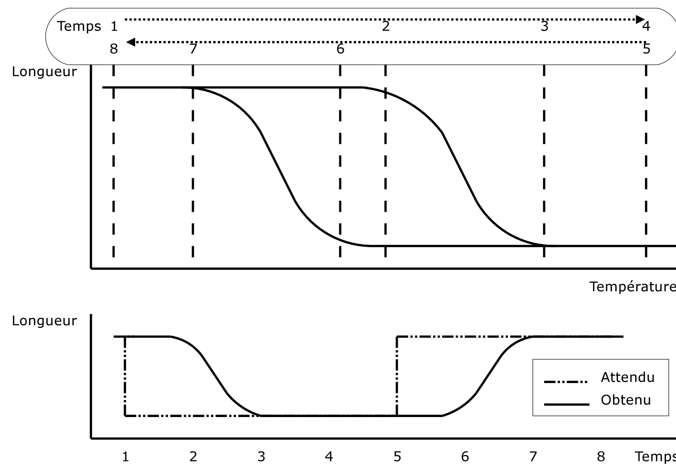


FIG. 8 – Illustration du retard causé par la saturation en longueur

On voit, suivant cette méthode, qu'il faut toujours attendre une période de temps inutile avant d'avoir une réaction à notre commande, soit la période où la température revient à un point qui n'est plus en saturation. On devra donc s'arranger pour ne pas trop s'éloigner en saturation, en restant dans l'entourage de la courbe d'hystérésis.

3.4.3 Contrôle précis de la température du muscle

Le but ici est de contrôler la température du muscle en boucle ouverte, afin de pouvoir éventuellement éviter de se retrouver en saturation de longueur. On désire, pour ce faire, utiliser un micro-

[9]De l'anglais :***Pulse Width Modulation***.

[10]Soit : on veut contracter, on donne du courant ; on veut détendre, on arrête de donner du courant.

contrôleur. La méthode la plus simple serait d'envoyer un voltage analogique au muscle. La puissance ainsi dissipée par le muscle serait directement reliée au voltage appliqué. Par contre, il faudrait pour cela un convertisseur numérique/analogique et un amplificateur pouvant donner un courant de sortie élevé. Il faut aussi penser que plusieurs bits de sortie sont nécessaires, sauf si on fait du multiplexage, ce qui complexifie encore davantage le montage.

Une manière plus efficace serait plutôt de procéder par modulation en largeur d'impulsion (PWM) en utilisant les propriétés naturelles du muscle comme filtre passe-bas (temps requis pour un changement de température, hystérésis sur la longueur du muscle par rapport à la température). De cette manière, on n'a qu'à faire passer un bit de sortie du micro-contrôleur vers un simple *driver* de courant, exactement comme on l'a fait avec le premier circuit *driver* pour le testeur de robustesse[11]. La tension acheminée au muscle est donc soit une constante, soit 0V. C'est cette technique qui sera utilisée lors du développement de ce modèle.

3.4.4 Réalisation du circuit à PWM

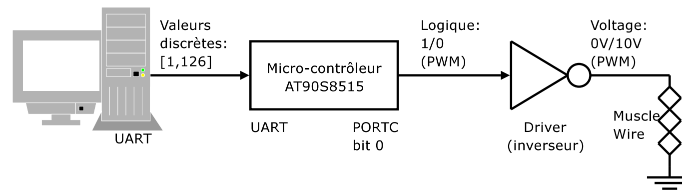


FIG. 9 – Séquence des informations de l'ordinateur au muscle

Pour implémenter un contrôleur dans *Matlab*, on doit assurer une communication entre l'ordinateur et le micro-contrôleur. Cette communication sera assurée par les ports RS232 présents sur l'ordinateur et sur le micro-contrôleur. Afin de simplifier la communication (et parce qu'on n'aurait pas besoin de plus de précision), on se limitera à un octet de 7 bits[12]par commande. On réservera par contre les commandes 0 et 127 pour la communication avec le micro-contrôleur (lors de la prise de données, par exemple - nous en reparlerons plus loin).

C'est donc le micro-contrôleur qui modulera le signal reçu par l'ordinateur en une série de pulsations de largeur variable. Le calcul de la fréquence du signal n'en est pas vraiment un : on peut croire que la vitesse maximale à laquelle le micro-contrôleur pourra opérer sera celle qui donnera les meilleurs résultats. On utilisera donc cette hypothèse, et il n'y aura pas de délai inséré dans la boucle principale du micro-contrôleur.

Le pourcentage de la pulsation active (*duty cycle*) est défini linéairement entre 0% et 100% pour les valeurs entre 0 et 127. Autrement dit, pour une valeur de 126, le duty cycle approche 100% ; la sortie est donc pratiquement toujours active. Notons toutefois qu'à cause de l'inversion de signal causée par le driver, la sortie active du micro-contrôleur correspond à un zéro logique.

D'autre part, la résistance présente dans le circuit de la figure 6(b) a été enlevée. Seulement la résistance du muscle sera maintenant considérée, ce qui nous permet d'utiliser une tension de 10V au lieu de 18V, soit la même tension qui alimente l'interface STK500.

3.4.5 Montage physique

Bien qu'on n'ait besoin que d'un seul muscle, le montage ci-dessous en comporte deux, mis bout à bout. Cette façon de faire comporte l'avantage de limiter la tension nécessaire pour faire passer

[11] Voir la figure 6(b)

[12] *Matlab* se limite à la table ASCII standard pour la communication série. Les valeurs entre 128 et 255 ne sont pas toujours définies et diffèrent d'un système d'exploitation à l'autre.

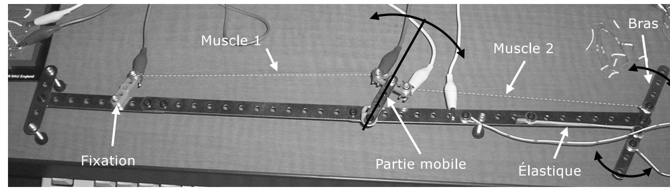


Fig. 10 – Réalisation du bras multi-niveaux

le courant désiré (deux fois le courant recommandé[13]). En effet, en coupant le muscle en deux, la résistance du fil s'en trouve réduite de moitié ; la tension nécessaire diminue donc elle aussi de moitié. On peut donc utiliser en toute sécurité les *drivers* de courant actuels, soient les TC4426, qui ont une tension d'entrée maximale de 18V, avec une tension nécessaire de 10V alors qu'il nous faudrait 20V si le muscle n'était pas séparé.

Dans le cas actuel, chaque muscle est branché sur la sortie d'un driver, pour limiter le courant de sortie (le TC4426 utilisé permet un courant instantané maximal de 1A). Les muscles utilisés ont un diamètre de 100μm et sont installés de façon à ce qu'une contraction maximale du muscle puisse faire passer le bras d'environ $-30°$ à $30°$ par rapport à l'horizon (l'horizon étant normal à l'axe du montage). L'élastique est calibré pour donner environ 150g en tension constante dans les muscles[14].

3.4.6 Programmation du micro-contrôleur

Jusqu'ici, la programmation du micro-contrôleur s'est avérée très facile : seulement quelques assignations de sorties et des délais dans une boucle principale. Pour ce montage, il faut par contre instaurer le PWM et la réception de l'UART. Voici le pseudo-code utilisé pour la programmation du bras multi-niveaux :

Programme principal

```
Initialiser toutes les variables à 0
Accepter les interruptions sur l'UART
Effectuer sans arrêt la boucle principale
```

Boucle principale

```
Si le compteur PWM < niveau actuel
    Contracter le muscle
Sinon
    Détendre le muscle
Fin de la condition

Incrémenter le compteur PWM

Si le compteur PWM > 126
    niveau actuel = prochain niveau
    compteur PWM = 0
Fin de la condition
```

Interruption : Réception de l'UART

```
prochain niveau = donnée reçue
```

Contracter le muscle

```
sortie = 0
```

Détendre le muscle

```
sortie = 1
```

3.4.7 Programmation de *Matlab*

Bien entendu, pour envoyer des données à partir du port série, on doit programmer quelques fonctions sur l'ordinateur. Heureusement, *Matlab* contient un module pour la communication série. Afin

[13]Rappelons-nous les résultats du deuxième circuit *driver* du testeur de robustesse : nous avions remarqué qu'il n'est pas grave de donner deux fois le courant recommandé pour une petite période de temps, car ça ne laisse pas le temps au muscle de surchauffer. C'est ce principe qu'on utilise avec le PWM : chaque pulsation a un courant deux fois plus élevé que le courant recommandé. Or, tant qu'on se situe en-dessous d'un *duty cycle* de 50%, il n'y a aucun risque pour le muscle.

[14]Une caractérisation de l'élastique est nécessaire pour déterminer quelle force celui-ci exerce sur le muscle lorsqu'il est soumis à une certaine déformation. La non-linéarité et la dégradation de l'élastique ne lui confèrent pas une bonne robustesse ; par contre, c'est suffisant pour un prototype.

de simplifier les commandes à envoyer pour l'initialisation et les manipulations de données relatives à l'UART, les fichiers *Matlab* suivants seront utilisés :

initserial.m

```
%Syntax : initserial
%by Felix Chenier
%Started : June 2nd, 2004
%Cette fonction sans paramètre initialise le port série pour pouvoir
%envoyer des données au micro-contrôleur.

function initserial
global com1;
com1 = serial('COM1', 'BaudRate', 115200);
fopen(com1);
```

sendlevel.m

```
%Syntax : sendlevel(level)
%by Felix Chenier
%Started : June 2nd, 2004
%Cette fonction envoie le niveau de PWM désiré (level) au
%micro-contrôleur. level doit être entre 1 et 126 (inclus.)

function sendlevel(level)
global com1;
if ((level>126) | (level<0))
    fprintf('level should be between 1 and 126')
    return
end
fwrite(com1,char(level));
```

closeserial.m

```
%Syntax : closeserial
%by Felix Chenier
%Started : June 2nd, 2004
%Cette fonction ferme le port série ouvert avec initserial.

function closeserial
global com1;
fclose(com1);
```

3.4.8 Expérimentation et résultats

Suivant les programmes présentés dans les deux sections précédentes, on s'attend à ce que lors de la mise sous tension du prototype, le muscle ne reçoive aucun courant. On peut, pour tester le tout, écrire un petit programme *Matlab* comme celui montré ci-bas.

test1.m

```
initserial;                     %Initialiser le port série

sendlevel(15);                  %Déplacer légèrement le bras
tic;
while (toc<4)                   %Attendre 4 secondes
end;

sendlevel(63);                  %Amener le bras à sa position maximale
tic;                            %en toute sécurité
while (toc<4)                   %Attendre 4 secondes
end;

sendlevel(1);                   %Ramener le bras à sa position initiale
tic;
while (toc<4)                   %Attendre 4 secondes
end;

sendlevel(126);                 %Amener le bras à sa position maximale
tic;                            %très rapidement
while (toc<0.2)                 %Attendre 2 dixièmes de seconde
end;

sendlevel(63);                  %Conserver cette position maximale en
tic;                            %toute sécurité
while (toc<4)                   %Attendre 4 secondes
end;

sendlevel(1);                   %Ramener le bras à sa position initiale

closeserial;                    %Fermer le port série
```

Les résultats à une telle simulation ont donné des résultats encourageants. Par contre, deux points importants sont encore à développer :

- On n'a aucune idée de l'inclinaison réelle du bras, seulement une vague idée d'après la commande envoyée. Il nous manque donc un élément très important pour l'asservissement du bras.
- Il n'y a pas de protection face à un abus de courant dans le muscle. En effet, si on envoie la commande 126, par exemple, et qu'on néglige de revenir à une commande plus raisonnable, le muscle ne mettra pas beaucoup de temps à brûler. Or, il est difficile d'implanter un contrôleur tenant compte de cette contrainte.

4 Modèle final

4.1 Capteur d'inclinaison

4.1.1 Type de capteur

Avant de passer au modèle final, on se doit d'introduire un dispositif permettant une rétroaction au système, soit un capteur d'inclinaison. Heureusement, il existe sur le marché des accéléromètres très intéressants limitant au minimum le traitement du signal fourni par ces dispositifs. Le dispositif utilisé ici est le MXD2020ML de la compagnie *MEMSIC*[15]. Il s'agit d'un accéléromètre à deux axes utilisant le spectre de chaleur pour évaluer l'accélération qui lui est appliquée. Dans le cas qui nous intéresse, c'est l'accélération gravitationnelle projetée sur un axe qu'on veut mesurer. Comme la gravité est un vecteur constant vers le bas, la valeur renvoyée correspond au sinus de l'angle entre l'axe de l'accéléromètre et l'horizon. Le tableau 3 illustre bien cette projection.

Accéléromètre (vecteur directeur normalisé)	↑	↗	→	↘	↓
Gravité (vecteur normalisé)	↓	↓	↓	↓	↓
Produit scalaire	-1	$-\frac{\sqrt{2}}{2}$	0	$\frac{\sqrt{2}}{2}$	1

TAB. 3 – Illustration de la lecture de l'inclinaison à partir de la gravité.

[15]Site internet : www.memsic.com

La partie la plus intéressante de ce dispositif est le fait que la valeur renvoyée est directement modulée en PWM, à une fréquence de 100Hz. Il est donc très facile de créer l'interface entre le capteur et le micro-contrôleur, le signal étant déjà numérique. On n'aura donc besoin que d'une entrée.

4.1.2 Terminologie

Pour fin de clarification, le terme *accéléromètre* référera au simple dispositif utilisé pour mesurer l'accélération, tandis que le terme *capteur d'inclinaison* référera à tous le circuit et à la programmation entourant ce dispositif.

FIG. 11 – Accéléromètre MXD2020ML

4.1.3 Branchement de l'accéléromètre

Comme le montre le schéma de la figure 12, l'interface entre l'accéléromètre et le micro-contrôleur est tout simplement une résistance. Pour ce qui est de l'alimentation de l'accéléromètre, le voltage V_{TRG} venant du STK500 (réglé à 4,9V) passe dans le suiveur U1 (un ampli-op μA741) avant de se connecter au circuit d'alimentation suggéré[16].

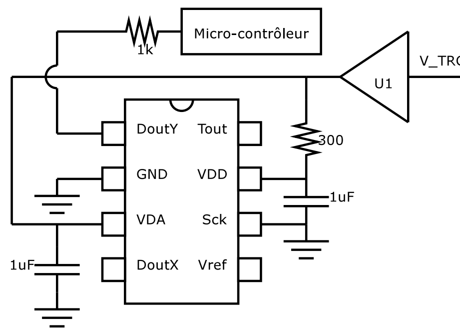


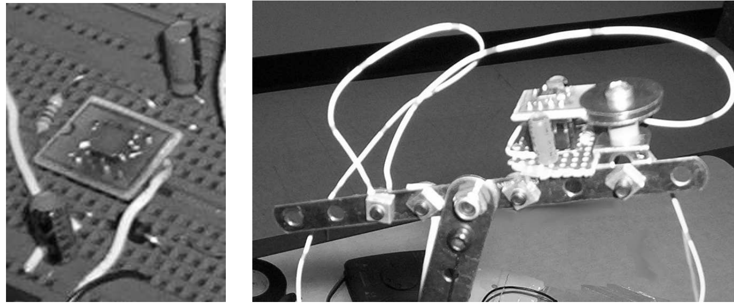
FIG. 12 – Schéma électrique du circuit entourant l'accéléromètre

4.1.4 Montage de l'accéléromètre

L'accéléromètre n'est pas facile à manipuler étant donnée sa petitesse. C'est pourquoi la première étape effectuée lors de son installation a été de le monter sur un adaptateur DIP-8, tel que montré sur la figure 13(a).

L'adaptateur a ensuite été monté sur le circuit de la figure 12, celui-ci ayant été installé sur le bras multi-niveaux (voir la figure 13(b)). Notons que le montage du bras multi-niveaux est passé d'une position horizontale à une position verticale, pour une raison évidente.

[16] L'architecture et les valeurs des résistances et condensateurs de découplages sont bien détaillées sur les feuilles de spécification de l'accéléromètre.



(a) Adaptateur DIP-8 pour l'accéléromètre

(b) Accéléromètre monté sur le bras

FIG. 13 – Montage de l'accéléromètre

4.1.5 Programmation du capteur d'inclinaison

Pour être le plus précis possible, le calcul de l'inclinaison se fait sur une base d'interruptions. Ces interruptions sont gérées par une fonction qui est appelée seulement lorsqu'on désire connaître cette inclinaison. Le but est de compter le nombre d'unités de temps entre un front montant et un front descendant. L'unité utilisée pour la représentation de l'inclinaison sera donc un nombre de coups d'horloge du chronomètre.

Calculer l'inclinaison

```
etat = rien
Nettoyer le tampon d'interruptions sur le capteur d'inclinaison

Initialiser les interruptions sur le front montant
Tant que etat n'est pas [Front montant détecté], attendre

Remettre le chronometre a zero
Initialiser les interruptions sur le front descendant
Tant que etat n'est pas [Front descendant détecté], attendre

inclinaison = chronometre
```

Interruption : événement sur le capteur d'inclinaison

```
Si etat = rien
    etat = Front montant detecte
Sinon, si etat = Front montant detecte
    etat = Front descendant detecte
Fin de la condition

Arreter les interruptions sur le capteur d'inclinaison
```

4.1.6 Programmation de la communication de l'inclinaison

Maintenant que le micro-contrôleur est capable de déterminer sur demande l'inclinaison du bras, il ne reste qu'à le rendre capable de la communiquer à *Matlab*. C'est ici qu'entre en jeu la commande 127, qui était jusqu'alors interdite mais inutilisée. Lorsque *Matlab* voudra savoir l'inclinaison du bras, il n'aura qu'à envoyer cette commande ; le micro-contrôleur déterminera alors cette inclinaison, et la lui enverra par le port série, sous forme de 3 octets de 7 bits (le plus important d'abord).

Ceci implique une modification dans l'interruption sur la réception par l'UART montrée précédemment. Le pseudo-code ci-après montre les quelques modifications et ajouts à faire pour rendre le tout fonctionnel.

Interruption : Réception de l'UART (modification)

```
Si la donnee recue n'est pas 127
    prochain niveau = donnee reçue
Sinon
    Calculer l'inclinaison
    Envoyer l'inclinaison
Fin de la condition
```

Envoyer l'inclinaison

```
Attendre que l'UART soit disponible en emission
Envoyer les 7 bits les plus importants de l'inclinaison
Attendre que l'UART soit disponible en emission
Envoyer les 7 prochains bits de l'inclinaison
Attendre que l'UART soit disponible en emission
Envoyer les 7 bits les moins importants de l'inclinaison
```

Il importe maintenant de créer une fonction dans *Matlab* qui renverra automatiquement l'inclinaison, en passant pour nous par les étapes nécessaires à cette communication. La fonction *gettilt* aura ce mandat.

De plus, comme il arrive de temps à autre qu'on interrompt brusquement une expérimentation à l'aide des touches CTRL+C, il peut arriver que cette interruption se situe exactement dans un processus de requête d'inclinaison. L'UART se retrouve donc emplie de caractères qui ne seront pas nécessairement lus, ce qui entraîne un décalage d'octets lors des prochaines lectures. Il est donc important de vider l'UART après avoir interrompu une expérimentation. Cette fonction sera exercée par la commande *clearserial*.

gettilt.m

```
function tilt=gettilt()
%Syntax : tilt=gettilt
%by Felix Chenier
%Started : June 10th, 2004
%This function without parameters returns the current tilt.

global com1;
global oldtilt;                         %afin d'ignorer des valeurs extremes

fwrite(com1,127);                       %envoyer la commande 127

H1 = fread(com1,1);                     %lire l'inclinaison renvoyee
H0 = fread(com1,1);
L1 = fread(com1,1);

tilt = 256*H1 + 128*H0 + L1;            %reconstruction de l'inclinaison

if ((tilt > 11700)|(tilt < 7000))       %filtrage des valeurs extremes (glitch)
    tilt = oldtilt;
end

oldtilt = tilt;                         %mise en memoire de la nouvelle incl.
```

clearserial.m

```
function clearserial()
%Syntax : clearserial
%by Felix Chenier
%Started : June 14th, 2004
%This function without parameters empties the serial port from outsync data.
%This could happen when we interrupt (CTRL-C) a "gettilt" command.

global com1;

while (1)
    fprintf('Clearing 1 entry... Press CTRL-C if this line is not duplicating.\n');
    fread(com1,1);
end
```

4.2 Protecteur contre la surchauffe

4.2.1 Principe

On a vu qu'avec le bras multi-niveaux, aucune protection n'était offerte quant à une trop longue exposition à un courant plus élevé que le courant recommandé. On va donc créer cette protection à l'aide d'un calcul d'intégrale discrète. Il suffit simplement d'évaluer à tout moment l'intégrale sur le

niveau demandé, à partir d'un certain nombre d'itérations passées, et ce jusqu'au moment actuel. Si cette intégrale dépasse une valeur seuil (ici le niveau sécuritaire), on tronque le niveau demandé pour le remplacer par le niveau sécuritaire. La figure 14 illustre bien cette procédure. Notons que le niveau sécuritaire est généralement assez élevé pour soutenir le bras à sa position maximale. Les performances du bras ne se trouvent donc pas altérées par l'ajout de cette protection.

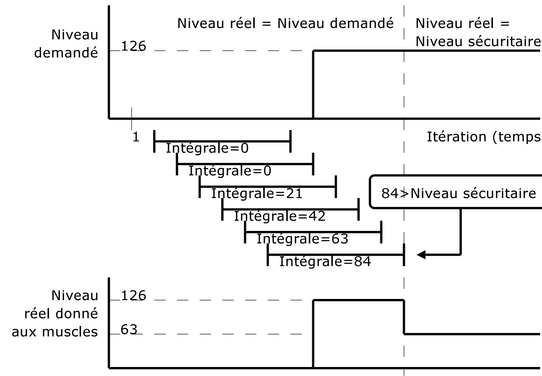
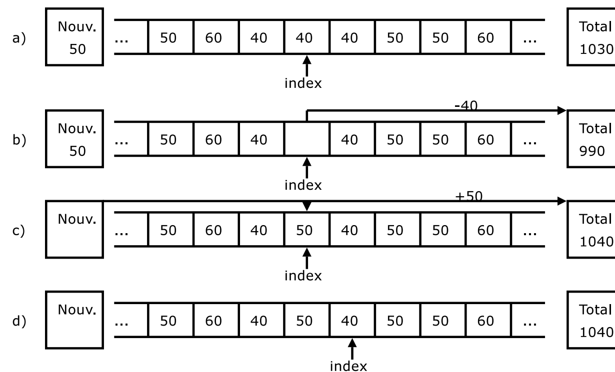


FIG. 14 – Calcul d'une intégrale discrète pour protéger les muscles

Comme cette procédure doit être implémentée dans un micro-contrôleur, on doit donc créer un tableau qui contiendra tous les niveaux demandés situés dans l'intervalle de l'intégrale. On doit aussi, pour conserver une bonne vitesse de calcul, conserver le total de l'intégrale en mémoire. La figure 15 illustre très bien la méthode utilisée pour la mise à jour de l'intégrale.



a) On désire enlever la valeur le plus vieille (désignée par l'index) et la remplacer par la nouvelle valeur, soit 50. b) On enlève du total la valeur pointée par l'index. c) On ajoute au total la nouvelle valeur, puis on assigne cette valeur à celle pointée par l'index. d) On incrémente l'index. Si l'index se retrouve à la limite extrême du tableau, on le réassigne à 0 afin de compléter un cycle.

FIG. 15 – Méthode de mise à jour de l'intégrale

4.2.2 Programmation du protecteur

La condition sur le fait de contracter on non le muscle sera implémentée directement dans la fonction *Contracter le muscle*. Ainsi, si on appelle cette fonction et qu'elle ne retourne rien, on sait que le muscle a bien été contracté. Si, par contre, elle retourne -1, c'est que le muscle n'a pas été contracté. On lui

passera aussi en paramètre une valeur booléeenne signifiant si elle doit forcer le muscle à se contracter tout de même, ou bien simplement laisser tomber. D'autre part, on doit ajouter le calcul de l'intégrale, et changer quelque peu la boucle principale.

Contracter le muscle (modification)

```
Si l'integrale nous le permet, ou bien si on contracte de force
    Sortie = 0
    Bit de protection = 0
    Retourner 0
Sinon
    Sortie = 1
    Bit de protection = 1
    Retourner -1
Fin de la condition
```

Recalculer l'intégrale

```
Enlever du total la valeur dans le tableau correspondant a l'index
Ajouter au total la nouvelle valeur
Remplacer la valeur dans le tableau par la nouvelle valeur
Incrementer l'index
Si l'index n'est plus dans le tableau, index = 0
```

Boucle principale (modification)

```
Incrementer le compteur d'enregistrement
Si le compteur d'enregistrement > Valeur a laquelle on doit enregistrer
    Recalculer l'integrale
    compteur d'enregistrement = 0
Fin de la condition

Si le compteur PWM < niveau actuel
    Tenter de contracter le muscle sans forcer
    Si ca na pas fonctionne et que PWM < niveau securitaire
        Forcer la contraction du muscle
    Fin de la condition
Sinon
    Detendre le muscle
Fin de la condition

Incrementer le compteur PWM
Si le compteur PWM > 126
    niveau actuel = prochain niveau
    compteur PWM = 0
Fin de la condition

Incrementer le compteur d'inclinaison
Si le compteur d'inclinaison > Valeur a laquelle on doit calculer l'inclinaison
    Calculer l'inclinaison
    compteur d'inclinaison = 0
Fin de la condition
```

4.2.3 Communication du bit de protection

Afin d'asservir efficacement un système aussi non-linéaire, on aura idéalement besoin du bit de protection comme variable d'état du système. Pour simplifier ceci, on enverra cette information en même temps qu'on envoie les informations sur l'inclinaison. On modifiera donc la fonction *sendtilt* dans le micro-contrôleur pour envoyer un dernier caractère représentant ce bit (il aura la valeur de 0 ou 1 dans la table ASCII), et on modifiera la fonction *gettilt* dans *Matlab* pour ajouter une lecture de l'UART après celles de l'inclinaison.

4.3 Schéma-bloc du modèle final

On peut maintenant générer le schéma-bloc avec lequel on travaillera dorénavant pour l'asservissement. Ce schéma est illustré à la figure 16.

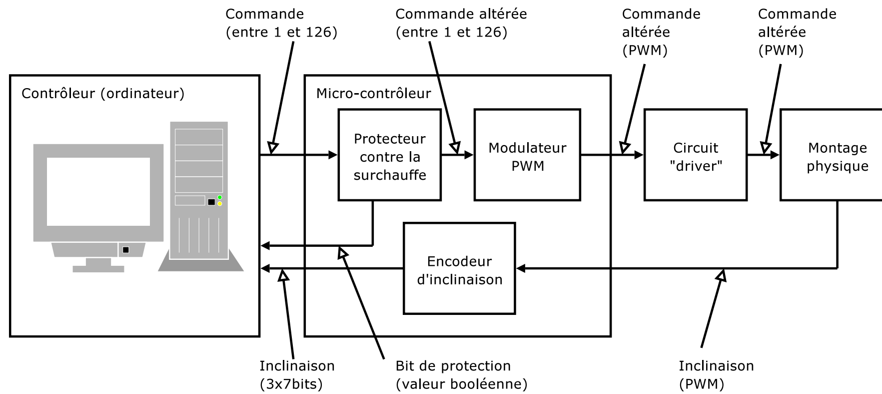


FIG. 16 – Schéma-bloc représentant le système final à asservir.

Deuxième partie
Asservissement du système

L'asservissement de la longueur des muscles est un sujet très vaste qui n'a été ici qu'effleuré. Ainsi, cette partie est plutôt courte : l'étude se dirigeait vers un asservissement avec réseaux de neurones, mais n'a pas vraiment abouti à cause de l'énorme travail à faire dans ce domaine.

5 Type d'asservissement

5.1 Caractéristiques du système

Maintenant qu'on a un prototype fonctionnel capable de communiquer avec *Matlab*, il est bon de se rappeler de ses principales caractéristiques et du type d'asservissement qu'on veut lui implanter :

1 Le système est non-linéaire ;

2 Il possède deux entrées, soient l'inclinaison du bras et le bit de protection contre la surchauffe ;

3 Il possède une sortie, soit le niveau correspondant au *duty cycle* qu'on veut envoyer ;

4 On veut lui implanter un asservissement en position, soit l'obtention d'un angle désiré le plus rapidement possible et de la manière la plus robuste (bonne précision face aux imprévus).

Pour ce faire, on peut penser aux méthodes traditionnelles, comme par exemple le classique contrôleur PID, ou bien aux méthodes plus récentes, comme les réseaux de neurones.

6 Contrôleur PID

Le traditionnel contrôleur PID fonctionne plutôt bien sur les fils contractibles contrôlés par un signal PWM. En effet, on tente de rester dans la partie plus linéaire du système, donc il est faisable de contrôler ce système par un contrôleur linéaire, en autant que la force exercée sur le muscle reste constante le temps de la stabilisation.

On doit aussi faire attention à ne pas trop monter la valeur du gain dérivé, car ceci a tendance à faire osciller l'état du protecteur contre la surchauffe, ce qui rend le système très non-linéaire. Par contre, c'est un contrôleur très simple qui est assez limité, mais qui fonctionne toute de même.

Il n'a pas été jugé utile d'écrire l'implémentation *Matlab* du contrôleur, vu la nature très commune de ce contrôleur.

7 Principe de la méthode *p-delta* sur un perceptron parallèle

Malgré les résultats concluants du contrôleur linéaire, on peut s'imaginer qu'un contrôleur non-linéaire donnerait de meilleurs résultats. Pour implémenter ce contrôleur, un réseau de neurones utilisant l'algorithme *p-delta* a été utilisé. Les prochaines sections résument brièvement le principe d'un perceptron parallèle et de la méthode *pdelta*. Notons que le principe entier est très bien expliqué dans l'article de P. Auer, H. M. Burgsteiner et W. Maass. : *The p-Delta Learning Rule for Parallel Perceptrons* (2).

7.1 Perceptron

Un perceptron est une cellule simple ayant plusieurs entrées et une sortie. Cette sortie est directement reliée aux entrées par la formule

$$o = \begin{cases} 1 & \text{si } \vec{w} \cdot \vec{z} \geq 0 \\ -1 & \text{si } \vec{w} \cdot \vec{z} < 0 \end{cases}$$

où o est la sortie et $\vec{w}$ est un vecteur ligne des poids respectifs pour les entrées définies dans le vecteur colonne $\vec{z}$. Il est courant de définir un seuil variable différent de 0. Ceci peut être fait sans changer le principe du perceptron en ajoutant une entrée z_d et en lui assignant la valeur constante de -1. Ainsi, le poids w_d devient en fait le seuil variable.

7.2 Perceptron parallèle

Un perceptron parallèle est une série de perceptrons partageant les mêmes entrées z_i. La sortie du perceptron parallèle est reliée à la somme des sorties des perceptrons le composant par une fonction limitante $s(x)$. La figure 17 illustre bien cette structure.

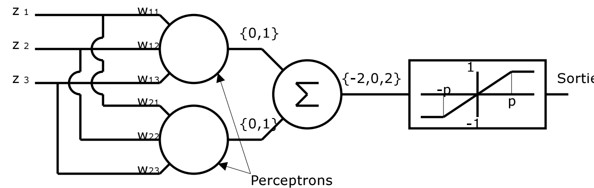


FIG. 17 – Fonctionnement d'un perceptron parallèle

Suivant cette logique, on obtient la sortie du perceptron parallèle :

$$\text{Sortie} = s\left(\sum_{i=1}^{n} \vec{w_i} \cdot \vec{z_i}\right) \quad \text{où} \quad s(x) = \begin{cases} 1 & \text{si } x > p \\ x/p & \text{si } -p \leq x \leq p \\ -1 & \text{si } x < p \end{cases}$$

et n est le nombre de perceptrons.

On voit bien que la sortie ne sera jamais continue, à cause de la valeur fixée à ± 1 pour les différents perceptrons à l'intérieur du perceptron parallèle. Par contre, plus on augmente n, plus on augmente la précision possible en sortie (par contre, il faut plus d'essais pour entraîner plus de neurones). Il ne faut pas non plus oublier que notre système est plutôt lent à réagir. Comme le réseau de neurones agira en temps réel, il n'y aura pas de différence notable entre une sortie qui oscille rapidement entre deux valeurs proches l'une et l'autre, et une sortie qui équivaut exactement à la moyenne de ces deux valeurs.

7.3 Algorithme d'apprentissage *p-delta*

La méthode d'apprentissage *p-delta* ressemble beaucoup aux méthodes d'apprentissage traditionnelles, en ce fait qu'une erreur est mesurée entre la sortie obtenue et la sortie désirée, et que cette erreur est traitée pour modifier certains poids de certains perceptrons. Par contre, comme le perceptron parallèle ne comporte qu'un étage de neurones, il n'y a pas lieu de faire une rétropropagation du gradient, ce qui sauve bien des calculs.

L'algorithme d'apprentissage est donc assez simple, d'où l'intérêt de cette méthode. En résumé, cet algorithme se résume ainsi :

1 Une entrée au hasard est assignée à l'entrée du perceptron parallèle ;

2 La sortie correspondante est calculée (en fonction des poids assignés aux entrées des perceptrons) ;

3 On calcule la différence entre la valeur attendue pour cette entrée et la valeur obtenue ;

4 Si la valeur obtenue est plus grande que celle attendue plus l'erreur acceptable, on corrige les poids de chaque perceptron individuel dont la sortie est positive (donc qui apporte une contribution au dépassement de la sortie) ;

5 Si, au contraire, la valeur obtenue est plus faible que celle attendue moins l'erreur acceptable, on corrige les poids de chaque perceptron individuel dont la sortie est négative ;

6 De plus, si le produit scalaire d'un perceptron individuel est près de zéro, on modifie ses poids afin d'avoir un $+1$ ou un -1 robuste. Cette condition est définie par un paramètre nommé la marge ;

7 On recommence pour une autre entrée.

L'algorithme complet incluant le détail des calculs à effectuer est présenté ci-après. Tout d'abord, voici une liste des variables utilisées dans cet algorithme :

Variables et paramètres

```
w = Matrice n par d représentant les poids --> w(i,j) = Poids pour la jième entrée du ième perceptron
z = Matrice colonne de dimension d représentant les entrée. Note : z(d) = -1
o = Matrice ligne de dimension n représentant les sorties pour chaque perceptron.
delta = Matrice n par d représentant les deltas pour chaque poids de chaque perceptron.
output = Sortie du perceptron parallèle
desired = Sortie désirée
error = Erreur qu'on veut bien tolérer              (paramètre)
d = dimension du système + 1                        (paramètre)
n = nombre de perceptrons par perceptron parallèle  (paramètre)
p = paramètre de la fonction limitante              (paramètre)
eta = constante d'apprentissage                     (paramètre)
mu,gamma = paramètres de marge                      (paramètre)
```

Voici maintenant l'algorithme détaillé, sans correction automatique de la marge[17].

Départ de l'algorithme

```
Générer les poids au hasard pour chaque entrée de chaque perceptron
Boucle principale
    Réinitialiser le delta
        Pour tous les échantillons sampleindex d'une batch
            Prendre un échantillon d'entrée au hasard
            Calculer les sorties pour chaque perceptron du perceptron parallèle
            Calculer la sortie totale
            Calculer la sortie désirée et l'assigner à [desired]
            Ajuster le delta pour chaque perceptron du perceptron parallèle
        Fin de la batch
    Ajuster les poids en fonction des deltas calculés
Fin de la boucle principale
```

Générer les poids au hasard pour chaque entrée de chaque perceptron

```
Pour tout i dans n
    Pour tout j dans d
        w(i,j) = Petite valeur non-nulle choisie au hasard (dans l'ordre de 0.1)
    Fin de la boucle
Fin de la boucle
```

[17]Une méthode de correction automatique de la marge et du rythme d'apprentissage peut être utilisée pour rendre plus efficace une implémentation pratique de cet algorithme. Cette méthode ne sera pas utilisée ici.

Réinitialiser le delta

```
delta = matrice nulle, dimension n par d
```

Prendre un échantillon d'entrée au hasard

```
z = [rand rand ... rand 1]', dimension d avec rand dans [-1,1]
```

Calculer les sorties pour chaque perceptron du perceptron parallèle

```
Pour tout i dans n
    Si le vecteur ième ligne de w multiplié scalairement avec z >= 0
        o(i) = 1
    Autrement
        o(i) = -1
    Fin de la condition
Fin de la boucle
```

Calculer la sortie totale

```
Si la somme sur i des o(i) > p
    output = 1
Sinon, si la somme sur i des o(i) < -p
    output = 1
Sinon
    output = somme sur i des o(i), divisée par p
Fin de la condition
```

Calculer les deltas pour chaque perceptron du perceptron parallèle

```
Pour tout i dans n
    Si (output > desired + error) et (o(i) >= 0)
        delta(i) = delta(i) - z'
    Sinon, si (output < desired - error) et (o(i) < 0)
        delta(i) = delta(i) +z'
    Sinon, si (output <= desired + error) et (0 <= o(i) < gamma)
        delta(i) = delta(i) + mu * (+z')
    Sinon, si (output >= desired - error) et (-gamma < o(i) < 0)
        delta(i) =delta(i) + mu * (-z')
    Sinon
        Ne pas modifier delta(i)
    Fin de la condition
Fin de la boucle
```

Ajuster les poids en fonction des deltas calculés

```
Pour tout i dans n
    w(i) = w(i) + eta * delta(i) --> Ne pas oublier que w(i) et delta(i) sont des vecteurs ligne
    Normaliser w(i)
Fin de la boucle
```

7.4 Résultats préliminaires sur *Matlab*

L'algorithme précédent a été programmé sur *Matlab*, et a donné le résultat illustré à la figure 18 pour les paramètres suivants :

```
%Nombre d'itérations
nbiter = 400;
%Nombre d'échantillons par batch
nbsamples = 5;
%Nombre de perceptrons dans le perceptron parallèle
n = 8;
%Dimension du système (ajouter 1 pour le seuil)
d = 3;
%Rythme d'apprentissage
eta = 0.05;
%Paramètres de marge
gamma = 0.05;
mu = 1;
%Erreur tolérable
error = 1.5/n;
%Calcul de p
p = n-1;
```

À la lumière de ce résultat, on peut assumer que l'algorithme fonctionne plutôt bien. Il suffit maintenant de trouver un moyen de l'implanter sur notre système.

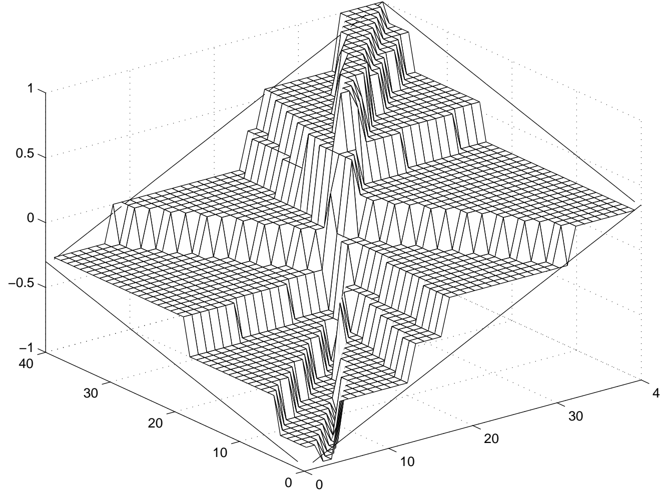


FIG. 18 – Approximation d'un plan par la méthode *p-delta*

8 Implantation de la méthode *p-delta* sur le système

8.1 Principe

Pour un premier et seul essai, on tentera une manière simple. Premièrement, Les entrées n'auront aucun décalage dans le temps. C'est-à-dire qu'il n'y aura que deux entrées, soit la commande et l'erreur proportionnelle instantanée. Les anciennes commandes et les anciennes erreurs proportionnelles (par exemple, la commande à l'itération $n-1$ où n est l'itération courante) ne seront pas incluses.

La valeur ϵ utilisée pour l'apprentissage sera tout simplement l'erreur proportionnelle après deux secondes. Ainsi, on laisse deux secondes au système pour se stabiliser, et on s'arrange pour qu'après ces deux secondes, l'erreur proportionnelle soit rendue nulle.

Le tableau 4 résume les entrées et sorties du réseau de neurones pour cette situation.

Entrées	Sortie	Erreur (ϵ)
Commande Erreur proportionnelle	Niveau à envoyer au micro-contrôleur	Erreur proportionnelle

TAB. 4 – Entrées et sorties du réseau de neurones

Afin de maximiser l'efficacité du réseau de neurones, les données traitées sont situées dans l'intervalle [−1,1]. Un angle minimal a donc une commande d'entrée de -1 et un angle maximal, une commande de 1. L'erreur proportionnelle est tout simplement l'angle obtenu moins l'angle désiré (chacun des deux angles est représenté dans l'intervalle [−1,1]).

8.2 Résultats

Les résultats ont été décevants sur toute la ligne. Premièrement, le système ne semble jamais se stabiliser. Une raison évidente est l'absence d'un contrôle contre l'oscillation du système. En effet, après deux secondes, le système peut être complètement instable, mais si (par hasard) à l'instant où la mesure de l'erreur proportionnelle est prise, cette erreur est nulle, le réseau ne va pas modifier ses poids, croyant avoir atteint un bon résultat. Le système peut donc ne jamais se stabiliser, même en augmentant le nombre d'échantillons.

Une solution possible serait d'ajouter l'intégrale de l'erreur proportionnelle entre le début d'un échantillon et la fin de celui-ci. Ainsi, si le système a passé son temps à osciller, l'erreur proportionnelle pourrait être nulle, mais l'erreur totale serait très grande.

Voyant à quel point cette avenue demande énormément de temps et de patience, il a été cru bon de laisser tomber l'asservissement par réseaux de neurones en faveur de l'utilisation plus poussée des fils contractibles. Toutefois, l'asservissement par réseaux de neurones reste une bonne idée, dans la mesure où une étude consacrée entièrement à ce sujet est faite.

Troisième partie
Fabrication du robot

Les éléments de base ayant été vus lors de la première partie, on peut maintenant débuter la conception d'un robot utilisant ces techniques. Notons qu'il n'y aura pas de concepts totalement nouveaux dans cette partie, seulement des adaptations et généralisations de ce qui a déjà été discuté précédemment. Nous noterons aussi qu'il n'y a pas d'asservissement, les résultats de la deuxième partie n'ayant pas été suffisamment satisfaisants.

9 Modèle physique

9.1 Principe de motion

Vu le mouvement spécifique des fils contractibles, il est beaucoup plus intéressant de faire un robot marcheur qu'un robot rouleur. Toutefois, pour des raisons de simplicité, le robot doit nécessiter un nombre minimal de muscles (le circuit *driver* se complexifie[18]et doit se répéter sur chaque muscle).

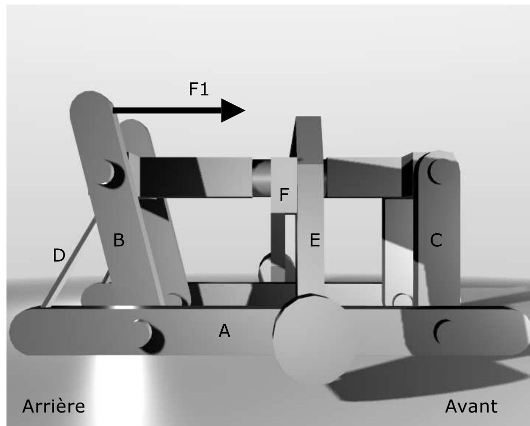
L'idée retenue pour la conception du robot est illustrée à la figure 19.

Pour faire un pas, le robot doit d'abord mettre son poids sur la jambe qu'il veut utiliser et lever l'autre. Ceci est accompli à l'aide de deux muscles agissant respectivement sur les parties (E) et (F). Ainsi, si le robot veut faire un pas avec la jambe gauche, il n'a qu'à tirer selon la force F2 pour abaisser la partie droite de (E) (figure 19(b)). La jambe droite se retrouve donc dans les airs.

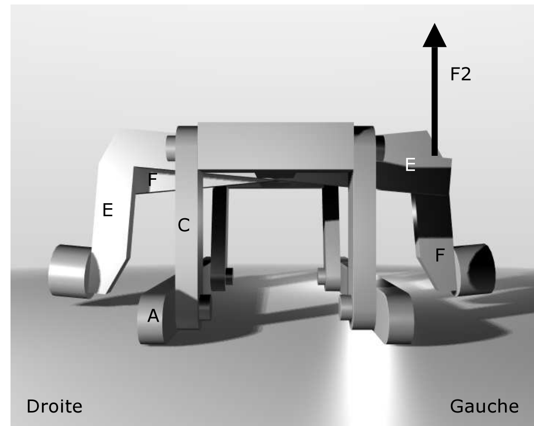
Ensuite, pour achever le mouvement, il doit reculer son pied. Ce mouvement est accompi en tirant selon la force F1. Ainsi, si la jambe gauche est dans les airs, la force F1 crée un moment dans (B), ce qui entraîne (A) vers l'arrière. Le robot a donc fait un pas vers l'avant.

Évidemment, il doit avancer son pied de nouveau pour pouvoir le reculer ensuite. Son pied sera avancé lorsque celui-ci sera dans les airs, soit lorsque le muscle agissant sur (E) sera tendu. Par exemple, si le pied droit est levé et reculé (une force est donc exercée selon F1 et F2), on n'a qu'à relâcher la force F1 pour que l'élastique (D) remette le pied à sa position d'origine. On procède alors au recul de l'autre pied, on relâche la force F2, et ainsi de suite.

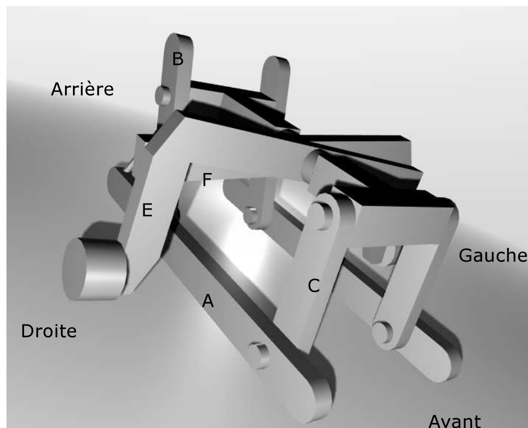
[18]Nous le verrons dans la section 10.1



(a) Vue de côté



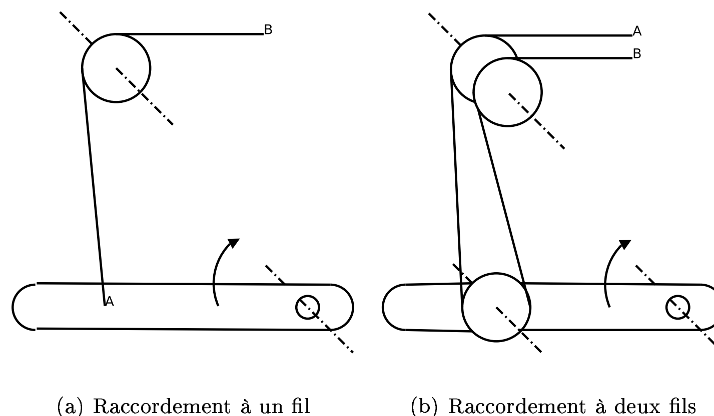
(b) Vue de face



(c) Vue en angle

(d) Photo du système de motion

FIG. 19 – Modélisation 3D du principe de motion du robot



(a) Raccordement à un fil (b) Raccordement à deux fils

FIG. 20 – Raccordement des membres à l'aide de deux fils par articulation

9.2 Dimensions

Si les rapports entre les différentes dimensions sont plutôt fidèles à la figure 19(d), on s'attend à avoir des résultats intéressants avec un déplacement de 1,5cm pour la force F1 et d'environ 1,2cm pour la force F2 (la longueur d'un pied est d'environ 15cm). On a donc besoin, pour F1, d'un muscle d'une longueur de $1{,}5 \div 0{,}03 = 50$cm et pour F2, une longueur de $1{,}2 \div 0{,}03 = 40$cm.

Par contre, comme la structure est relativement lourde, on sera obligé de mettre deux fils en parallèle pour chaque muscle du robot, afin d'avoir assez de force pour que le robot avance sans danger. Il n'est toutefois pas une bonne idée de simplement fixer les fils contractibles un à côté de l'autre, car on ne peut pas assurer que les deux fils forceront réellement ensemble : si un fil est légèrement plus long qu'un autre au repos, celui-ci ne forcera pas autant. L'idée est donc d'utiliser un seul muscle deux fois plus long, et de fixer celui-ci à une poulie plutôt que directement sur le membre à bouger. Les forces s'équilibreront donc d'elles-mêmes lors de la tension. La figure 20 illustre ce principe. Les connexions électriques se font sur les points A et B.

9.3 Montage

Selon les récentes conclusions en matière de longueur et disposition des fils contractibles, on peut arriver à un modèle semblable à celui présenté à la figure 21. Ici, un intense agencement de poulies est situé à l'avant du robot pour permettre la longueur nécessaire de fils. Par exemple, pour la force F1 de la figure 19(a), un fil part de l'arrière du robot, va jusqu'à l'avant, revient vers la jambe à actionner (une poulie est installée à la base de la force F1), et retourne à l'arrière du robot en repassant par l'avant. Pour ce qui est de la force F2 (figure 19(b)), un fil part du centre du robot (sur le dessus), va jusqu'à l'avant, revient au centre et change d'axe (haut-bas) pour aller vers la partie à déplacer (une poulie est installée à la base de la force F2). Le fil revient vers le haut, change de nouveau d'axe (avant-arrière), retourne à l'avant et revient finalement au centre du robot.

10 Circuits électroniques

10.1 Circuit *driver*

Évidemment, le circuit *driver* ne pourra pas être le même que celui qu'on a utilisé précédemment. En effet, la longueur des fils contractibles atteint 80cm et 100cm, ce qui donne un courant maximum nécessaire d'environ $0{,}36\text{A} \times 150\Omega/\text{m} \times 1\text{m} = 54\text{V}$. À première vue, c'est un très haut voltage comparé

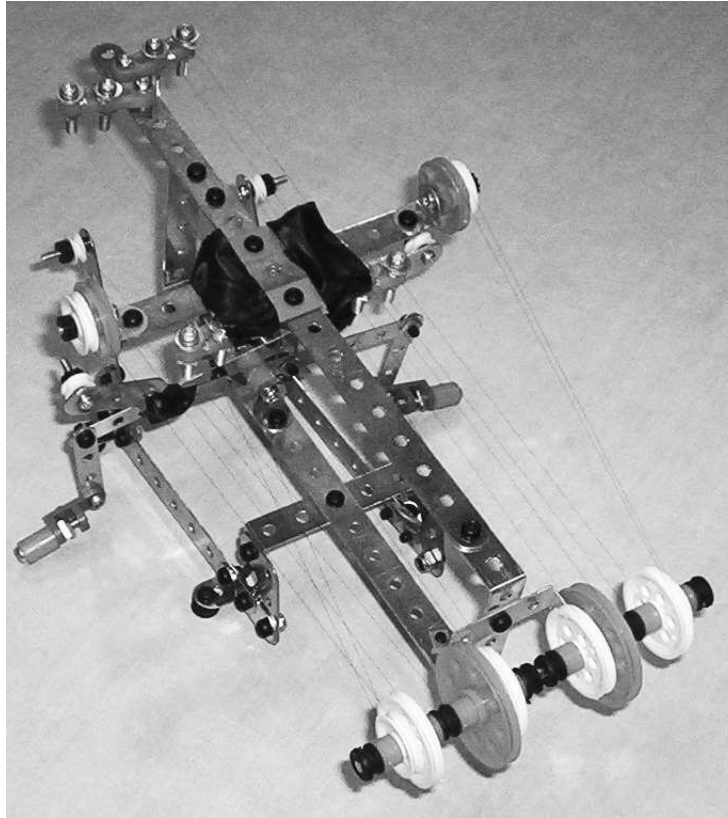


Fig. 21 – Photo du robot

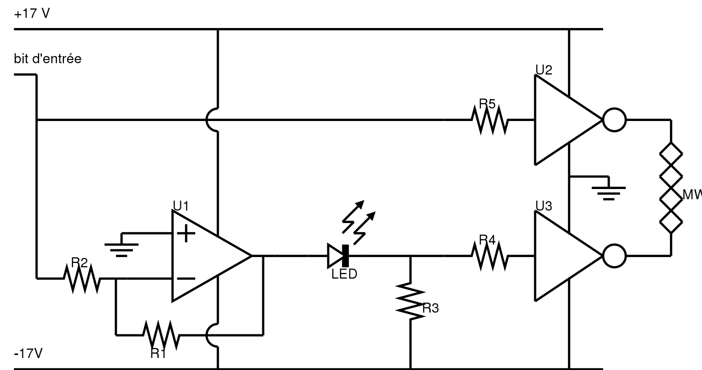


FIG. 22 – Schéma électrique du *driver* de courant utilisé pour les muscles du robot

à ce qui a déjà été traité. Par contre, en pratique, la résistance d'un muscle d'un mètre est plutôt de 120Ω, et on n'est pas obligé d'atteindre exactement 360mA, car il s'agit de toute façon du double du courant recommandé. On se contentera donc d'un robot moins rapide mais plus simple à fabriquer (avec les mêmes pièces que pour les modèles préliminaires).

On sait que la tension maximale d'un inverseur TC4426 est de 18V. Or, si on en met deux en série, on peut atteindre 36V, ce qui nous donne un courant maximal, pour les fils d'un mètre, de $36\text{V} \div 120\Omega =$ 300mA, ce qui est plus qu'acceptable. Par contre, pour éviter d'endommager les composantes ou de les user trop rapidement, on jugera acceptable une tension de ± 17V.

La figure 22 montre ce nouveau *driver*.

10.1.1 Spécification du circuit *driver*

Le concept de ce circuit est d'utiliser deux inverseurs (U2, U3 = TC4426) en série. Si les entrées des inverseurs sont toujours complémentaires (jamais deux ayant la même entrée), la différence de potentiel entre les sorties ne peux que valoir 0 ou 34V. Le problème est la conversion de l'entrée pour l'inverseur U3, qui doit avoir son entrée entre -17 et -16V pour une entrée 0, ou bien entre -15 et -0V[19]pour une entrée 1.

On utilise donc un ampli inverseur dont le gain sera choisi afin d'entrer en saturation négative pour un bit d'entrée 1, et de rester près de 0 pour un bit d'entrée 0. Le gain choisi pour cette fonction est de 4, conduisant aux valeurs R1 = 1kΩ et R2 = 3,9kΩ. Ainsi, selon les spécifications du micro-contrôleur et de l'ampli-op U1 = μA741, la sortie maximale pour un bit d'entrée 0 sera de $-0{,}2 \times -4 = 0{,}8$V et la sortie minimale pour un bit d'entrée 1 sera de $4{,}8 \times -4 = -19{,}2$V (Après compression due à la saturation : $= -15$V.

Or, on constate que les tensions obtenues sont, dans les deux cas, légèrement supérieures aux tensions maximales désirées ($0{,}8 > -0$ et $-15 > -16$). On abaisse donc la sortie de l'ampli-op d'une tension de 1,5V grâce à une diode électroluminescente (LED). La résistance R3 = 10kΩ sert à polariser cette diode, tandis que les résistances R4 et R5 = 1kΩ limitent tout simplement le courant dans l'entrée des inverseurs.

10.2 Montage sur circuit imprimé

Comme on voudra éventuellement rendre le robot le plus autonome possible, on ne doit pas laisser le circuit électronique à l'extérieur du robot. On devra donc le monter sur un circuit imprimé, qui sera posé sur le "dos" du robot.

Les spécifications à retenir pour le circuit imprimé sont les suivantes :

- On doit pouvoir reprogrammer le micro-contrôleur à volonté ;

[19]Il ne faut pas dépasser 0V, sinon on pourrait endommager l'inverseur.

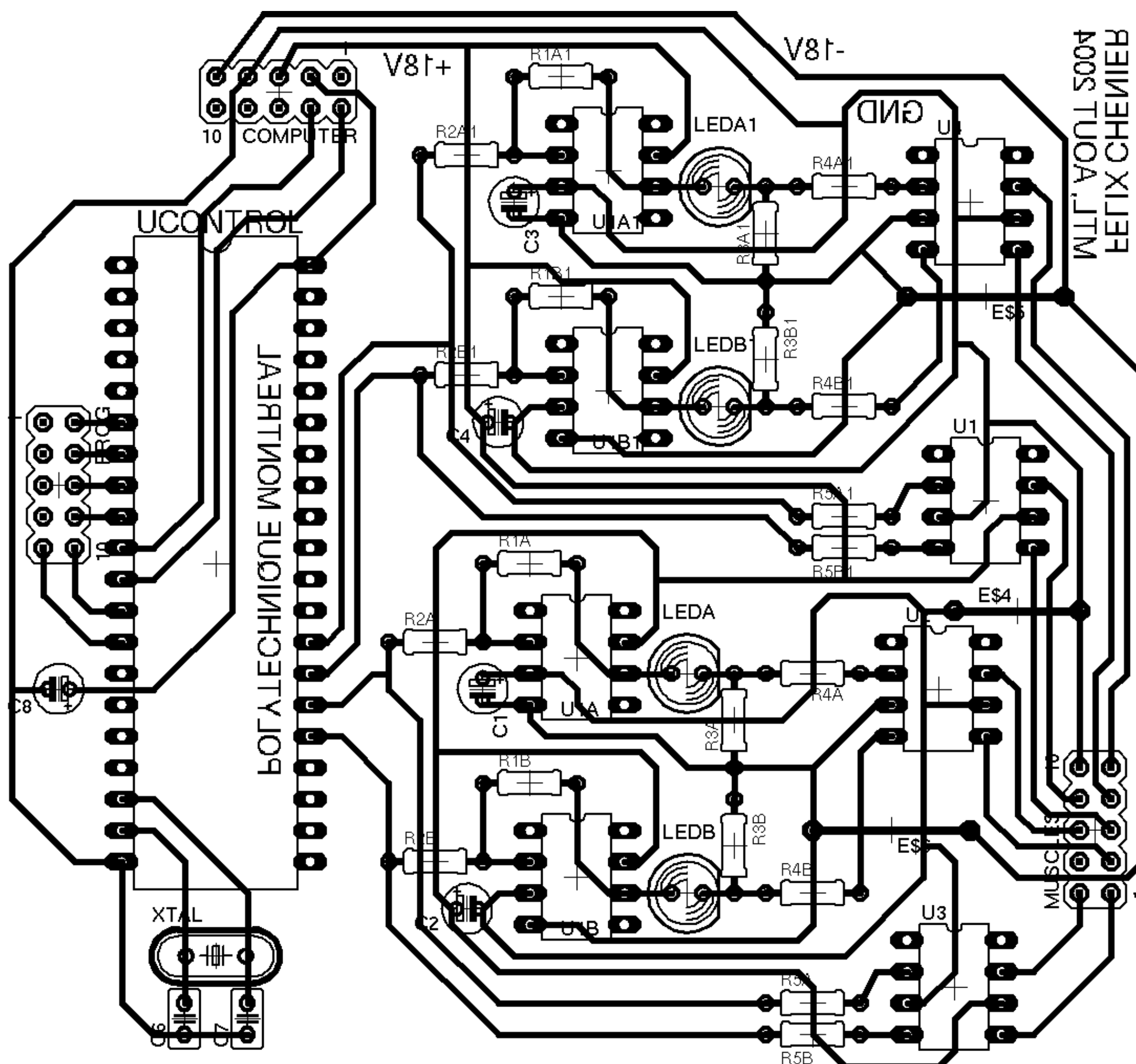


FIG. 23 – Circuit imprimé pour le contrôle du robot

- Le micro-contrôleur doit pouvoir communiquer avec l'extérieur grâce au port série[20] ;
- Le circuit doit être facilement amovible pour fin d'entretien, il doit donc être muni de simples connecteurs faciles à enlever, et non de fils soudés en permanence.

Suivant ces spécifications, on en arrive à l'exemple présenté à la figure 23 (Cet exemple est celui qui sera utilisé pour les manipulations futures). La table 5 identifie les valeurs des différentes composantes du circuit imprimé.

10.3 Circuit de conversion de l'UART

Comme les données provenant du port série de l'ordinateur sont représentées ainsi :

$$\begin{array}{l|l} \text{État logique 1} & -x\text{V} \\ \text{État logique 0} & x\text{V} \end{array} \quad \text{où } x \approx 12$$

[20] Éventuellement, afin de simplifier les choses, les autres circuits sur le robot communiqueront aussi entre eux par leurs ports série respectifs.

Symbole	Valeur	Symbole	Valeur
UCONTROL	AT90S8515	COMPUTER	Connecteur 10 pins
U1A	μA741	MUSCLES	Connecteur 10 pins
U1A1	μA741	PROG	Connecteur 10 pins
U1B	μA741	U1	TC4426
U1B1	μA741	U2	TC4426
LEDA	Led 1,5V	U3	TC4426
LEDB	Led 1,5V	U4	TC4426
LEDA1	Led 1,5V	R1A	4kΩ
LEDB1	Led 1,5V	R1A1	4kΩ
R2A	1kΩ	R1B	4kΩ
R2A1	1kΩ	R1B1	4kΩ
R2B	1kΩ	XTAL	Crystal 3,68MHz
R2B1	1kΩ	C6	4,7pF
Autres résistances	1kΩ	C7	4,7pF
Autres condensateurs	1μF		

TAB. 5 – Spécifications sur les symboles du circuit imprimé

et que le micro-contrôleur représente ces états ainsi :

État logique 1	5V
État logique 0	0V

il nous faut donc un convertisseur de voltage. Une méthode simple serait de tout simplement acheter un convertisseur déjà tout fait. Une autre solution est de faire nous-même ce convertisseur, qui est plutôt simple. C'est la deuxième solution qui a été adoptée ici. Le schéma utilisé est montré à la figure 24. Notons que l'alimentation de 5V est aussi générée par ce circuit à l'aide d'un régulateur de tension, ce qui élimine le besoin de fournir nous-même cette tension supplémentaire.

Ce circuit sera donc placé juste avant le connecteur COMPUTER du circuit imprimé, et l'alimentation de 5V qu'il génère sera passée au micro-contrôleur.

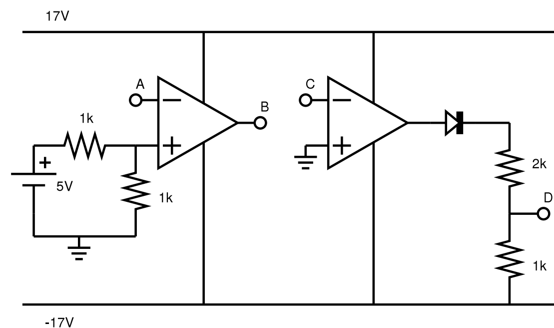


FIG. 24 – Convertisseur UART - 0/5V

Notons que ce circuit fonctionne très bien, toutefois à cause du *slew-rate* relativement faible d'un ampli μA741, on ne peut pas continuer de communiquer aussi rapidement (115200bps). On doit donc baisser la fréquence. Une fréquence de 19600bps est donc choisie. Par contre, comme le robot fonctione sur une base d'interruptions et que le micro-contrôleur s'occupe lui-même du PWM, la différence de vitesse de communication n'est pas notable.

La figure 25 montre le présent circuit monté sur une plaquette de montage commune.

Symbole	Connection	Valeurs possibles
A	TX du micro-contrôleur	0/5V
B	RX de l'ordinateur	−15/15V
C	TX de l'ordinateur	−15/15V
D	RX du micro-contrôleur	0/4,8V

TAB. 6 – Description des symboles de la figure 24

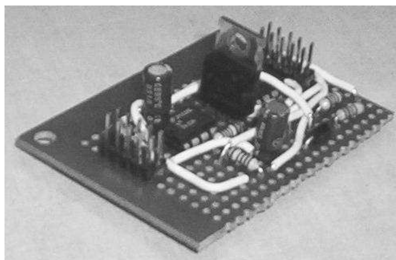


FIG. 25 – Photo du convertisseur UART - 0/5V

10.4 Traitement de l'inclinaison

L'inclinaison n'est plus une donnée très intéressante. Par contre, on peut toujours brancher le circuit déjà réalisé dans les entrées d'interruptions. Ces entrées sont accessibles par le connecteur PROG. Il suffit d'inclure les résistances de 1kΩ servant à limiter le courant et le tour est joué. Bien sûr, comme il est plus intéressant d'avoir l'inclinaison sur deux axes, une modification majeure dans le calcul de l'inclinaison doit être effectuée. Ceci sera vu plus loin, à la section 11.2.

11 Programmation

Pour le bon fonctionnement du robot, la programmation change quelque peu face à ce qui a été fait précédemment. En effet, l'ajout de trois muscles conplexifie un peu les choses, et l'inclinaison se calcule maintenant par rapport à deux axes.

11.1 Communication des informations

Le programme qu'on utilisait jusqu'à maintenant ne recevait d'informations que pour un muscle. Afin d'ajouter d'autres muscles, on doit passer d'une commande d'un octet à une commande de deux octets. La table 7 montre les octets à envoyer et leurs effets respectifs.

Octet 1	Octet 2	Effet
0	(aucun)	Arrêt d'urgence
1 à 4	1 à 127	Assigner le niveau (octet 2) au muscle no.(octet 1)
127	(aucun)	Renvoyer l'inclinaison (dans l'ordre : x, y)
126	1 à 4	Renvoie le bit de protection du muscle no.(octet 2)

TAB. 7 – Commandes reconnues par le programme

La manière utilisée pour y parvenir est tout simplement d'avoir une variable se rappelant du dernier octet reçu. Si on s'arrange pour que cette valeur soit nulle lorsqu'une commande est terminée, on peut toujours savoir si on se trouve dans une commande à deux octets ou une commande à un octet. Par exemple, si on reçoit 126 et que cette variable vaut 1, on doit assigner le niveau 126 au muscle 1, puis remettre la variable à 0. Par contre, si on reçoit 126 et que la variable vaut 0, on sait que le prochain

bit sera le numéro du muscle dont on enverra le bit le protection. On assigne donc la variable à 126 et on continue les opérations normalement jusqu'à la réception du deuxième octet.

11.2 Calcul de l'inclinaison sur deux axes

La première solution qui nous vient à l'esprit pour calculer une inclinaison supplémentaire est tout simplement l'ajout d'une autre fonction qui calculera, après la première, l'inclinaison sur l'autre axe. Cette manière de procéder est très simple, mais n'est pas optimale vis à vis le temps d'exécution du calcul d'inclinaison. Il est beaucoup plus intéressant de calculer les deux inclinaisons en même temps.

L'ancienne fonction permettant de calculer l'inclinaison peut être vue comme une machine à état où l'état suivant n'est déterminé que par l'état actuel, et où l'état change lors d'une interruption. La nouvelle fonction peut aussi être vue comme une machine à états fonctionnant sur la base de deux interruptions. Toutefois, en plus de déterminer quand le système doit passer au prochain état, l'interruption décide quel sera cet état. Le schéma de la figure 26 montre le diagramme d'états servant à calculer ces inclinaisons.

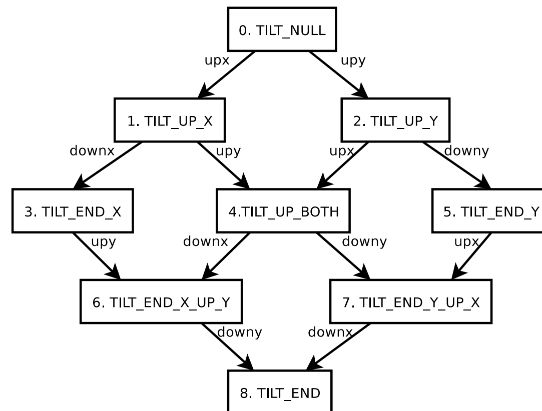


FIG. 26 – Diagramme d'états pour le calcul des deux inclinaisons

Les événements *upx* et *upy* signifient un front montant sur l'entrée d'interruption correspondant à x ou à y, tandis que les événements *downx* et *downy* signifient un fron descendant.

Outre cette différence dans le diagramme d'états, le principe de calcul reste le même. La fonction commence toujours à l'état `TILT_NULL`, où le chronomètre est remis à zéro puis démarré. Lors d'un événement *upx*, par exemple, une valeur de temps initiale est mémorisée pour l'inclinaison en x. Lorsque l'événement *downx* surviendra, on sait qu'il aura fallu le temps du chonomètre moins la valeur de temps initiale pour passer d'un front montant à un front descendant. Le même principe s'applique pour y. Si on observe le diagramme d'états, on remarque que lorsque les inclinaisons selon x et selon y seront calculées, on sera inévitablement rendu à l'état `TILT_END`, c'est-à-dire la fin de la fonction.

12 Résultat final

Afin de mieux voir l'interaction entre les différentes parties du robot, la figure 27 montre un schéma des différents circuits le composant.

La photo de la figure 28 montre le robot lorsque tout est monté. Le circuit principal est sur le dessus, tandis que la circuit de conversion de l'UART se trouve en bas, où le connecteur vers la base est branché.

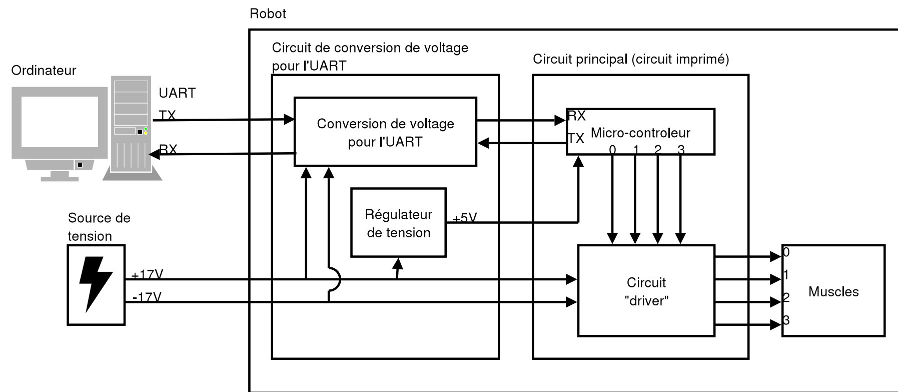


FIG. 27 – Schéma du système du robot

Conclusion et améliorations

Comme on peut le constater, une bonne partie du développement fait lors de ce document ne se retrouve pas dans le robot final. Par exemple, la protection contre la surchauffe a été coupée (voir le code C en annexe) à cause de la différence des niveaux sécuritaires pour une longueur de 80cm et de 100cm : comme les voltages aux bornes des fils sont les mêmes, les courants les traversant sont différents. Il serait donc bien de spécialiser l'algorithme de protection pour tenir compte de ces différences de longueur.

On remarque aussi qu'aucun asservissement n'est pratiqué sur le robot. Il serait intéressant d'utiliser des capteurs divers, par exemple pour déterminer les positions relatives des membres et la position du robot dans l'espace. Ainsi, un asservissement efficace serait possible, et le robot pourrait devenir plus autonome. Il suffit d'un autre micro-contrôleur pour traiter ces informations, qui communiquerait lui-même en série avec le micro-contrôleur actuel.

Bien entendu, ce robot est un premier prototype, et est tout de même loin d'être parfait. En effet, l'amplitude et la précision de ses mouvements restent faibles, et l'ajout de poids cause bien des problèmes. Il y a aussi le problème de l'alimentation : le robot, sous sa forme actuelle, peut utiliser jusqu'à 17W. Ceci empêche l'utilisation d'une batterie, et contraint le robot à traîner un fil.

Ces points restent donc à évaluer et à traiter, dans l'espoir d'obtenir un jour un dispositif fonctionnel articulé par des fils contractibles.

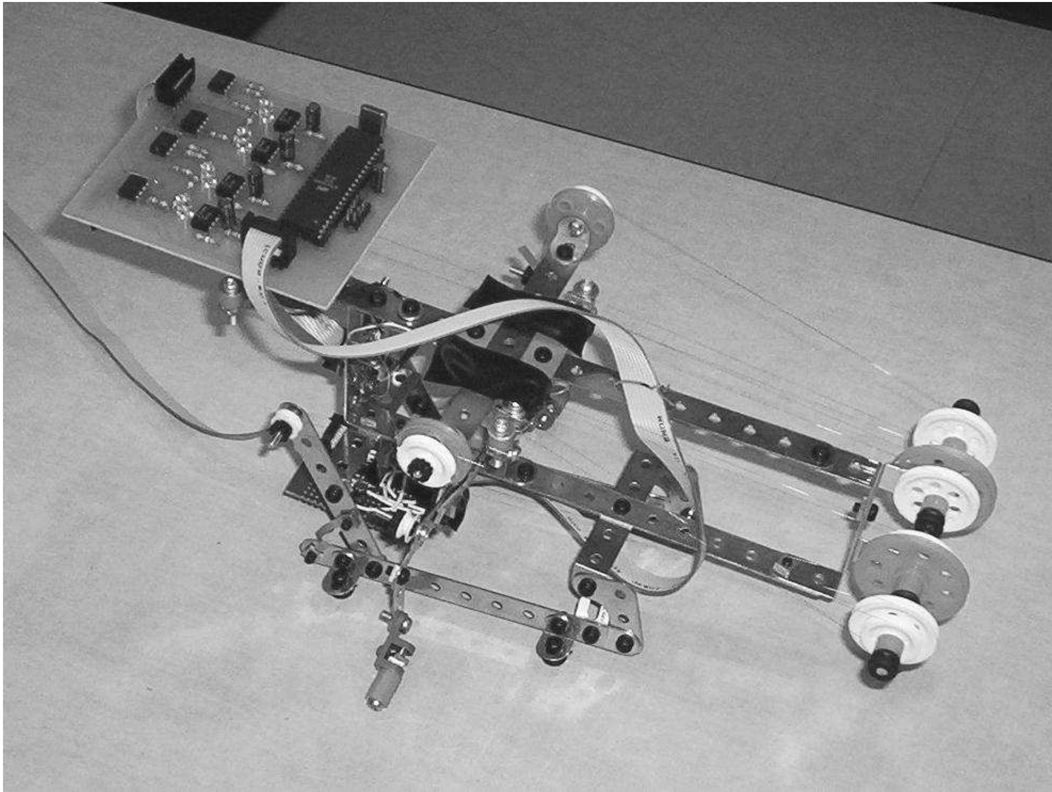


FIG. 28 – Photo de l'aspect final du robot

Annexe

A Notes techniques sur l'utilisation du robot

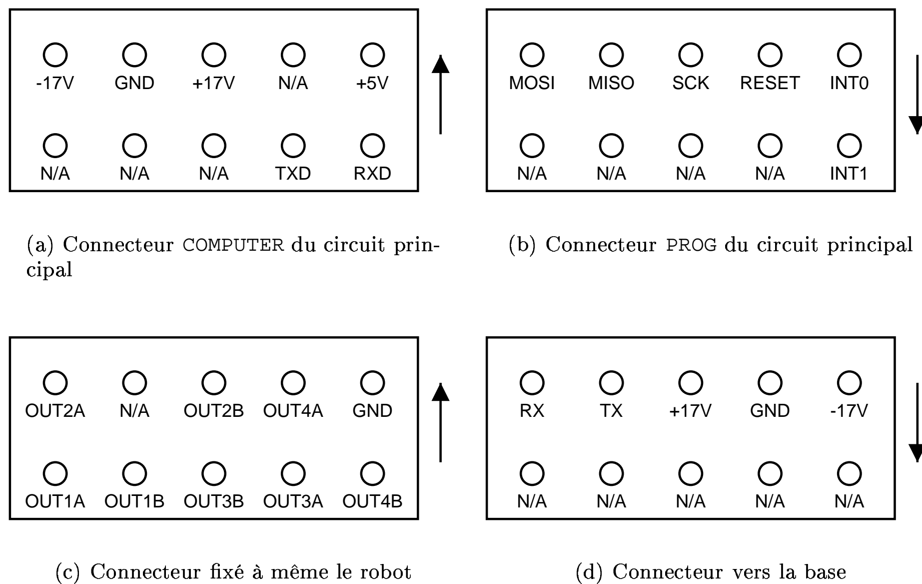
Ces notes viennent préciser des points spécifiques au prototype réalisé pour fin de manipulations ultérieures. Les points les plus importants figurent dans le corps du document.

A.1 Alimentation

Le robot nécessite une tension positive et une tension négative. La tension la plus efficace et la plus sécuritaire est ±17V. À 18V, il ne devrait pas y avoir de problème pour les inverseurs, mais le circuit de conversion de voltage pour l'UART[21]a tendance à donner plus que 5V dans l'entrée du micro-contrôleur, ce qui peut évidemment l'endommager. Il faudrait donc changer ce circuit pour être certain de ne pas endommager le micro-contrôleur. Au-delà de 18V et sans aucune modification du circuit, il est certain qu'une pièce brûlera éventuellement.

La tension minimale dépend beaucoup du circuit de conversion de voltage pour l'UART, car c'est ce circuit qui risque de ne pas envoyer un voltage assez haut pour le micro-contrôleur. Ici encore, une solution serait de modifier ce circuit. Par contre, le seul problème possible en baissant le voltage est une perte de communication avec l'ordinateur. En outre, en-dessous d'un certain seuil, les muscles n'auront pas assez de courant pour se contracter. En dessous de 2,7V, le micro-contrôleur ne fonctionnera tout simplement pas.

[21]Voir la section 10.3



(a) Connecteur COMPUTER du circuit principal

(b) Connecteur PROG du circuit principal

(c) Connecteur fixé à même le robot

(d) Connecteur vers la base

FIG. 29 – Connecteurs

A.2 Connecteurs

La figure 29 montre les différents connecteurs présents sur le robot, la signification de chaque pin et l'orientation du fil (illustrée par la direction de la flèche). Il convient d'indiquer le sens de branchement des connecteurs : En tout temps, le câble est branché sur un circuit avec le fil venant de l'extérieur. Autrement dit, les fils sortent du circuit, on ne doit pas cacher le circuit avec le fil. La flèche pointe donc l'extérieur du circuit.

A.3 Manipulation du robot

On peut constater que des élastiques sont fixés au milieu du robot. Ces élastiques agissent dans la même direction que les muscles, pour contrer quelque peu le poids du robot. Il est donc clair que lorsqu'on désire prendre le robot pour le déplacer, les deux parties servant à lever les jambes du robot vont avoir une nette tendance à vouloir s'abaisser. Or, en s'abaissant, les muscles risquent de quitter les sillons des poulies. On doit donc, pour déplacer le robot sans que les muscles ne décrochent, prendre le robot par ces parties, en tenant l'équilibre du robot par la paume des mains.

Si, par malheur, un muscle s'échappe du sillon d'une poulie, on doit s'arranger pour garder toujours un certain rayon de courbure au muscle. Autrement dit, la meilleure manière pour le replacer est en mettant notre doigt où irait la dernière poulie (ne pas le tenir autrement sans quoi le rayon de courbure risquerait d'être trop petit) et en le ramenant vers cette poulie. Ensuite, on détend la partie actionnée par cette poulie (pour approcher la poulie du muscle), puis on dépose le muscle dans le sillon.

Notons qu'on doit impérativement enlever le courant avant de manipuler le robot. Les muscles sont conducteurs et pratiquement invisibles. De plus, comme la structure est en métal, un simple contact entre un muscle et le métal peut s'avérer dangereux pour le robot et pour l'utilisateur.

B Code en C du micro-contrôleur

```
#define __AVR_AT90S8515__
#define OC1 PD5
```

Nom de la pin	Signification
Figure 29(a)	
-17V	Entrée : Alimentation de -17V
GND	Mise à la terre
+17V	Entrée : Alimentation de +17V
+5V	Entrée : Alimentation de +5V
TXD	Sortie : Pin TXD du micro-contrôleur
RXD	Entrée : Pin RXD du micro-contrôleur
Figure 29(b)	
MOSI	Entrée : Pin MOSI du micro-contrôleur
MISO	Sortie : Pin MISO du micro-contrôleur
SCK	Entrée : Clock pour la programmation ISP
RESET	Entrée : Reset du micro-contrôleur
INT0	Entrée : Interruption 0 du micro-contrôleur
INT1	Entrée : Interruption 1 du micro-contrôleur
Figure 29(c)	
OUT1A	Sortie : Une des deux connexions pour le muscle 1
OUT1B	Sortie : L'autre connexion pour le muscle 1
OUT2A	Sortie : Une des deux connexions pour le muscle 2
...	...
GND	Mise à la terre
Figure 29(d)	
RX	Entrée : À connecter au TX de l'ordinateur
TX	Sortie : À connecter au RX de l'ordinateur
17V	Entrée : Alimentation de +17V
GND	Mise à la terre
-17V	Entrée : Alimentation de -17V

TAB. 8 – Significations des pins des connecteurs de la figure 29

```
#define DDROC DDRD
#define OCR OCR1A
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <avr/io8515.h>

//#define TILTPLUGGED //Uncomment if we plug the tilt sensor

#define bintodec(b7,b6,b5,b4,b3,b2,b1,b0) \\
(b0)+2*(b1)+4*(b2)+8*(b3)+16*(b4)+32*(b5)+64*(b6)+128*(b7)
#define getbit(byte,position) ((byte)>>(position))%2

#define MUSCLEPORT PORTC
#define LEDPORT PORTB
#define HISTORYLENGTH 100 //In what time will it stop with a too high current
#define RECORDRATE 2 //Recording rate for history
#define PROTECTIONTRIGGER 1000 //From wich level should we stop
#define PROTECTIONLEVEL 1000 //To wich level should we come back when we stop

#define TILT_NULL 0
#define TILT_UP_X 1
#define TILT_UP_Y 2
```

Muscle	Fonction
1	Reculer la jambe gauche
2	Reculer la jambe droite
3	Lever la jambe gauche
4	Lever la jambe droite

Tab. 9 – Numéros et fonctions des muscles

```
#define TILT_END_X 3
#define TILT_UP_BOTH 4
#define TILT_END_Y 5
#define TILT_END_X_UP_Y 6
#define TILT_END_Y_UP_X 7
#define TILT_END 8

#define ERRORUNKNOW 0
#define ERRORSHUTDOWN 1
#define ERRORINVALIDCOMMAND 2
#define ERRORCHARTOOBIGTOSEND 3
#define ERRORINVALIDSERIALFIRSTBYTE 4
#define ERRORTILTINTERRUPT 5
#define ERRORUNDEFINEDTILTSTATE 6

#define NBCYCLESTILT 10

#define FORCE 1
#define DONTFORCE 0

#define NBMUSCLES 4

unsigned char muscleport;
volatile unsigned char level[NBMUSCLES]; //PWM
volatile unsigned char nextlevel[NBMUSCLES]; //read by the serial port,
                                             //then will be transmited
unsigned char history[NBMUSCLES][HISTORYLENGTH];
unsigned int historyindex;
unsigned int integraltotal[NBMUSCLES];
volatile unsigned char tiltread;
volatile unsigned char oldtiltread;
volatile unsigned char nexttilt;
unsigned char tilthighx;
unsigned char tiltlowx;
unsigned char tilthighstartx;
unsigned char tiltlowstartx;
unsigned char tilthighy;
unsigned char tiltlowy;
unsigned char tilthighstarty;
unsigned char tiltlowstarty;

volatile unsigned char serialfirstbyte;
unsigned char protectionbit[NBMUSCLES];
unsigned char musclebit[NBMUSCLES];

//Update the integral for the overheat protection
void updateintegral(unsigned char muscle, unsigned char value) {
```

```
  integraltotal[muscle] -= history[muscle][historyindex];
  integraltotal[muscle] += value;
  history[muscle][historyindex] = value;
}

//Send muscles and protection bits to the muscle port
void sendmusclestate() { //This function is not valid if NBMUSCLES != 4
  MUSCLEPORT =
    (musclebit[0] << 4) +
    (musclebit[1] << 5) +
    (musclebit[2] << 6) +
    (musclebit[3] << 7) +
    (protectionbit[0] << 1) +
    (protectionbit[1] << 2) +
    (protectionbit[2] << 3) +
    (protectionbit[3] << 4);
}

//Sets a muscle to on. If force is true, won't even check if the integral
//is low enough. Used for the PWM.
int setmuscleon(int muscle, unsigned char force) {
  //check if we can
  if ((integraltotal[muscle] < PROTECTIONTRIGGER * HISTORYLENGTH)||
      (force==FORCE)) { //we can
    musclebit[muscle] = 0;
    protectionbit[muscle] = 0;
    sendmusclestate();
    return 0;
  } else {
    musclebit[muscle] = 1;
    protectionbit[muscle] = 1;
    sendmusclestate();
    return -1;
  }
}

//Set a muscle to off. Used for the PWM.
void setmuscleoff(int muscle) {
  musclebit[muscle] = 1;
  sendmusclestate();
}

//Simply wait tens tens of second.
void wait(int tens) {

  unsigned char i;
  outp(0, TCNT0);
  outp(bintodec(0,0,0,0,0,1,0,1), TCCR0);
  i = 0;
  while (i<2*tens) {
    if (inp(TCNT0)>=184) { // Clock = 3680 kHz
      outp(0,TCNT0);
      i++;
    }
  }
}
```

```
//Turn off all muscles and stop showing the error type.
void fatalerror(unsigned char error) {
  unsigned char i=0;
  MUSCLEPORT = 255;
  for (;;) {
    if (i==0) i=128; else i=0;
    LEDPORT = 255-i-error;
    wait(1);
  }
}

//Send a byte to the UART
void sendchar(unsigned char tosend) { //Must be between 0 and 127 !
  unsigned char temp=0;
  if (tosend > 127) fatalerror(ERRORCHARTOOBIGTOSEND);
  while (getbit(temp,5)==0) temp = USR; //wait for UART ready
  UDR = tosend;
}

//Send the tilt information to the UART.
void sendtilt() {
  sendchar(tilthighx >> 4);
  sendchar((tilthighx << 4) >> 4);
  sendchar(tiltlowx >> 4);
  sendchar(tilthighy >> 4);
  sendchar((tilthighy << 4) >> 4);
  sendchar(tiltlowy >> 4);
}

//Send the protection bit for the muscle muscle to the UART.
#define sendprotectionbit(muscle) sendchar(protectionbit[(muscle)])

//Interruption on UART Receiving.
SIGNAL(SIG_UART_RECV) {
  unsigned char byteread;
  int i;
  byteread = UDR;

  if (byteread == 0) //panic
    fatalerror(ERRORSHUTDOWN);
  else {
    switch(serialfirstbyte) {
    case 1:
    case 2:
    case 3:
    case 4: { //sendlevel
      nextlevel[serialfirstbyte-1] = byteread;
      serialfirstbyte = 0;
      break;
    }
    case 126: { //send protection bit
      sendprotectionbit(byteread-1);
      serialfirstbyte = 0;
      break;
    }
```

```
    case 0 : {
      switch(byteread) {
      case 1:
      case 2:
      case 3:
      case 4: { //next byte will be the level to send
serialfirstbyte = byteread;
break;
      }
      case 127: { //send tilt
sendtilt();
serialfirstbyte = 0;
break;
      }
      case 126: { //next byte will be the protection bit to send
serialfirstbyte = byteread;
break;
      }
      default : { //invalid command
fatalerror(ERRORINVALIDCOMMAND);
break;
      }
      }
      break;
    }
    default : { //invalid serialfirstbyte
      fatalerror(ERRORINVALIDSERIALFIRSTBYTE);
      break;
    }
    }
  }
}

//Interruption on INT1
SIGNAL(SIG_INTERRUPT1) {
  GIMSK = bintodec(0,0,0,0,0,0,0,0); //Disable external interrupt
  oldtiltread = tiltread;
  switch(tiltread) {
  case TILT_NULL : {
    tiltread = TILT_UP_Y;
    break;}
  case TILT_UP_X : {
    tiltread = TILT_UP_BOTH;
    break;}
  case TILT_UP_Y : {
    tiltread = TILT_END_Y;
    break;}
  case TILT_END_X : {
    tiltread = TILT_END_X_UP_Y;
    break;}
  case TILT_UP_BOTH : {
    tiltread = TILT_END_Y_UP_X;
    break;}
  case TILT_END_X_UP_Y : {
    tiltread = TILT_END;
    break;}
```

```
  default : {
    fatalerror(ERRORUNDEFINEDTILTSTATE);
    break;}
  }
}

//Interuption on INT0
SIGNAL(SIG_INTERRUPT0) {
  GIMSK = bintodec(0,0,0,0,0,0,0,0); //Disable external interrupt
  oldtiltread = tiltread;
  switch(tiltread) {
  case TILT_NULL : {
    tiltread = TILT_UP_X;
    break;}
  case TILT_UP_X : {
    tiltread = TILT_END_X;
    break;}
  case TILT_UP_Y : {
    tiltread = TILT_UP_BOTH;
    break;}
  case TILT_UP_BOTH : {
    tiltread = TILT_END_X_UP_Y;
    break;}
  case TILT_END_Y : {
    tiltread = TILT_END_Y_UP_X;
    break;}
  case TILT_END_Y_UP_X : {
    tiltread = TILT_END;
    break;}
  default : {
    fatalerror(ERRORUNDEFINEDTILTSTATE);
    break;}
  }
}

//Implementation of the state-machine to read tilt on both axis
void readtilt() {

  if (GIMSK != 0) fatalerror(ERRORTILTINTERRUPT);

  tiltread = TILT_NULL;
  MCUCR = bintodec(0,0,0,0,1,1,1,1); //Set INT0 and INT1 on Rising edge
  GIFR = bintodec(1,1,0,0,0,0,0,0); //Clear the last interrupt if there's one
  GIMSK = bintodec(1,1,0,0,0,0,0,0); //Enable external interrupt on
                                     //INT0 and INT1

  while (tiltread == TILT_NULL) {}

  //Set 0 to the 16 bits counter
  TCNT1H = 0; //High byte = 0
  TCNT1L = 0; //Low byte = 0

  if (tiltread==TILT_UP_X) {
    tiltlowstartx = TCNT1L;
    tilthighstartx = TCNT1H;
    MCUCR = bintodec(0,0,0,0,1,1,1,0); //Set INT0 on Falling edge and
```

```
                                    //INT1 on Rising edge
  GIMSK = bintodec(1,1,0,0,0,0,0,0); //Enable external interrupt on
                                    //INT0 and INT1
  while (tiltread == TILT_UP_X) {}
}

if (tiltread==TILT_UP_Y) {
  tiltlowstarty = TCNT1L;
  tilthighstarty = TCNT1H;
  MCUCR = bintodec(0,0,0,0,1,0,1,1); //Set INT1 on Falling edge and
                                    //INT0 on Rising edge
  GIMSK = bintodec(1,1,0,0,0,0,0,0); //Enable external interrupt on
                                    //INT0 and INT1
  while (tiltread == TILT_UP_Y) {}
}

if (tiltread==TILT_END_X) {
  tiltlowx = TCNT1L - tiltlowstartx;
  tilthighx = TCNT1H - tilthighstartx;
  //MCUCR stays unchanged (INT1 on Falling edge)
  GIMSK = bintodec(1,0,0,0,0,0,0,0); //Enable external interrupt on INT1
  while (tiltread == TILT_END_X) {}
}

if (tiltread==TILT_END_Y) {
  tiltlowy = TCNT1L - tiltlowstarty;
  tilthighy = TCNT1H - tilthighstarty;
  //MCUCR stays unchanged (INT0 on Falling edge)
  GIMSK = bintodec(0,1,0,0,0,0,0,0); //Enable external interrupt on INT1
  while (tiltread == TILT_END_Y) {}
}

if (tiltread==TILT_UP_BOTH) {
  if (oldtiltread == TILT_UP_X) { //We just got a Rising edge on INT1
    tiltlowstarty = TCNT1L;
    tilthighstarty = TCNT1H;
  } else { //The Rising edge is on INT0
    tiltlowstartx = TCNT1L;
    tilthighstartx = TCNT1H;
  }
  MCUCR = bintodec(0,0,0,0,1,0,1,0); //Set INT1 and INT0 on Falling edge
  GIMSK = bintodec(1,1,0,0,0,0,0,0); //Enable external interrupt on INT1
                                    //and INT0
  while (tiltread == TILT_UP_BOTH) {}
}

if (tiltread==TILT_END_X_UP_Y) {
  if (oldtiltread == TILT_END_X) { //We just got a Rising edge on INT1
    tiltlowstarty = TCNT1L;
    tilthighstarty = TCNT1H;
  } else { //We just got a Falling edge on INT0
    tiltlowx = TCNT1L - tiltlowstartx;
    tilthighx = TCNT1H - tilthighstartx;
  }
  MCUCR = bintodec(0,0,0,0,1,0,1,0); //Set INT1 and INT0 on Falling edge
  GIMSK = bintodec(1,0,0,0,0,0,0,0); //Enable external interrupt on INT1
```

```
    while (tiltread == TILT_END_X_UP_Y) {}
  }

  if (tiltread==TILT_END_Y_UP_X) {
    if (oldtiltread == TILT_END_Y) { //We just got a Rising edge on INT0
      tiltlowstartx = TCNT1L;
      tilthighstartx = TCNT1H;
    } else { //We just got a Falling edge on INT1
      tiltlowy = TCNT1L - tiltlowstarty;
      tilthighy = TCNT1H - tilthighstarty;
    }
    MCUCR = bintodec(0,0,0,0,1,0,1,0); //Set INT1 and INT0 on Falling edge
    GIMSK = bintodec(0,1,0,0,0,0,0,0); //Enable external interrupt on INT0
    while (tiltread == TILT_END_Y_UP_X) {}
  }

  if (tiltread==TILT_END) {
    if (oldtiltread == TILT_END_X_UP_Y) { //We just got a Falling edge on
                                          //INT1
      tiltlowy = TCNT1L - tiltlowstarty;
      tilthighy = TCNT1H - tilthighstarty;
    } else { //We just got a Falling edge on INT0
      tiltlowx = TCNT1L - tiltlowstartx;
      tilthighx = TCNT1H - tilthighstarty;
    }
  }
}

//Main loop, called after init.
void mainloop() {

  unsigned char pwmcounter;
  unsigned int tiltcounter=0;
  unsigned int recordcounter=0;
  unsigned char muscle;

  for (;;) {

    recordcounter++;
    if (recordcounter % RECORDRATE == 0) {
      for (muscle=0;muscle<NBMUSCLES;muscle++)
if (protectionbit[muscle]==0) updateintegral(muscle,level[muscle]);
else updateintegral(muscle,PROTECTIONLEVEL);

      historyindex++;
      if (historyindex>=HISTORYLENGTH) historyindex=0;
      recordcounter=0;
    }

    for (pwmcounter = 0; pwmcounter < 128; pwmcounter++) {
      //PWM modulation
      for (muscle=0;muscle<NBMUSCLES;muscle++) {
if (pwmcounter < level[muscle]) {
  if (setmuscleon(muscle,DONTFORCE)==-1) {
    if (pwmcounter <= PROTECTIONLEVEL) setmuscleon(muscle,FORCE);
    else setmuscleoff(muscle);
```

```
  }
} else setmuscleoff(muscle);
      }
    }

    tiltcounter++;
    if (tiltcounter >= NBCYCLESTILT) {
#ifdef TILTPLUGGED
      readtilt();
#endif
      tiltcounter = 0;
    }

    for(muscle=0;muscle<NBMUSCLES;muscle++)
      level[muscle] = nextlevel[muscle];
  }
}

//Init
int main() {

  int muscle;
  unsigned char ledstatus;

  //Data Out Init
  outp(0xff,DDRB);
  outp(0xff,DDRC);


  for(muscle=0;muscle<NBMUSCLES;muscle++) {
    integraltotal[muscle] = 0;
    for(historyindex=0;historyindex<HISTORYLENGTH;historyindex++)
      history[muscle][historyindex] = 0;
  }
  historyindex = 0;

  //Serial Port Init
  // Set the baudrate to 19,200 bps using a 3.6864MHz crystal
  UBRR = 11;
  // Set the control register to enable reception and receive interrupt
  UCR = bintodec(1,0,0,1,1,0,0,0);//

  for(muscle=0;muscle<NBMUSCLES;muscle++) {
    musclebit[muscle] = 1;
    protectionbit[muscle] = 0;
    level[muscle] = 0;
    nextlevel[muscle] = level[muscle];
  }

  tilthighx = 0;
  tiltlowx = 0;

  serialfirstbyte = 0;

  //Init 16 bits counter
  //OCR1AH = 255;
```

```
  //OCR1AL = 255;
  //TCCR1A = bintodec(0,1,0,0,0,0,0,0);
  TCCR1B = bintodec(0,0,0,0,0,0,0,1); //Sync on the clock

  //Init the 8 bit counter
  TCCR0 = bintodec(0,0,0,0,0,1,0,1);

  //enable interrupts
  sei();

  mainloop();
}
```

C Fichiers *Matlab* pour la communication

C.1 Initialisation du port série

```
%Syntax : initserial
%by Felix Chenier
%Started : June 2nd, 2004
%This function without parameter initializes the serial port to be able to
%send data to the controller.

function initserial
global com1;
global settilt_lastcommand;
com1 = serial('COM1', 'BaudRate', 19200);
fopen(com1);
```

C.2 Envoyer un octet

```
function sendbyte(bytetosend)
%Syntax : sendlbyte(bytetosend)
%by Felix Chenier
%Started : July 21st, 2004
%This function sends the desired command to the micro-controller.
%To send a muscle level, you can use this command two times or use
%the command sendlevel.
%bytetosend must be between 0 and 127.

global com1;

if ((bytetosend>127) | (bytetosend<0))
    fprintf('bytetosend be between 0 and 127')
    return
end

fwrite(com1,char(bytetosend));
```

C.3 Assigner un niveau à un muscle

```
%Syntax : initserial
%by Felix Chenier
%Started : June 2nd, 2004
%This function without parameter initializes the serial port to be able to
```

```
%send data to the controller.

function initserial
global com1;
global settilt_lastcommand;
com1 = serial('COM1', 'BaudRate', 19200);
fopen(com1);
```

C.4 Attendre n secondes

```
function wait(sec)
%Syntax : wait(sec)
%by Felix Chenier
%Started : July 21st, 2004
%This function waits "sec" seconds and returns.

tic;
while(toc<sec)
end
```

C.5 Marcher

```
function walk(steps)
%Syntax : walk(steps)
%by Felix Chenier
%Started : July 21st, 2004
%This function makes the robot do a simple walk.
%steps is the number of steps to do.

waittime = 1;
hilevelshort = 70;
hilevellong = 80;
lolevel = 1;


for i=1:steps

fprintf('\nStep %i',i);

sendlevel(3,hilevelshort);
wait(waittime);

sendlevel(2,hilevellong);
wait(waittime);

sendlevel(1,lolevel);
wait(waittime);

sendlevel(3,lolevel);
wait(waittime);

sendlevel(4,hilevelshort);
wait(waittime);

sendlevel(1,hilevellong);
wait(waittime);
```

```
sendlevel(2,lolevel);
wait(waittime);

sendlevel(4,lolevel);
wait(waittime);
end

fprintf('\nRelaxing');

sendlevel(1,lolevel);
sendlevel(3,lolevel);
```

D Matériel utilisé

D.1 Expérimentation

Construction des prototypes : Jeu *Meccano Multi-Models - 50 modèles* (www.meccano.com)
Micro-contrôleur : *Atmel* ATS908515 (www.atmel.com)
Programmation du micro-contrôleur : *Atmel* STK500 (www.atmel.com)
Compilateur C : *WinAVR* (winavr.sourceforge.net)
Logiciel de contrôle pour l'UART : *Matlab* (www.mathworks.com)

D.2 Rédaction de ce document

Traitement de texte : *Emacs* et LaTeX sur *Debian GNU/Linux - Sid* (www.debian.org)
Figures en deux dimensions : *Dia* (www.gnome.org/projects/dia)
Figures en trois dimensions : *Blender* (www.blender3d.org)
Édition des photos : *The Gimp* (www.gimp.org)

E Références

1. Roger G. Gilbertson. *Muscle Wires Project Book, A Hands-on Guide to Amazing Robotic Muscles that Shorten When Electrically Powered.* Mondo-Tronics, 2000, troisième édition.
2. P. Auer, H. M. Burgsteiner, W. Maass. *The p-Delta Learning Rule for Parallel Perceptrons*, Institute for Theoretical Computer Science, Technische Universität Graz, Mars 2002.
3. `www.avrfreaks.net`. Site internet traitant de plusieurs aspects des micro-contrôleurs Atmel.
4. Atmel Corporation. *AVR STK500 User Guide.* Rev. 1925B-AVR-04/02.
5. Atmel Corporation. *AVR RISC Microrontroller Data Book.* Août 1999.

F Remerciements

Je remercie sincérement M. Jean-Jules Brault pour m'avoir confié ce projet, et pour m'avoir alloué toutes les ressources nécessaires pour arriver à ces résultats.

Je remercie également MM. Gilles Guérette et Jacques Girardin pour les circuits imprimés, et surtout pour l'expérience technique qu'ils m'ont apportée.