

# RAPPORT DE FIN DE PROJET

## DOCUMENTATION TECHNIQUE Lévitation Acoustique



*Auteurs :*

Arthur BENOIT  
Malo BEILLARD  
Raphaël DAUVERS  
Mattéo DENIS  
Jean FREGEVILLE  
Mohamed LAAROUSSI  
Elisa LE BOHEC  
Mehdi MANSSOURI

*Encadrants :*

M. BOU MATAR-LACAZE  
M. TALBI

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Qu'est-ce que la lévitation acoustique ? . . . . .	3
1.2	Attentes du partenaire . . . . .	4
1.2.1	Contexte et présentation du projet . . . . .	4
1.2.2	Partenaire du projet . . . . .	4
1.2.3	Contacts . . . . .	4
1.2.4	Objectifs . . . . .	5
1.2.5	Résultat escompté . . . . .	5
1.2.6	Compétences attendues . . . . .	5
<b>2</b>	<b>Première phase : TinyLev</b>	<b>6</b>
2.1	Pourquoi ce prototype ? . . . . .	6
2.2	Notice de montage du TinyLev . . . . .	6
2.3	Notice d'utilisation du TinyLev . . . . .	10
2.4	Résultats . . . . .	11
2.5	Vers la phase 2 . . . . .	13
<b>3</b>	<b>Deuxième phase : lévitateur à une plaque</b>	<b>14</b>
3.1	Pourquoi ce prototype ? . . . . .	14
3.2	Les théories sous-jacentes . . . . .	14
3.2.1	Théorie générale de la lévitation acoustique . . . . .	14
3.2.2	La théorie Optimizer . . . . .	16
3.2.3	La théorie Vortex . . . . .	17
3.2.4	Programmes divers . . . . .	23
3.3	Notice de montage du lévitateur à une plaque . . . . .	24
3.3.1	Étapes clés . . . . .	24
3.3.2	Liste du matériel et commande . . . . .	24
3.3.3	Conception et montage du prototype 2 . . . . .	26
3.3.4	Soudure du PCB et ajout des connecteurs sur les nappes de câbles . . . . .	28
3.3.5	Vérification de la continuité du courant dans les fils avec un multimètre . . . . .	31
3.3.6	Montage des transducteurs sur la plaque . . . . .	32
3.3.7	Assignation des pins . . . . .	35
3.4	Notice d'utilisation du lévitateur à une plaque . . . . .	36
3.4.1	Codes Arduino . . . . .	36
3.4.2	Mise en Route et Utilisation . . . . .	39
3.5	Résultats . . . . .	43
3.6	Problèmes rencontrés durant la phase 2 . . . . .	44
3.7	Vers la phase 3 . . . . .	46
3.7.1	Résumé de l'étude de brevets . . . . .	46
3.7.2	Justification du choix d'application . . . . .	46
3.7.3	Ondes progressives périodiques . . . . .	47
3.7.4	Ondes stationnaires . . . . .	48
3.7.5	Pourquoi privilégier les ondes stationnaires ? . . . . .	48
3.7.6	Conclusion . . . . .	50
3.7.7	Proposition de modèle pour la phase 3 . . . . .	51
<b>4</b>	<b>Troisième phase : lévitateur à deux plaques</b>	<b>53</b>
4.1	Pourquoi ce prototype ? . . . . .	53
4.2	Evolution des théories physiques et nouveaux programmes . . . . .	53
4.2.1	Méthode optimizer . . . . .	53
4.2.2	Méthode vortex . . . . .	55

4.2.3	Pistes de développements . . . . .	57
4.3	Notice de montage du lévitateur à deux plaques . . . . .	57
4.3.1	Étapes clés . . . . .	58
4.3.2	Liste du matériel et commande . . . . .	58
4.3.3	Conception et montage du prototype 3 . . . . .	58
4.4	Notice d'utilisation du lévitateur à deux plaques . . . . .	60
4.4.1	Étapes clés . . . . .	60
4.4.2	Prise en main des codes Arduino . . . . .	60
4.4.3	Prise en main pratique . . . . .	62
4.5	Résultats . . . . .	64
<b>5</b>	<b>Conclusion</b>	<b>65</b>
<b>6</b>	<b>Bibliographie</b>	<b>66</b>
6.1	Notre GitHub . . . . .	66
6.2	Vers la phase 3 . . . . .	66
<b>7</b>	<b>Annexes</b>	<b>67</b>
7.1	ArduinoMega . . . . .	67
7.2	NewDriverMega . . . . .	70
7.2.1	Explications des modifications . . . . .	73
7.3	Code python : Passage des phases réelles aux phases en bit . . . . .	75
7.4	Sketch Arduino - prototype à une plaque . . . . .	77
7.5	Communication à deux cartes Arduino . . . . .	79
7.5.1	Carte <i>maître</i> . . . . .	79
7.5.2	Carte <i>esclave</i> . . . . .	81
7.6	Programme de calcul de phase . . . . .	83
7.6.1	Méthode optimizer . . . . .	83
7.6.2	Méthode vortex . . . . .	86

# 1 Introduction

## 1.1 Qu'est-ce que la lévitation acoustique ?

Une onde acoustique est une onde mécanique correspondant à la propagation d'une variation de pression dans un milieu matériel tel que l'air, l'eau ou un solide. Lorsqu'une source — par exemple un haut-parleur ou un transducteur ultrasonore — vibre, elle comprime puis détend localement le milieu. Cette perturbation se propage alors de proche en proche sous la forme d'une onde longitudinale (i.e. les particules du milieu oscillent dans la même direction que la propagation de l'onde). Ce phénomène est illustré par la figure 1

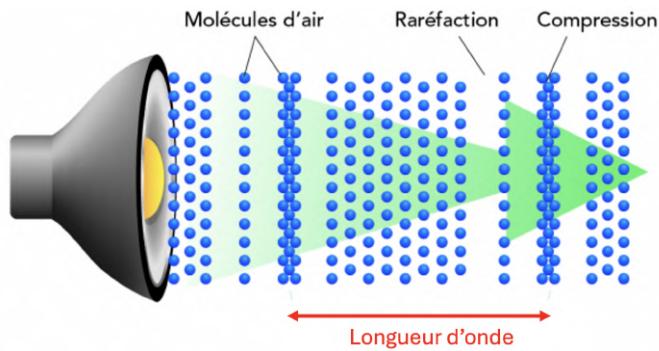


FIGURE 1 – Schéma d'une onde acoustique - [parlonssciences.ca](http://parlonssciences.ca)

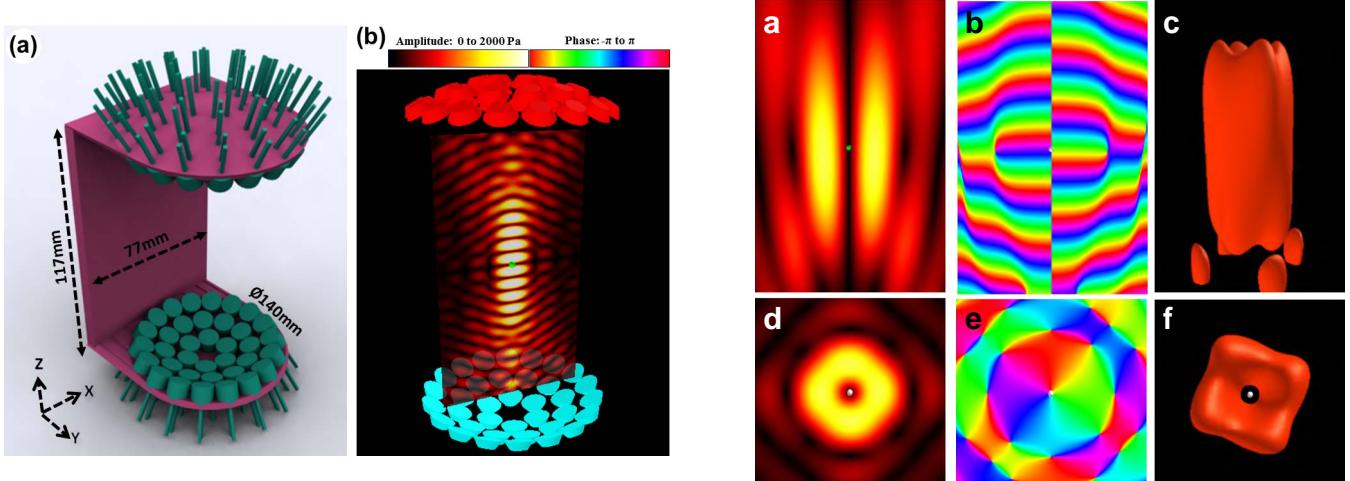
Dans l'air, ces variations de pression sont transportées sans déplacement de matière et véhiculent de l'énergie. Leur vitesse dépend des propriétés du milieu (densité, compressibilité) et leur fréquence conditionne le domaine d'audibilité : entre 20 Hz et 20 kHz pour l'oreille humaine. Au-delà, on parle d'ultrasons, typiquement entre 20 kHz et quelques centaines de kilohertz, fréquences utilisées dans les systèmes de lévitation acoustique.

La lévitation acoustique est, quant-à-elle, un procédé physique permettant de suspendre des objets solides ou liquides dans l'air grâce à l'action de forces générées par des ondes ultrasonores.

Lorsqu'un champ acoustique suffisamment intense interagit avec une particule, celle-ci subit une force de radiation acoustique, résultant de l'interférence entre l'onde incidente et les ondes qu'elle diffuse. Si cette force compense le poids de la particule, un équilibre est atteint et la particule peut alors léviter sans contact. Cette approche ouvre la voie à des manipulations précises, propres et reproductibles, particulièrement utiles dans les domaines de la microfluidique aérienne, de l'analyse chimique ou de la manipulation de matériaux sensibles.

La lévitation acoustique exploite typiquement deux types de champs ultrasonores :

- Les ondes stationnaires, qui créent des nœuds de pression stables permettant un piégeage robuste ;
- Les ondes progressives structurées, telles que les vortex acoustiques, capables de générer des minima locaux de pression et d'exercer des forces latérales ou rotationnelles.



(a) (a) Rendu 3D du modèle, (b) Transducteurs importés dans la simulation ; leur couleur indique la phase d'émission. Une coupe du champ d'amplitude est affichée, accompagnée d'une particule positionnée dans l'un des noeuds acoustiques.

(b) Piège à vortex généré avec un réseau plat  $20 \times 20$ , à 12 cm du centre. Champ d'amplitude (a,d), champ de phase (b,e) et isosurfaces d'amplitude de 2 kPa (c,f).

FIGURE 2 – Comparaison des deux types de champs ultrasonores

Le projet présenté dans ce rapport s'inscrit dans cette thématique. Il vise à concevoir, modéliser et réaliser plusieurs prototypes de lévitateurs ultrasonores, en combinant des approches matérielles (matrices de transducteurs, drivers électroniques, architecture mécanique) et des approches théoriques (calculs de phases, synthèse inverse, potentiel de Gorkov, optimisation numérique). L'enjeu est alors d'obtenir des pièges acoustiques contrôlables, précis et suffisamment stables pour permettre la manipulation de plusieurs particules dans l'espace.

## 1.2 Attentes du partenaire

### 1.2.1 Contexte et présentation du projet

Le groupe **AIMAN-FILMS** développe depuis quelques années des pinces acoustiques pour des applications biomédicales (déplacement de cellules sans contact) grâce aux technologies de microfabrication en salle blanche. Vous pouvez consulter leurs travaux ici : [Nature Communications](#).

Dans le cadre de ce projet, nous souhaitons réaliser un système **faible coût** de manipulation sans contact d'objets millimétriques (billes de polystyrène, insectes) dans l'air en utilisant les mêmes concepts de pinces acoustiques. Un exemple de réalisation est disponible via ce lien : [Ultrasonic Levitation in Air](#).

**Mots clés :** Lévitation acoustique, manipulation d'objets sans contact, impression 3D, électronique, capteurs.

### 1.2.2 Partenaire du projet

Le projet sera réalisé en partenariat avec le groupe **AIMAN-FILMS** de l'**IEMN** (Institut d'Électronique, de Microélectronique et de Nanotechnologie) :

<https://www.iemn.fr/la-recherche/les-groupes/aiman-films>

### 1.2.3 Contacts

- Olivier Bou Matar-Lacaze, Professeur des Universités à Centrale Lille / IEMN : [olivier.boumatar@cermics.enpc.fr](mailto:olivier.boumatar@cermics.enpc.fr)

- **Abdelkrim Talbi**, Professeur des Universités à Centrale Lille / IEMN : [abdelkrim.talbi@centralelille.fr](mailto:abdelkrim.talbi@centralelille.fr)

#### 1.2.4 Objectifs

L'objectif du projet est de réaliser un système de lévitation acoustique permettant de manipuler sans contact des objets millimétriques (billes de polystyrène, insectes, etc.).

#### 1.2.5 Résultat escompté

Réaliser un **prototype fonctionnel** de lévitation acoustique permettant de manipuler sans contact des objets millimétriques et démontrer une ou plusieurs applications potentielles de ce système.

#### 1.2.6 Compétences attendues

- Principes physiques des pinces acoustiques.
- Pilotage d'un Arduino.
- Réalisation de cartes électroniques.
- Impression 3D.

## 2 Première phase : TinyLev

### 2.1 Pourquoi ce prototype ?

Ce premier prototype a servi de base expérimentale pour nous familiariser avec les principes fondamentaux de la lévitation acoustique. Le TinyLev est un prototype de lévitation acoustique de particule, composé de 2 faces de transducteurs en vis-à-vis. Cela crée des ondes stationnaires entre les deux plaques.

Pendant cette première phase, nous nous sommes formés sur la lévitation acoustique, pour comprendre le fonctionnement de ce prototype et des futurs prototypes que nous allions fabriquer. Au-delà de la théorie, ce premier prototype a été déterminant pour la prise en main des composants électroniques, l'apprentissage de la soudure sur carte électronique et l'implémentation du code Arduino. La réussite de ces premiers essais, marqués par la mise en lévitation effective de particules, a constitué un jalon essentiel et motivant pour la suite du projet.

Comme nous le verrons par la suite, nous nous sommes largement appuyé cette [documentation](#) pour construire ce premier prototype de lévitation.

### 2.2 Notice de montage du TinyLev

#### I - Matériel et préparation

**Étape 1 :** Rassemblez tous les composants nécessaires (transducteurs à ultrasons, Arduino Nano, driver de moteur L298N, alimentation, etc.), commandés sur ce [site](#).

#### Rassembler tout le matériel nécessaire :

- Imprimante 3D ;
- Fer à souder, étain et flux ;
- Pistolet à colle chaude ;
- Multimètre ;
- Pince à dénuder ;
- Tournevis et pinces ;
- Oscilloscope avec deux sondes.

**Étape 2 :** Imprimez en 3D la structure de base (support) qui accueillera les transducteurs, voici le lien [STL](#).

**Étape 3 :** Nettoyez la pièce imprimée (limez les bords et les trous) pour que les composants s'insèrent parfaitement. Le rendu final du socle devrait ressembler à la figure 3 (à la couleur de votre matériel d'impression près).

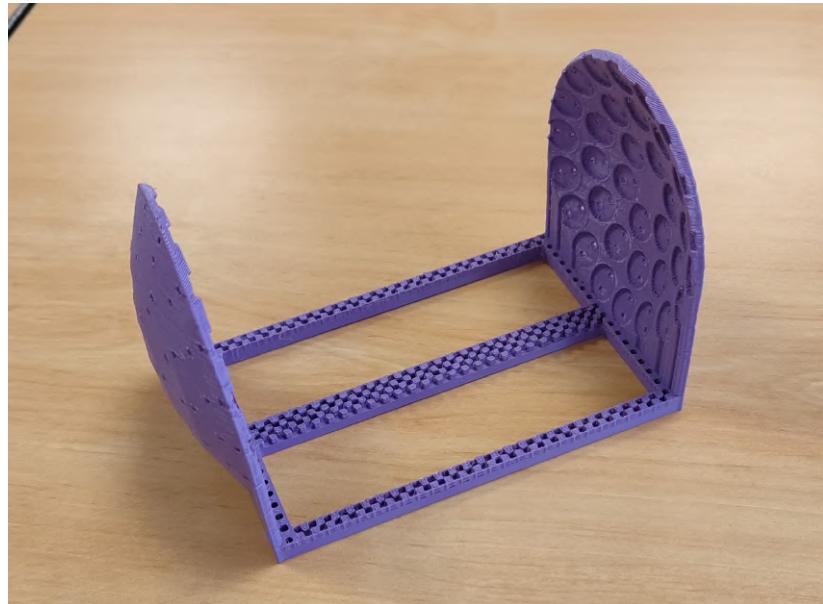


FIGURE 3 – Socle TinyLev imprimé en 3D

## II - Montage des transducteurs

**Étape 4 :** Identifiez la polarité des transducteurs (+/-) ; pour cela voir le paragraphe sur la polarité des transducteurs.

**Étape 5 :** A l'aide du pistolet à colle chaude, collez les transducteurs dans les logements de la base imprimée en veillant à orienter toutes les pattes marquées vers le centre.

**Étape 7 :** Reliez les pattes des transducteurs entre elles en enroulant du fil dénudé pour former des anneaux concentriques.

**Étape 8 :** Soudez les connexions faites précédemment afin d'assurer un bon contact électrique.

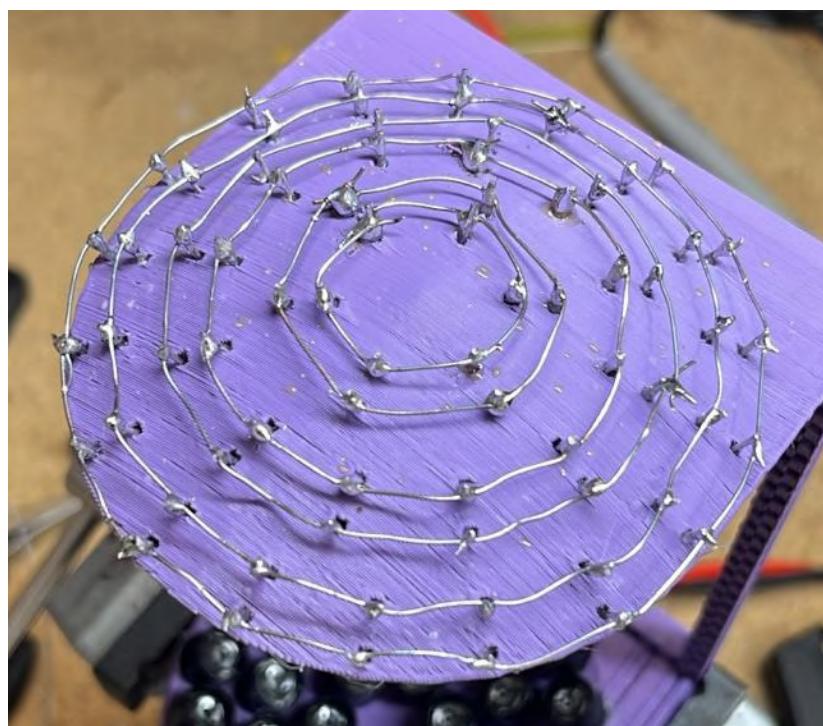


FIGURE 4 – Câblage des anneaux concentriques

### III - Câblage et électronique

**Étape 10 :** Préparez quatre longs fils (deux rouges, deux noirs) pour relier les rangées de transducteurs au driver.

**Étape 11 :** Soudez ces longs fils aux anneaux de transducteurs que vous avez créés.

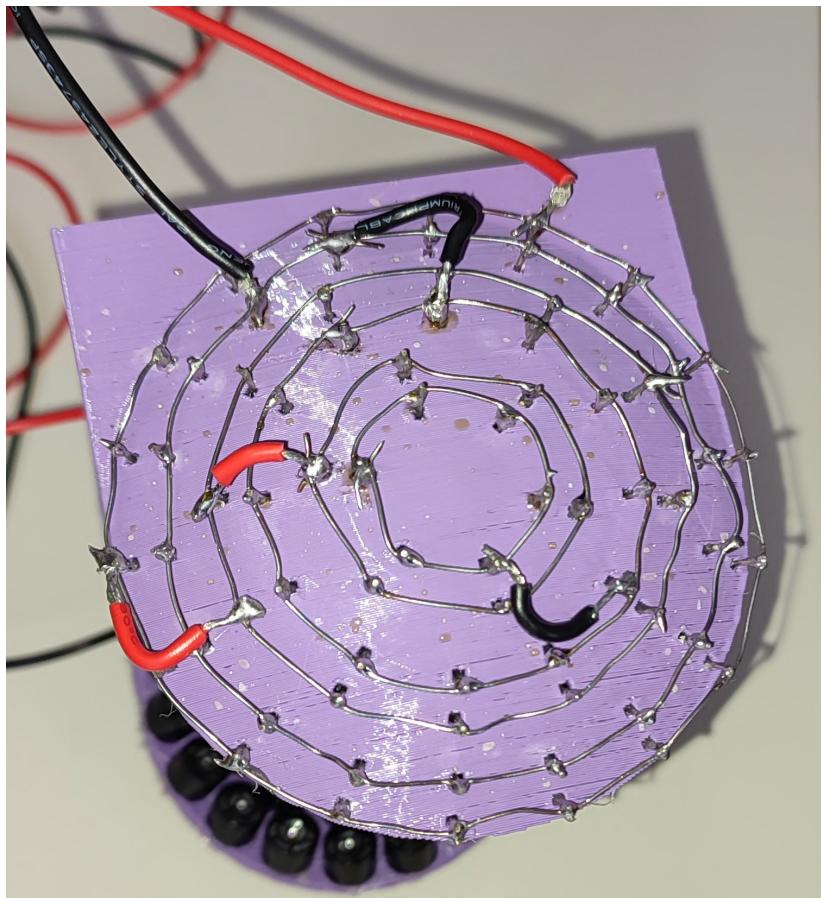


FIGURE 5 – Câblage des anneaux concentriques

**Étape 12 :** Soudez les barrettes de connexion (headers) sur l'Arduino Nano.

### IV - Assemblage final et tests

**Étape 13 :** Préparez le connecteur d'alimentation DC et l'interrupteur en soudant les fils nécessaires.

**Étape 14 :** Collez le connecteur et l'interrupteur, puis effectuez tout le câblage entre l'alimentation, le driver et l'Arduino selon le schéma.

La cablage entre l'Arduino et le prototype doit être similaire à celui-ci :

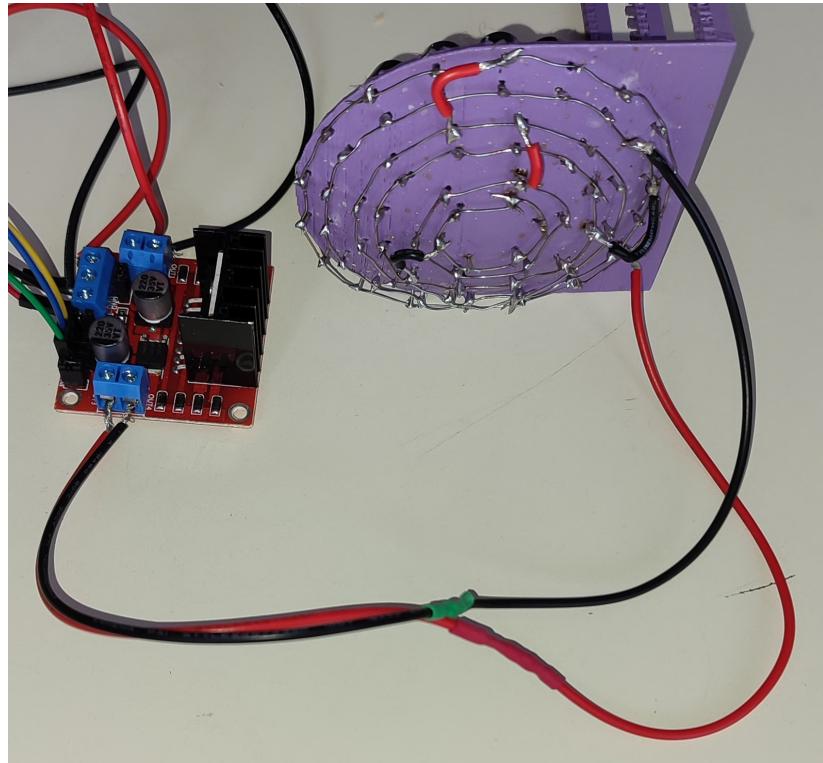


FIGURE 6 – Câblage entre l'Arduino Nano et le TinyLev

**Étape 15 :** Mettez sous tension pour tester le driver et vérifier (si possible avec un oscilloscope) qu'il génère bien un signal carré de 40kHz.

**Étape 16 :** Vérifiez avec un multimètre qu'il n'y a aucun court-circuit entre les fils rouges et noirs.

**Étape 17 :** Testez les transducteurs pour vous assurer qu'ils fonctionnent et sont en phase.

## 2.3 Notice d'utilisation du TinyLev

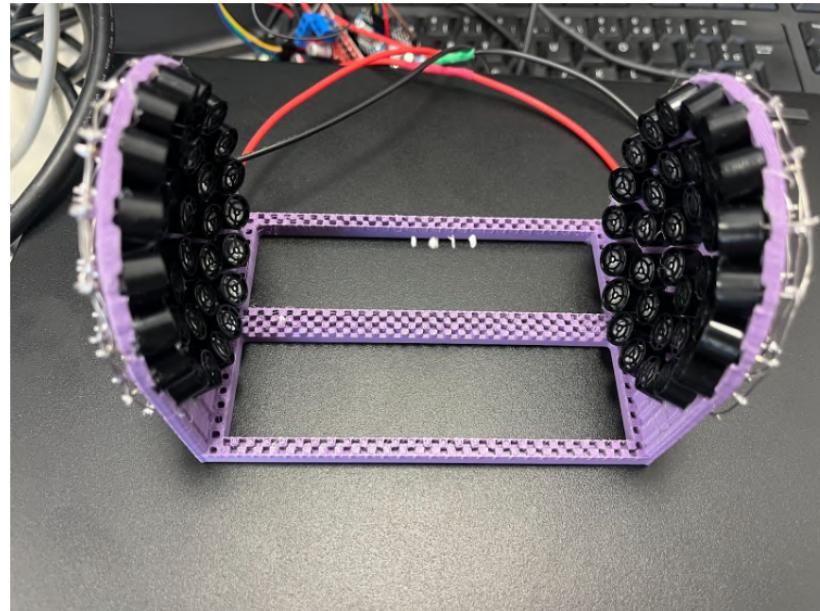


FIGURE 7 – TinyLev opérationnel

### Étape 1

Vérifier que tous les composants sont présents sur le système et que les branchements sont faits conformément au schéma ci-dessous :

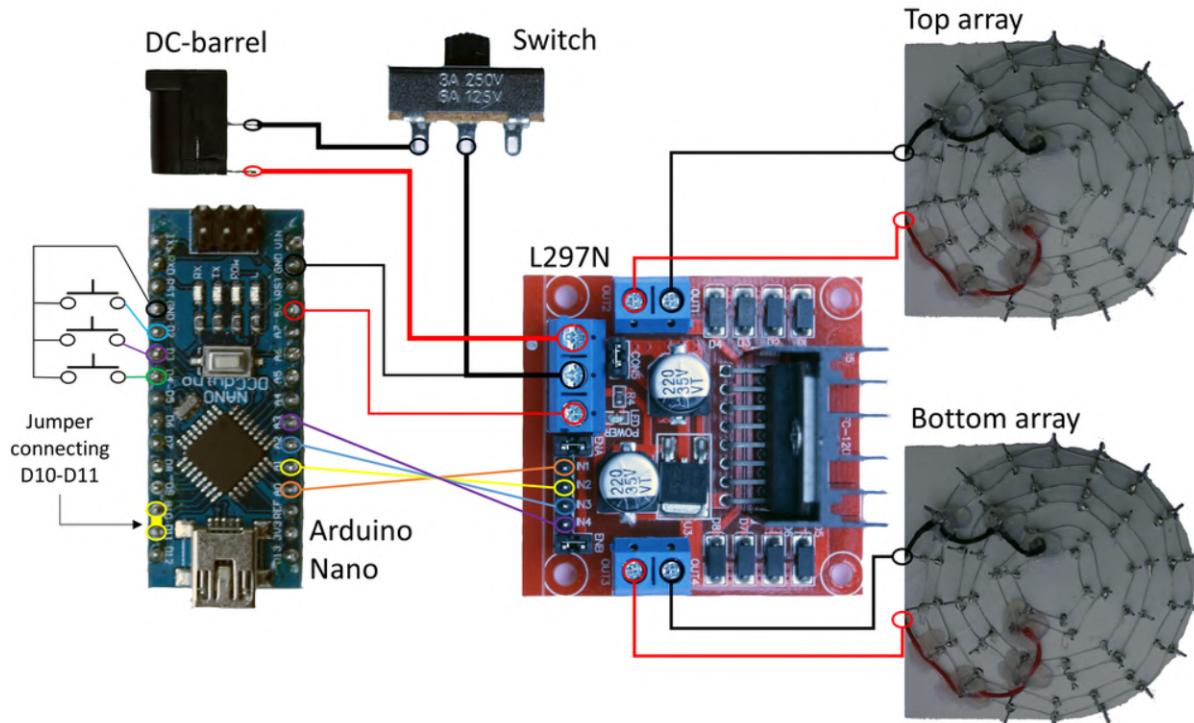


FIGURE 8 – Schéma de branchement du TinyLev - Source : [Instructables](#)

- **Vérification cruciale** : Vérifier en particulier les fils qui connectent l'Arduino Nano au L297N.
- **Jumper** : Vérifier qu'un fil connecte le port 10 au port 11 de l'Arduino (fil mâle-mâle).

## Étape 2

Brancher le système sur secteur avec le chargeur, sur le **DC-barrel**.

| **Attention :** *Un des chargeurs de la boîte ne fonctionne plus. Vérifiez que des lumières s'allument sur les cartes lors du branchement.*

## Étape 3

Branchez en USB la carte Arduino avec un PC muni du logiciel Arduino IDE.

## Étape 4

1. Rendez-vous sur la page : [Acoustic Levitator: 27 Steps - Instructables](#) ;
2. Ouvrez le fichier Arduino fourni à l'étape 13 du tutoriel.

## Étape 5 : Configuration du logiciel

Dans le menu **Outils** de l'Arduino IDE :

- **Type de carte** : Choisissez "Arduino Nano" ;
- **Processeur** : Choisissez "ATmega328P (Old Bootloader)" ;
- **Port** : Sélectionnez le port détecté (ex : COM3).

## Étape 6 : Mise en route

1. Cliquez sur le bouton **Vérifier** ;
2. Cliquez sur le bouton **Téléverser**.

## Étape 7 : Placer la bille

Le prototype 1 est maintenant en fonctionnement. Approchez une petite particule (petit morceau de polystyrène), elle devrait léviter.

| *Note technique : La distance minimale entre deux particules est de 4,25 mm.*

## 2.4 Résultats

Nous pouvons alors observer sur la figure 9 les noeuds et ventres d'une onde stationnaire générée par la géométrie du prototype.

Voici une image du TinyLev en fonctionnement :

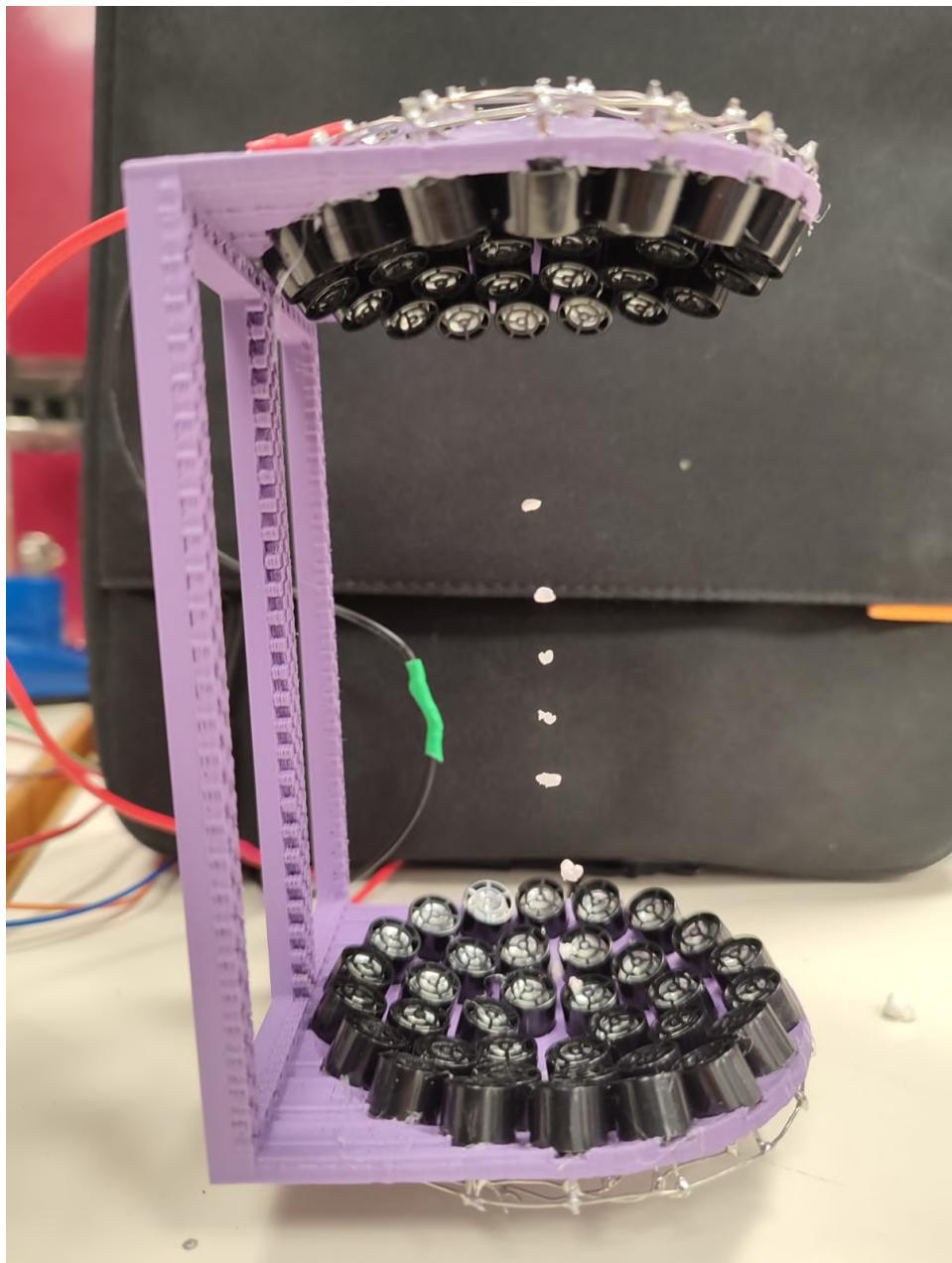


FIGURE 9 – TinyLev en fonctionnement

En effet, les transducteurs étant placés à une distance plus ou moins grande du point de focalisation souhaitée du fait de la géométrie, la loi de retard des ondes acoustiques assure une certaine forme d'ondes dans l'espace, donnant ici une onde stationnaire focalisée au centre des deux plaques selon le schéma suivant :

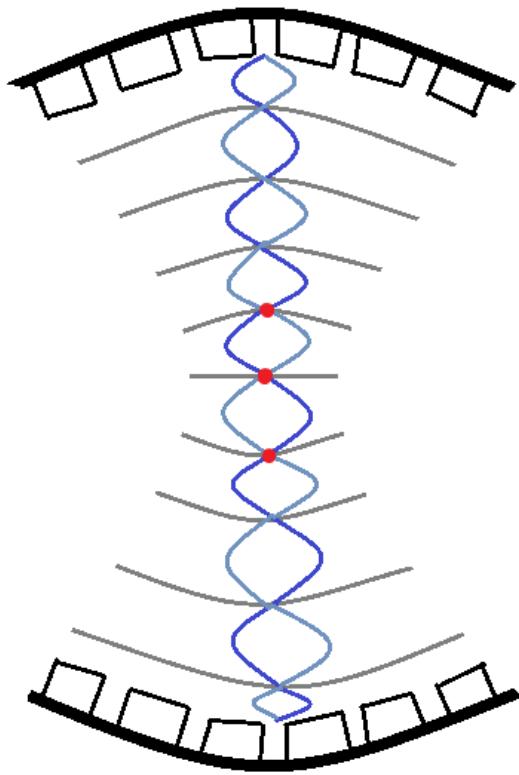


FIGURE 10 – Schéma des ondes stationnaires générées par la géométrie du TinyLev

## 2.5 Vers la phase 2

Comme expliqué en introduction, ce premier prototype avait pour objectif de nous familiariser avec le principe de lévitation acoustique. Dans cette configuration, la lévitation repose principalement sur la géométrie fixe du dispositif : c'est la forme même du prototype, et non un pilotage fin des transducteurs, qui permet de générer le champ stationnaire nécessaire à maintenir une particule en suspension.

Cette dépendance en la forme implique une limite majeure : le piège acoustique ne peut pas être déplacé sans modifier physiquement la structure du montage. Le champ généré reste parfaitement stationnaire, ce qui rend impossible tout déplacement contrôlé de la particule en lévitation.

Ainsi, pour progresser vers une manipulation dynamique des objets, il est indispensable de changer de modèle théorique et de s'orienter vers une approche permettant de créer un champ acoustique mobile. Cela nécessite un contrôle indépendant de chaque transducteur afin de pouvoir leur transmettre des phases distinctes. En ajustant en temps réel ces phases, il devient possible de déplacer le piège acoustique — et donc l'objet en lévitation — de manière contrôlée, ouvrant la voie à la seconde phase du projet.

### 3 Deuxième phase : lévitateur à une plaque

#### 3.1 Pourquoi ce prototype ?

Comme évoqué à la fin de la partie précédente, ce prototype a pour objectif premier de permettre le déplacement d'une particule.

Nous quittons donc le modèle stationnaire du TinyLev pour utiliser des ondes progressives générées par un ensemble de transducteurs commandés séparément.

La forme du champ théorique s'approchera ainsi du schéma suivant :

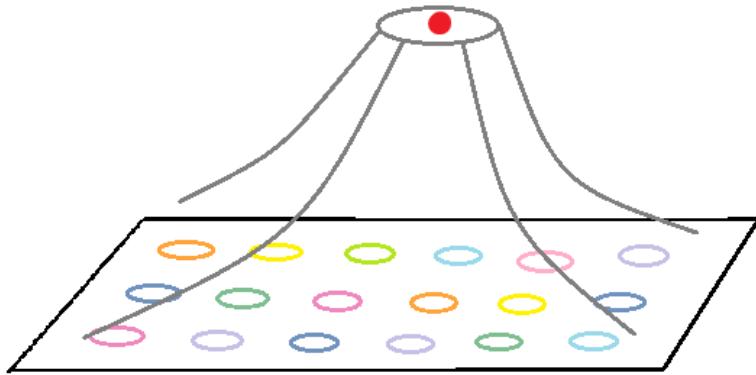


FIGURE 11 – Schéma théorique de la forme d'ondes du prototype 2

#### 3.2 Les théories sous-jacentes

##### 3.2.1 Théorie générale de la lévitation acoustique

###### Cadre théorique et hypothèses

La description théorique de la lévitation acoustique s'inscrit dans le cadre de l'acoustique linéaire faiblement non linéaire. Le milieu de propagation est supposé homogène, isotrope et newtonien, et les perturbations acoustiques sont considérées comme de faible amplitude devant les grandeurs thermodynamiques d'équilibre. Dans ce régime, les équations fondamentales de la mécanique des fluides — conservation de la masse, conservation de la quantité de mouvement et relation d'état — peuvent être linéarisées pour décrire la propagation des ondes acoustiques.

La particule mise en lévitation est généralement supposée sphérique et de rayon petit devant la longueur d'onde acoustique. Cette hypothèse, dite approximation de Rayleigh, permet d'obtenir des expressions analytiques simplifiées des forces acoustiques moyennes exercées sur la particule.

###### Force de radiation acoustique

Lorsqu'une particule est plongée dans un champ acoustique intense, elle subit une force moyenne résultant de l'interaction entre l'onde incidente et les ondes diffusées par la particule. Cette force, appelée force de radiation acoustique, est distincte des forces associées à la pression acoustique et

correspond à une force moyenne sur une période acoustique.

Dans le régime des petites particules, c'est-à-dire lorsque le rayon de la particule est faible devant la longueur d'onde acoustique, la force de radiation peut être décrite de manière analytique à partir du potentiel de Gorkov. Cette approche est largement utilisée pour analyser le piégeage acoustique de particules solides ou liquides dans l'air.

## Potentiel de Gorkov

Pour une particule sphérique de rayon  $R_p$  plongée dans un champ acoustique harmonique, le potentiel de radiation acoustique  $U$  peut être exprimé sous la forme :

$$U = \frac{V}{4} \left[ f_1 \frac{\langle p^2 \rangle}{\rho_0 c_0^2} - f_2 \rho_0 \langle \vec{v}^2 \rangle \right], \quad (1)$$

où :

- $V = \frac{4}{3}\pi R_p^3$  est le volume de la particule,
- $p$  est la pression acoustique,
- $\vec{v}$  est la vitesse particulaire du fluide,
- $\rho_0$  et  $c_0$  sont respectivement la densité et la vitesse du son dans le milieu,
- $\langle \cdot \rangle$  désigne une moyenne temporelle sur une période acoustique,
- $f_1$  et  $f_2$  sont des coefficients dépendant des propriétés mécaniques de la particule :

$$f_1 = 1 - \frac{\kappa_p}{\kappa_0}, \quad f_2 = \frac{2(\rho_p - \rho_0)}{2\rho_p + \rho_0}, \quad (2)$$

où  $\kappa_p$  et  $\kappa_0$  désignent respectivement les compressibilités de la particule et du milieu.

La force de radiation acoustique  $\vec{F}$  exercée sur la particule est alors donnée par le gradient du potentiel :

$$\vec{F} = -\nabla U. \quad (3)$$

L'existence d'un piège acoustique stable nécessite donc l'existence d'un minimum local du potentiel  $U$  dans les trois directions de l'espace.

## Caractérisation d'un piège stable

Pour qu'une particule lévite de manière stable, la composante verticale de la force de radiation acoustique doit compenser son poids :

$$F_z \geq m_p g, \quad (4)$$

où  $m_p$  est la masse de la particule et  $g$  l'accélération de la pesanteur.

Par ailleurs il est nécessaire d'assurer le confinement de la particule à la position souhaitée. Pour ce faire il faut s'assurer que la particule se trouve dans un puit de potentiel, ainsi dès qu'elle sortira du piège elle subira une force la ramenant à sa position d'équilibre.

Enfin, pour s'assurer que cette position d'équilibre soit stable il faut s'assurer que le piège se trouve à un minimum de pression.

### 3.2.2 La théorie Optimizer

#### Principe

Nous avons vu avec la théorie de la lévitation acoustique qu'un bon piège se caractérisait par deux conditions. Tout d'abord pour être assuré que le piège formé soit stable, il faut s'assurer que la pression en ce point soit minimale. Ensuite il faut s'assurer que la particule ne s'échappe pas. Pour ce faire il faut à la fois que la force exercée par le champ de pression soit nulle dans le piège (pour ne pas éjecter la particule du piège) et que dès lors que celle-ci sorte du piège, le champ de pression exerce une force qui la ramène dans celui-ci. Pour satisfaire cette condition, on va chercher à maximiser le gradient de force au point du piège.

De plus, la littérature nous permet à la fois de calculer la pression en un point et de calculer la force, et donc son gradient, via le potentiel de Gorkov. La seule condition étant de connaître les phases de chaque transducteurs.

De là vient l'idée de chercher informatiquement les phases permettant de minimiser la grandeur suivante :

$$\mathcal{O}(\phi_j) = |p| - \nabla^2 U \quad (5)$$

#### Implémentation informatique

Les programmes que nous avons développer permettent de calculer les phases à appliquer à chaque transducteurs d'une plaque ultrasonore afin de créer un champ de pression assurant un piège optimal. Ils prennent en compte la position des transducteurs ainsi que la position du piège souhaité.

#### Description générale du fonctionnement

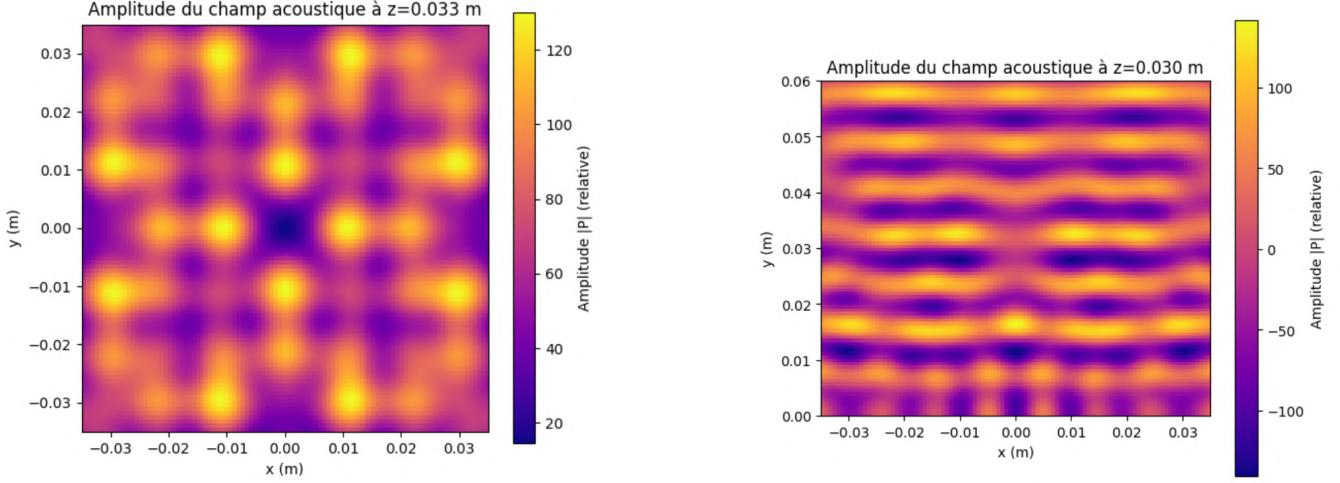
Les programmes sont tous construits comme suit :

1. Définition des paramètres physiques (paramètres liés aux transducteurs, liés à la particule, liés à l'air), puis création de l'espace de simulation (liste de position des transducteurs).
2. Fonctions permettant de calculer les paramètres dépendant exclusivement de la position du piège (les valeurs calculées sont ensuite sauvegardées dans un dictionnaire).
3. Fonctions permettant de calculer la fonction à minimiser (pression en un point et gradient de la force).
4. Fonction optimizer calculant les phases optimales et appel de celle-ci.

Ces programmes fournissent donc une liste de 64 phases permettant la creation d'un piège à la position choisie.

#### Réultats et limites

Avec cette méthode on arrive effectivement à produire un piège. Cependant les expérimentations ont révélé plusieurs limites. Tout d'abord la force de ces pièges est limitée, ensuite il arrive régulièrement que le piège ne soit pas tout à fait situé à la position souhaitée. Enfin, il apparaît que sa taille est un peu faible ce qui a tendance à rendre le piège instable.



(a) Champs de pression acoustique - Méthode Optimizer

(b) Champs de pression acoustique - Méthode Optimizer

FIGURE 12 – Champs de pression produit par la méthode Optimizer

### 3.2.3 La théorie Vortex

#### I - Définitions

##### I.1 Définition d'un vortex acoustique

Un vortex acoustique est un champ de pression acoustique structuré caractérisé par une variation hélicoïdale de la phase autour d'un axe de propagation. Contrairement à une onde acoustique plane ou à un faisceau focalisé classique, la phase d'un vortex n'est pas uniforme sur un plan transverse : elle varie continûment avec l'angle azimutal, ce qui conduit à l'apparition d'une singularité de phase sur l'axe central du faisceau.

Cette structure est généralement décrite par l'introduction d'une charge topologique notée  $m$ , correspondant au nombre de rotations de la phase de  $2\pi$  lors d'un tour complet autour de l'axe de propagation. Mathématiquement, cette propriété se traduit par une dépendance angulaire de la phase de la forme  $\exp(im\varphi)$ , où  $\varphi$  désigne l'angle azimutal en coordonnées cylindriques ou sphériques.

Dans notre cas on s'intéressera exclusivement au vortex sphérique, ceux-ci étant les seuls pouvant assurer la lévitation. En effet, dans le cas de vortex cylindrique lorsque  $m \neq 0$ , la phase devient indéfinie sur l'axe ( $r = 0$ ). Cette singularité impose nécessairement une annulation de l'amplitude du champ sur l'axe central du vortex ce qui réduit considérablement la force verticale produite par le piège.

##### I.2 Expression mathématique d'un vortex acoustique

Dans un cadre théorique général, un vortex acoustique peut être décrit par un champ de pression harmonique complexe de la forme :

$$p(r, \theta, \varphi, t) = A j_\ell(kr) P_\ell^m(\cos \theta) e^{i(m\varphi - \omega t)}, \quad (6)$$

où :

- $A$  est une amplitude complexe,
- $j_\ell$  est la fonction de Bessel sphérique d'ordre  $\ell$ ,

- $P_\ell^m$  est le polynôme associé de Legendre,
- $k$  est le nombre d'onde,
- $\omega$  est la pulsation,
- $(r, \theta, \varphi)$  sont les coordonnées sphériques,
- $m$  est la charge topologique du vortex.

Cette expression met en évidence plusieurs aspects essentiels de la structure du vortex. La dépendance en  $\varphi$  introduit la variation hélicoïdale de la phase, tandis que les fonctions radiale et polaire contrôlent la distribution spatiale de l'amplitude. En pratique, selon la géométrie du système et le type de source acoustique considérée, d'autres formulations équivalentes peuvent être employées, notamment à l'aide de faisceaux de Bessel cylindriques ou par superposition de sources ponctuelles.

Dans un système expérimental discret, tel qu'un réseau de transducteurs, cette expression continue ne peut être réalisée que de manière approchée. La génération d'un vortex acoustique repose alors sur l'attribution de phases spécifiques à chaque élément du réseau, de manière à reproduire localement la dépendance angulaire  $\exp(im\varphi)$ .

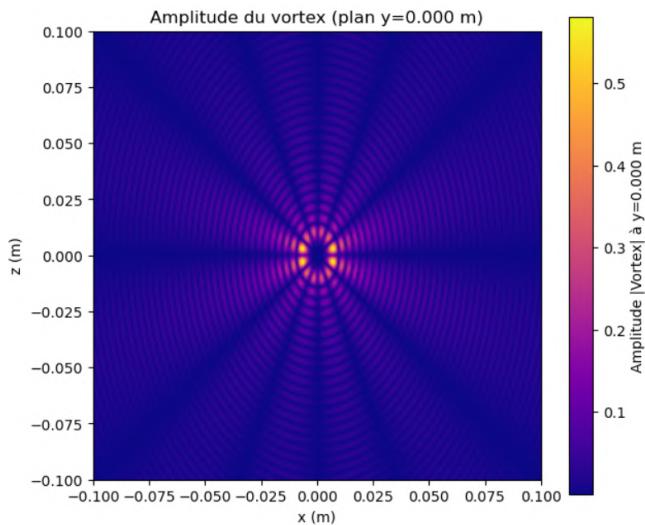


FIGURE 13 – Amplitude d'un vortex théorique (plan horizontal x,y) : A=1 / l=6 / m=1

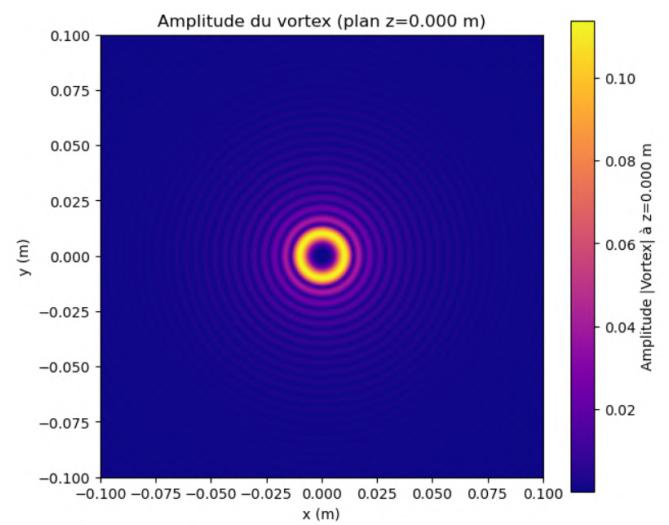


FIGURE 14 – Amplitude d'un vortex théorique (plan vertical x,z) : A=1 / l=6 / m=1

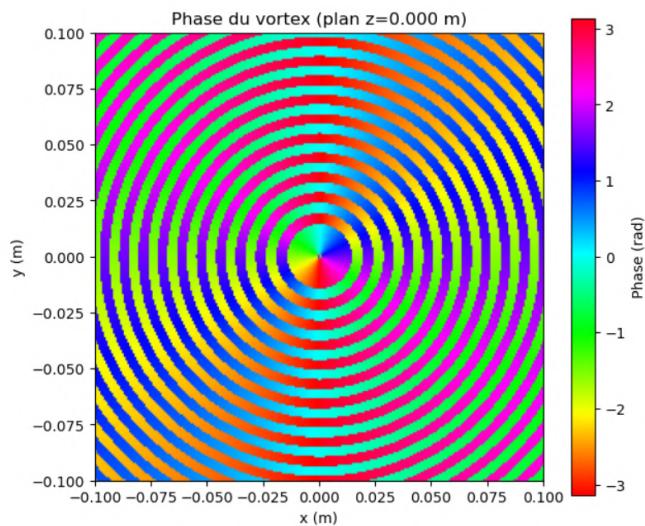


FIGURE 15 – Phase d'un vortex théorique (plan horizontal x,y) : A=1 / l=6 / m=1

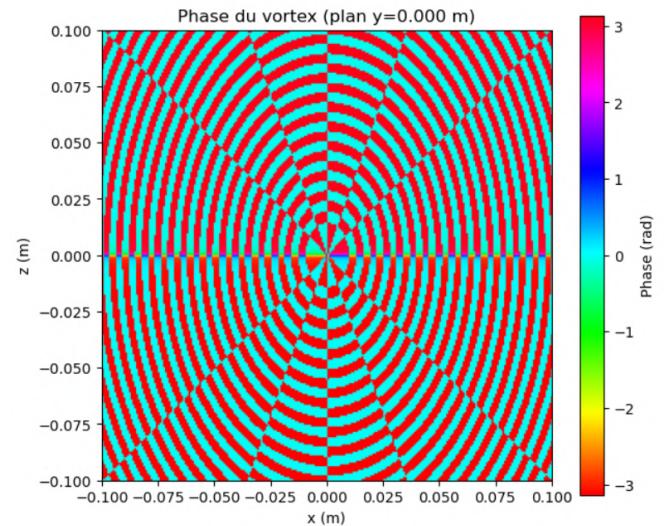


FIGURE 16 – Phase d'un vortex théorique (plan vertical x,z) : A=1 / l=6 / m=1

## I.3 Propriétés physiques des vortex acoustiques

Les vortex acoustiques possèdent plusieurs propriétés physiques distinctives qui les différencient des champs acoustiques plus conventionnels.

Premièrement, la présence d'une singularité de phase au centre entraîne une distribution annulaire de l'amplitude. L'intensité acoustique est minimale, voire nulle, au centre, et maximale sur un anneau dont le rayon dépend des paramètres du vortex, tels que la fréquence, la charge topologique et la géométrie de génération.

Deuxièmement, les vortex acoustiques sont associés à un moment angulaire orbital acoustique. Ce moment angulaire est directement lié à la charge topologique  $m$  et traduit la capacité du champ à transférer un couple à un objet placé dans le faisceau. Cette propriété, bien connue en optique, trouve un analogue direct en acoustique.

Troisièmement, du point de vue du piégeage de particules, la structure du champ induit des gradients de pression radiaux significatifs. Ces gradients peuvent générer des forces de radiation acoustique orientées vers ou loin de l'axe, selon les propriétés mécaniques de la particule, telles que sa densité et sa compressibilité. Les vortex acoustiques sont ainsi particulièrement intéressants pour le confinement latéral de particules.

## I.4 Implications pour le piégeage acoustique

L'intérêt des vortex acoustiques pour le piégeage réside principalement dans leur capacité à produire un confinement radial sans nécessiter de réflexion ou d'onde stationnaire. Cette propriété a motivé de nombreux travaux consacrés à la manipulation de particules par vortex acoustiques, notamment dans les domaines de la microfluidique et de la lévitation acoustique.

Néanmoins, la force de radiation est généralement plus faible que dans une onde stationnaire, et l'équilibre entre la force acoustique verticale et le poids de la particule n'est pas nécessairement assuré.

Ainsi, bien que les vortex acoustiques présentent une structure de champ riche et des propriétés physiques uniques, leur utilisation comme piège acoustique tridimensionnel stable nécessite des conditions particulières, telles qu'une intensité acoustique élevée, une géométrie adaptée ou l'introduction de réflexions supplémentaires. Ces limitations justifient l'analyse approfondie développée dans la suite de ce travail.

# II - Principe général de la génération de vortex

## II.1 Principe de génération d'un vortex par un réseau plan $8 \times 8$ de transducteurs

Dans ce travail, le vortex est généré à l'aide d'un réseau plan de transducteurs ultrasonores disposés selon une géométrie carrée  $8 \times 8$ .

Chaque transducteur est modélisé comme une source piston émettant une onde harmonique de fréquence  $f = 40\text{ kHz}$ . Le principe consiste à attribuer à chaque transducteur une phase spécifique  $\phi_i$  de manière à approximer un vortex acoustique.

## II.2 Réversibilité temporelle des équations acoustiques

Le principe fondamental à la base de la génération inverse d'un vortex acoustique repose sur la réversibilité temporelle des équations de l'acoustique linéaire. Dans un milieu homogène, isotrope et non dissipatif, l'équation d'onde acoustique est invariante par renversement du temps. Ainsi, si un champ de pression acoustique  $p(\vec{r}, t)$  est solution de l'équation d'onde, alors le champ  $p(\vec{r}, -t)$  constitue également une solution valide.

Cette propriété implique qu'un champ acoustique ayant évolué librement dans le milieu peut, en principe, être reconstruit en inversant sa propagation temporelle, à condition de disposer d'une information complète sur ce champ à une surface donnée.

Conceptuellement, on peut considérer que le champ cible  $p(\vec{r})$  est généré par une distribution continue de sources secondaires situées sur une surface fermée entourant la région d'intérêt. La propagation du champ depuis cette surface vers l'intérieur du domaine est décrite par les fonctions de l'équation d'onde acoustique.

Dans ce cadre, le champ acoustique en tout point du domaine peut être exprimé comme une superposition des contributions émises par chaque élément de la surface source, pondérées par leur amplitude complexe.

### **II.3 Principe de synthèse inverse**

Le principe de synthèse inverse consiste à inverser ce raisonnement. Au lieu de chercher le champ résultant d'une distribution de sources donnée, on impose comme donnée le champ cible  $p(\vec{r})$ , puis on calcule la valeur complexe que ce champ prend sur la surface où sont placés les transducteurs.

Dans le cas idéal, chaque point infinitésimal de la surface source est capable d'imposer exactement l'amplitude et la phase du champ complexe calculé. Si chaque transducteur de taille infinitésimal réemet alors une onde acoustique avec cette même amplitude et cette même phase, la superposition linéaire des ondes émises permet de reconstruire le champ cible dans la région d'intérêt.

Cette démarche est équivalente, dans le régime harmonique, à une opération de retournement temporel du champ acoustique.

### **II.4 Hypothèses du cas parfait**

La validité théorique de cette approche repose sur plusieurs hypothèses idéales :

- milieu de propagation linéaire, non dissipatif et non dispersif ;
- distribution continue de sources secondaires couvrant intégralement la surface de contrôle ;
- connaissance exacte du champ acoustique complexe cible ;
- contrôle parfait de l'amplitude et de la phase de chaque source.

Sous ces hypothèses, la synthèse inverse permet de recréer fidèlement un champ acoustique arbitraire, et en particulier un champ vortex, à partir d'une distribution appropriée de phases et d'amplitudes imposées aux transducteurs.

## **III - Synthèse inverse dans le cas réel : effets de la discréétisation et des contraintes expérimentales**

### **III.1 Discréétisation spatiale des sources**

Dans notre configuration, la surface source est constituée d'une ou deux plaques comportant chacune un nombre fini de transducteurs ultrasonores (64 éléments par plaque), chacun séparé par une distance finie (égale au diamètre des transducteurs soit 1 cm).

La distribution continue de sources supposée dans le cadre théorique est ainsi remplacée par un échantillonnage spatial discret du champ acoustique cible.

Cette discréttisation induit une perte d'information, en particulier dans les régions où la phase varie rapidement, comme à proximité de la singularité centrale d'un vortex acoustique.

La qualité du piège acoustique dépend également de la disposition des transducteurs, ainsi que de la position du piège par rapport à la grille discrète, selon si celui-ci se trouve au-dessus d'un transducteur ou alors entre deux, voire entre quatre transducteurs. Cela va directement impacter la précision du champ généré.

### **III.2 Quantification des phases et des amplitudes**

Une contrainte supplémentaire provient de la quantification des phases imposées aux transducteurs. Dans le dispositif utilisé, la phase de chaque transducteur ne peut être réglée que par pas finis, typiquement de l'ordre de  $\pi/5$ . Cette quantification empêche l'imposition exacte de la phase complexe calculée lors de la synthèse inverse. Des erreurs similaires peuvent également apparaître sur l'amplitude, selon les capacités de contrôle individuel des transducteurs.

### **III.3 Cas d'une seule plaque de transducteurs**

Dans le cas où la synthèse inverse est réalisée à l'aide d'une seule plaque de transducteurs, la surface de contrôle est nécessairement ouverte et ne permet pas d'encercler complètement la région d'intérêt. Cette configuration s'écarte fortement du cas idéal, dans lequel le champ est contrôlé sur une surface fermée.

La conséquence principale est une reconstruction partielle du champ acoustique cible. Les ondes émises ne peuvent interférer de manière optimale, ce qui conduit notamment à une focalisation asymétrique de l'énergie acoustique. Le champ reconstruit présente alors une dégradation de la structure théorique du vortex. On observe notamment la dégradation de la force verticale générée, avec l'apparition d'une zone de basse pression tout le long de l'axe vertical du vortex.

### **III.4 Cas de deux plaques opposées**

L'utilisation de deux plaques de transducteurs disposées face à face permet de se rapprocher davantage du cadre théorique idéal. Cette configuration améliore la couverture spatiale de la surface source et permet un meilleur contrôle du champ acoustique dans la région comprise entre les deux plaques.

Dans ce cas, les ondes émises par les deux réseaux peuvent interférer de manière plus symétrique, favorisant la reconstruction des structures de phase complexes du champ cible, en particulier pour les vortex acoustiques. Le confinement axial est renforcé et la qualité du piégeage acoustique s'en trouve améliorée.

Cependant, même dans cette configuration, la surface de contrôle reste discrète et limitée, et la reconstruction demeure une approximation du champ idéal.

### **III.5 Conséquences sur la reconstruction du vortex**

Malgré ces limitations, le principe de synthèse inverse reste valide : les transducteurs émettent une onde dont la distribution de phase et d'amplitude constitue une approximation discrète du champ cible. Par superposition, un champ présentant les caractéristiques globales d'un vortex acoustique est recréé dans la région d'intérêt, notamment la structure hélicoïdale de phase et le moment angulaire orbital associé.

Le champ reconstruit doit donc être interprété comme une approximation du champ théorique idéal, dont la fidélité dépend directement de la densité spatiale des transducteurs, de la résolution en phase et de la qualité de la synchronisation entre les sources.

### **III.6 Cadre de validité**

Cette approche reste pleinement pertinente tant que la longueur d'onde acoustique demeure grande devant l'espacement entre transducteurs, et que les erreurs de phase restent suffisamment faibles pour préserver la cohérence globale du champ. Dans ce régime, la synthèse inverse permet de générer expérimentalement des champs acoustiques complexes, tels que les vortex, avec un bon accord qualitatif avec la théorie.

## **IV - Notice d'utilisation du programme de calcul des phases d'un vortex acoustique**

Ces programmes permettent de calculer les phases et amplitudes à appliquer à chaque transducteur d'une plaque ultrasonore afin de recréer un champ acoustique vortex dans un point ou une région d'intérêt. Ils prennent en compte la position spatiale des transducteurs pour une synthèse inverse approximative du champ cible.

### **IV.1 Description générale du fonctionnement**

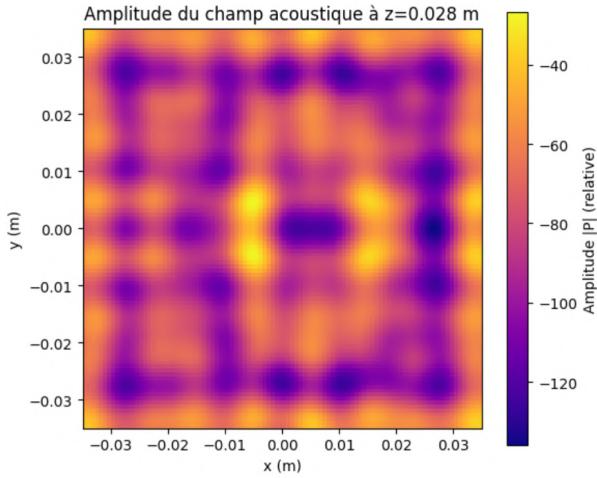
Les programmes sont tous construits comme suit :

1. Définition des paramètres physiques (paramètres liés aux transducteurs, liés à la particule, liés à l'air), puis création de l'espace de simulation (liste de position des transducteurs).
2. Fonction calculant les phases à transmettre aux transducteurs. Il s'agit simplement de calculer la valeur que prend le champ vortex aux positions des transducteurs (attention aux changements de référentiel).
3. Choix de la position du ou des vortex puis appel de la fonction.

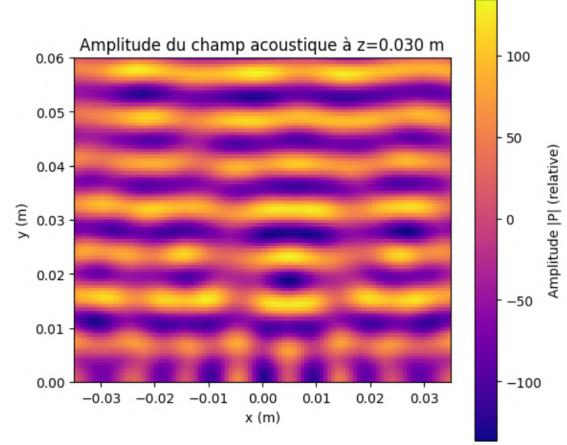
Il est à noter qu'il existe quatre variantes de ces programmes, en effet il existe des versions à une ou deux plaques de transducteur ainsi que des versions à un ou deux pièges. Cependant nous reviendrons sur ces programmes spécifiques dans la troisième phase.

## **V - Résultats et Limites**

Avec cette méthode on arrive effectivement à produire un piège permettant la lévitation. De plus, celui-ci est mieux localiser que ceux produit par la méthode optimizer. Pour la suite du projet il faut maintenant réussir à adapter ces programmes pour pouvoir faire léviter deux particule simultanément. Par ailleurs, il pourra être intéressant de chercher à augmenter la stabilité des pièges créés, notamment dans l'optique de déplacer des particules.



(a) Champs de pression - Méthode Vortex



(b) Champs de pression - Méthode Vortex

FIGURE 17 – champs de pression produit par la méthode Vortex

### 3.2.4 Programmes divers

Il nous était nécessaire de pouvoir visualiser les résultats, du moins théorique, de nos divers programmes de calcul de phases. Pour ce faire nous avons créé des programmes qui permettent de visualiser des coupes verticales ou horizontales des champs de pression produit par des sets de phases passés en paramètre. Ses programmes s'appuient sur un modèle d'onde piston circulaire et sont l'effet de chaque transducteur afin de calculer le champ de pression.

Par ailleurs, nous avons également créé des programmes permettant de visualiser l'évolution de la force créée par le champ de pression selon un axe vertical ou horizontal. Ce programme s'appuie directement sur les méthodes développées pour la méthode optimizer. En effet, il était déjà nécessaire dans celle-ci de calculer la force produite par le champ de pression en un point.

### 3.3 Notice de montage du lévitateur à une plaque

#### 3.3.1 Étapes clés

1. Liste du matériel et commande
2. Conception, impression et montage du prototype 2
3. Soudure des composants et montage des fils
4. Vérification continuité des fils
5. Montage des transducteurs sur la plaque
6. Pin Assignment

#### 3.3.2 Liste du matériel et commande

Pour réaliser notre prototype, qui est une matrice de transducteurs, nous avons réalisé plusieurs commandes dont la liste du matériel se trouve dans les pages suivantes.

Cette commande ainsi que celles des deux autres phases sont détaillées ici : [\*Matériel ultrasonic array - Google Sheets\*](#)

Nous avons aussi utilisé du matériel complémentaire de l'école :

1. Oscilloscope ;
2. Micro de mesure de la marque Brüel & Kjaer ;
3. Sondes de mesure ;
4. Fer à souder conventionnel ;
5. Manipulateur de composants CMS.

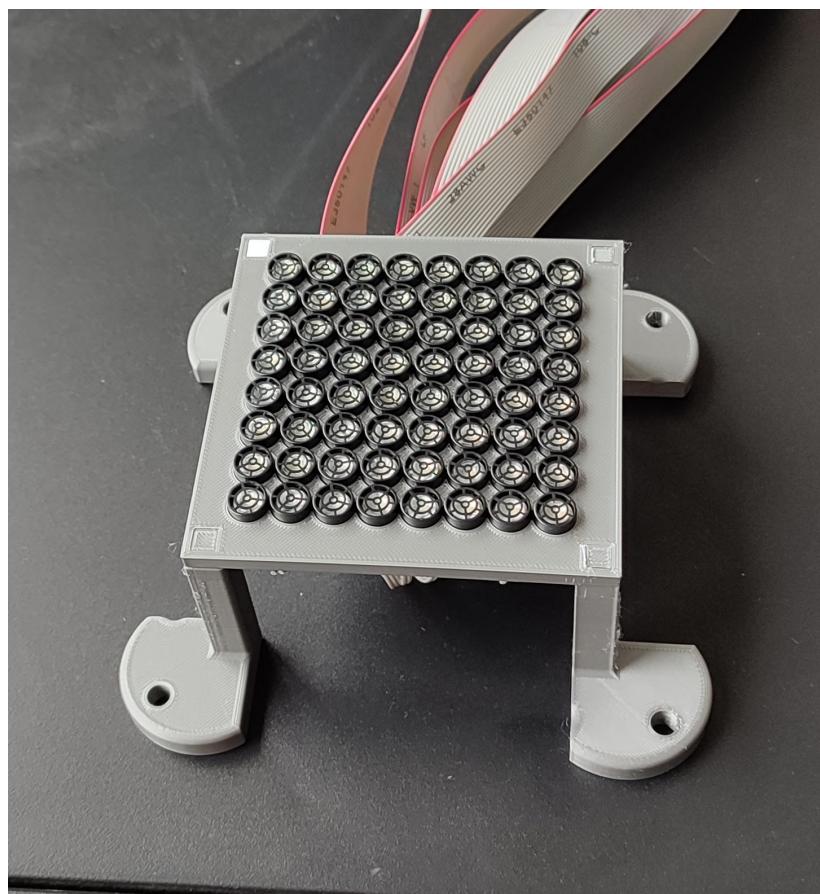


FIGURE 18 – Photo de notre prototype une plaque

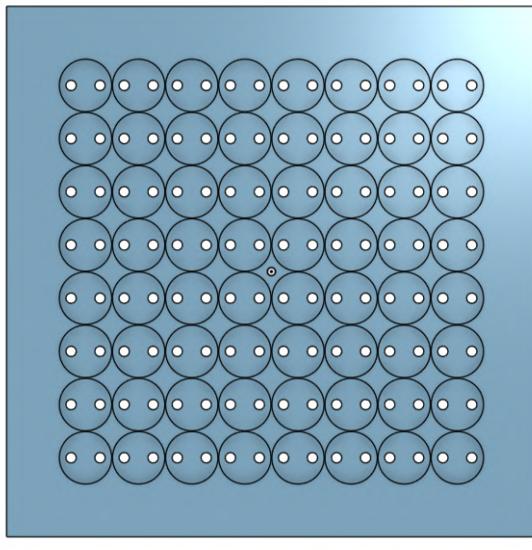
# Commande Composants – Projet de Lévitation Acoustique (Phase 3)

Référence	Réf fabricant	Lien
<b>RS-ONLINE</b>		
Condensateur céramique multicouche MLCC, CMS, $0.1\mu\text{F}$ , 100V c.c., X7R	GCJ21BR72A104KA01K	<a href="#">Lien</a>
Condensateur céramique multicouche MLCC, CMS, $4.7\mu\text{F}$ , 50V c.c., X5R	GRM21BR61H475KE51L	<a href="#">Lien</a>
Connecteur IDC Stelvio Kontek Femelle, 3 contacts, 1 rangée, pas 2.54mm	661003151922	<a href="#">Lien</a>
Embases de circuit imprimé Molex, C-Grid, 16 pôles, 2.54mm, 2 rangées	70246-1601	<a href="#">Lien</a>
Connecteur IDC RS PRO Femelle, 16 contacts, 2 rangées, pas 2.54mm	284-6492	<a href="#">Lien</a>
Embase à broches Samtec TLW, 6 pôles, 2.54mm, 1 rangée, Droit	TLW-106-05-G-S	<a href="#">Lien</a>
Embase à broches Samtec TLW, 8 pôles, 2.54mm, 1 rangée, Droit	TLW-108-05-G-S	<a href="#">Lien</a>
Embase à broches Samtec TSW, 6 pôles, 2.54mm, 1 rangée, Angle droit	TSW-106-08-L-S-RA	<a href="#">Lien</a>
Câbles en nappe RS PRO 16 voies, pas de 1.27mm, 28 AWG, Gris	289-9874	<a href="#">Lien</a>
Transistor MOSFET IX4427NTR, 8 broches, SOIC	IX4427NTR	<a href="#">Lien</a>
Connecteur femelle pour CI, 6 contacts, 1 rangée, Angle droit	613006143121	<a href="#">Lien</a>
Embase à broches Samtec TLW, 18 pôles, 2.54mm, 2 rangées, Droit	TLW-109-05-G-D	<a href="#">Lien</a>
<b>CONRAD</b>		
Connecteur basse tension embase femelle horizontale	FC68149	<a href="#">Lien</a>
<b>ROBOTSHOP</b>		
Kit Tinylev (64 transducteurs acoustiques)	ORK328ALKRB	<a href="#">Lien</a>

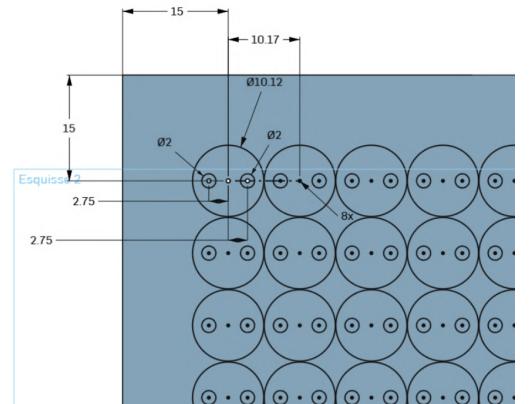
### 3.3.3 Conception et montage du prototype 2

Pour ce prototype 2, qui doit se baser sur une plaque de transducteurs émettant des ondes progressives, nous avons pensé à créer une matrice plane de 8x8 transducteurs. Pour la conception, nous différencierons donc la plaque comprenant la matrice, et les pieds.

Pour la plaque, c'est un carré de 100,7 mm de côté et de 5 mm d'épaisseur. À l'intérieur de la plaque, il y a 64 emplacements pour les transducteurs, c'est-à-dire 64 trous de 10,12 mm de diamètre, séparés de 10,17 mm (distance entre les centres). Ces trous sont creusés de 4,5 mm et deux petits trous sont percés entièrement pour accueillir les pattes des transducteurs. Nous avons utilisé un gabarit avec différentes tailles de trous pour mettre au point la taille idéale qui est donc de 10,12 mm.



(a) Plaque seule - Onshape



(b) Zoom esquisse - Onshape

FIGURE 19 – Modélisation CAO du socle

On crée également des emplacements sur l'autre face de la plaque pour insérer les 4 pieds, qui sont imprimés à part.

*Note : Nous avons modifié la façon de mettre les pieds au cours du projet donc certaines photos de ce document montrent un prototype qui n'utilise pas ce montage de pieds.*

Pour obtenir le montage final, on imprime en 3D la plaque et les 4 pieds et on assemble le tout.

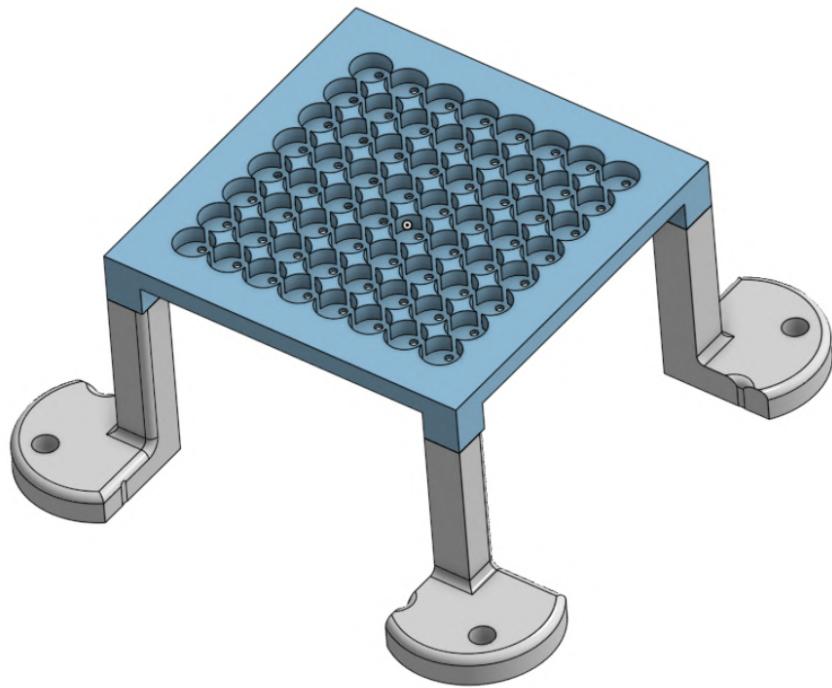
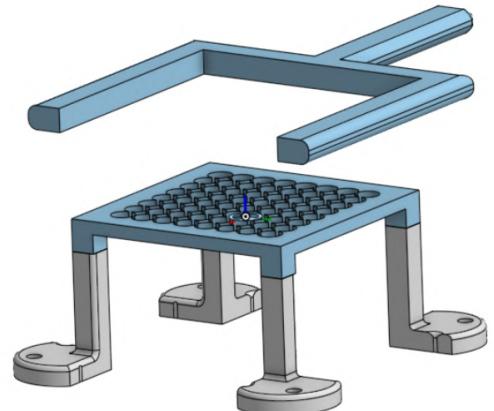


FIGURE 20 – Conception du prototype 2 - Onshape

Pour placer une particule dans le piège au dessus des transducteurs, nous avons imprimé un outil, une pince avec un film pour porter la particule et la placer à l'endroit souhaité. Ce qui est beaucoup plus facile d'utilisation qu'une pince traditionnelle ou en pincant avec les doigts.



(a) Pince



(b) Montage avec la pince - Onshape

FIGURE 21 – Conception de la pince

Nous avons tout imprimé en impression 3D sur les imprimantes Prusa de l'école.

Il convient de noter que ces impressions sont particulièrement longues donc doivent être prévues à l'avance.

### 3.3.4 Soudure du PCB et ajout des connecteurs sur les nappes de câbles

#### Introduction

Cette partie vise à présenter les étapes de réalisation de la partie électronique nécessaire à l'obtention d'une plaque de lévitation : soudure des différents éléments sur le PCB et sertissage des connecteurs sur les câbles en nappe. L'ensemble des éléments présentés dans cette partie sont réalisés sur la base des travaux d'Asier MARZO disponible sur ce repository [GitHub](#), notamment via l'usage du PCB mis à disposition à cette [adresse](#).

Voici pour commencer un schéma avec une vue de dessus du PCB utilisé :

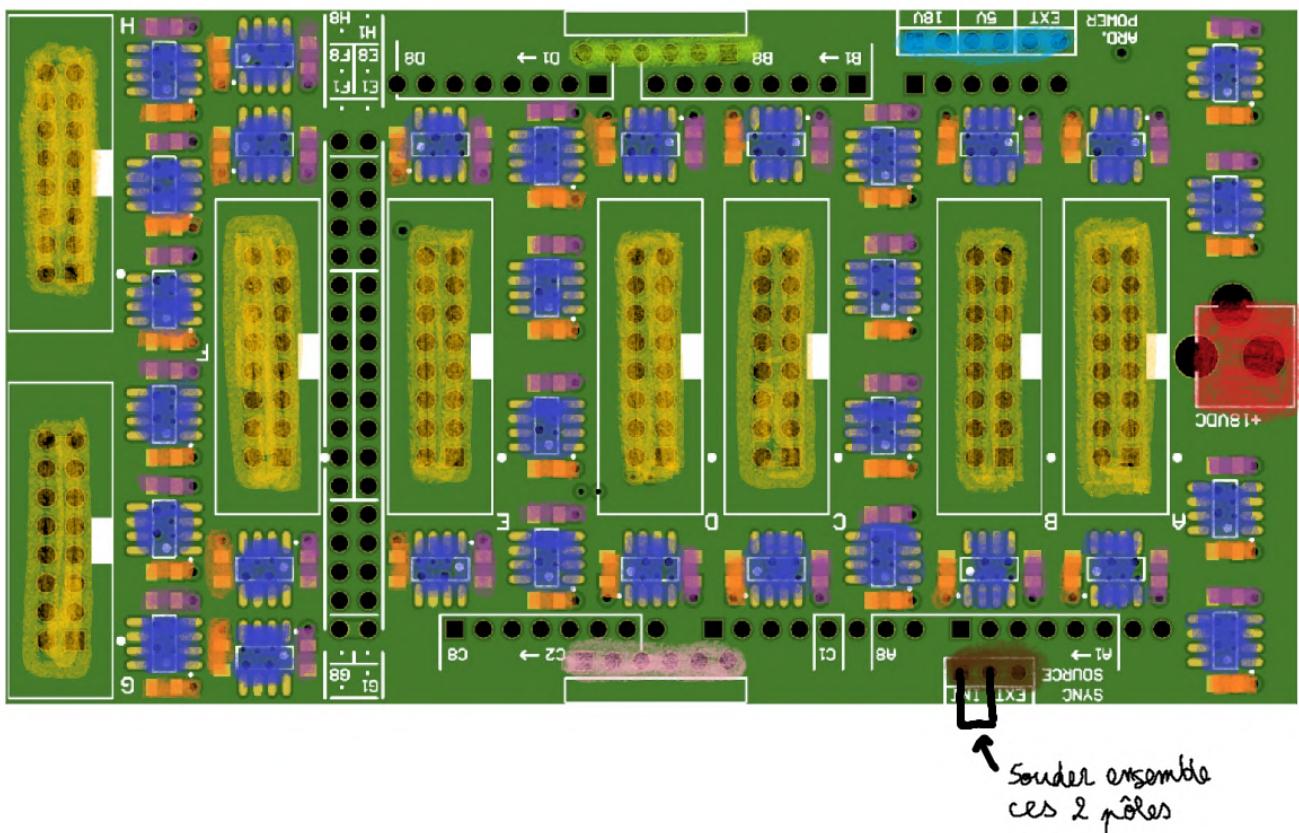


FIGURE 22 – Schéma de l'avant de la carte issu de la documentation [Ultraino](#) et annoté par nos soins.

Légende des composants :

- Condensateur  $0,1\mu F$
- Condensateur  $4,7\mu F$
- Driver TC4427CPA
- Connecteur basse tension embase femelle horizontale
- Barrette mâle à 16 pôles
- Barrette mâle coudée à 6 pôles
- Barrette mâle à 6 pôles
- Barrette mâle à 3 pôles
- Barrette femelle coudée à 6 pôles

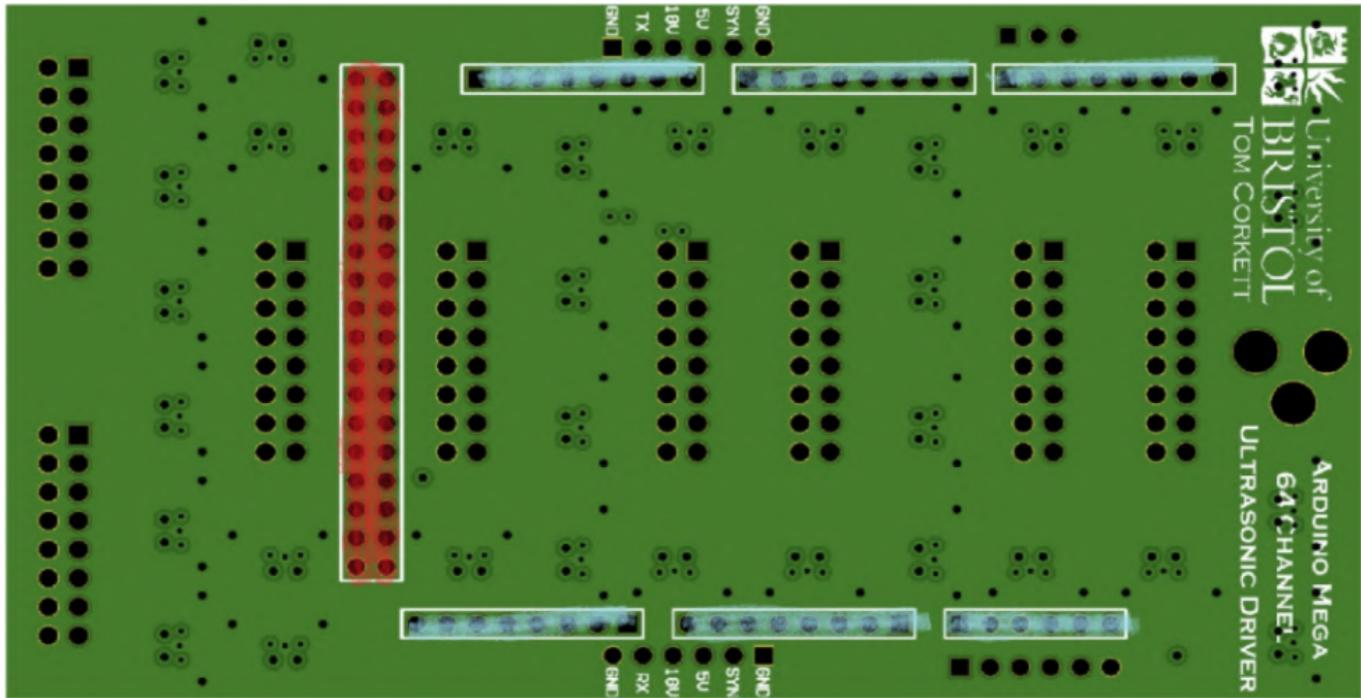


FIGURE 23 – Schéma de l'arrière de la carte issu de la documentation [Ultraino](#) et annoté par nos soins.

Composants arrière :

- Barrette mâle 2 rangées de 18 pôles
- Barrette mâle 1 rangée de 8 pôles

## Soudure CMS

1. La première étape consiste à déposer sur chaque piste de la pâte à soudure CMS à l'aide d'un manipulateur de composants CMS ;
2. Ensuite, il faut à nouveau utiliser le manipulateur afin de déposer, cette fois-ci, les composants CMS (drivers TC4427 et condensateurs) en veillant à respecter leur positionnement ;
3. Il ne reste alors plus qu'à passer le PCB au four à refusion afin de réaliser la soudure entre la carte électronique et les composants.

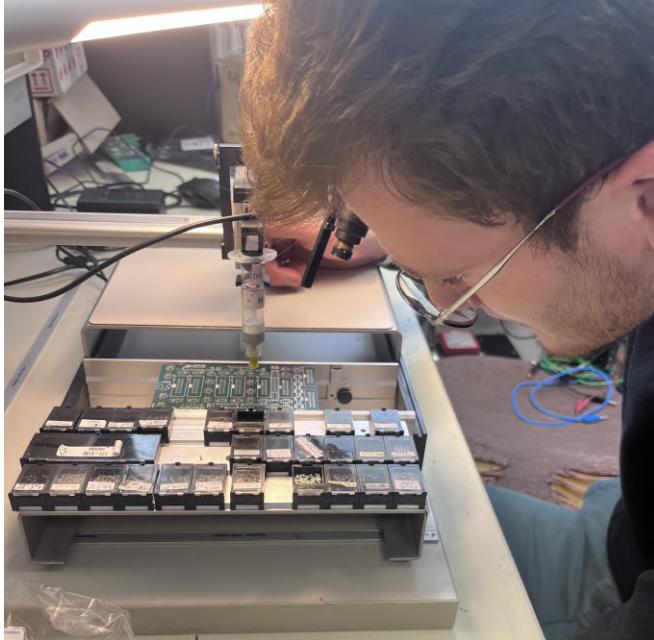


FIGURE 24 – Soudure CMS à l’atelier d’électronique

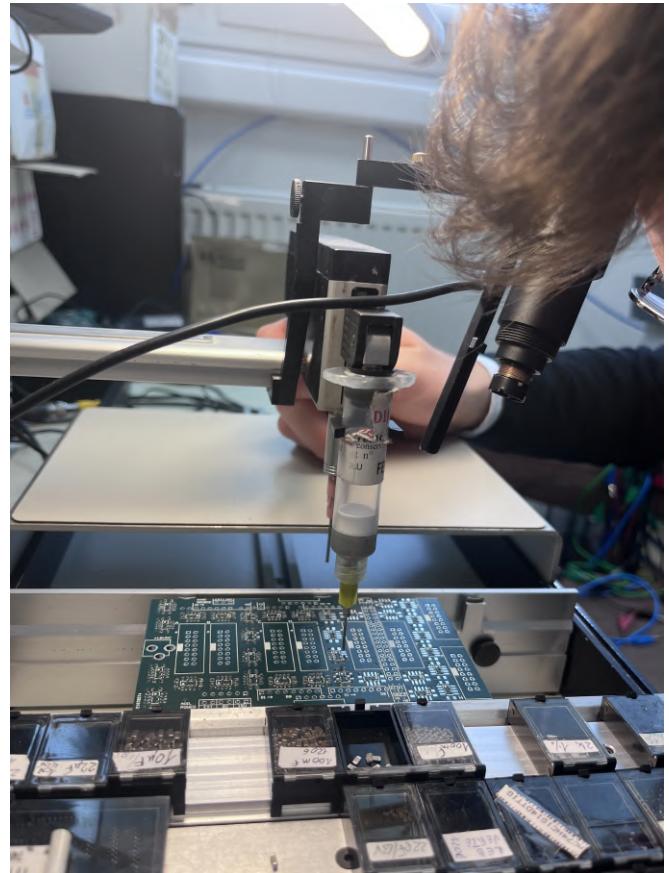


FIGURE 25 – Soudure CMS à l’atelier d’électronique

## Soudure conventionnelle

L’ensemble connecteurs étant composés de plastique, il n’est pas possible de les souder en utilisant la méthode CMS, il faut donc utiliser une méthode conventionnelle, à l’aide d’un fer à souder.

Pour réaliser cette étape, il suffit de positionner les différents connecteurs aux emplacements précisés par les figures 22 & 23. Le résultat final est présenté en figure 26.

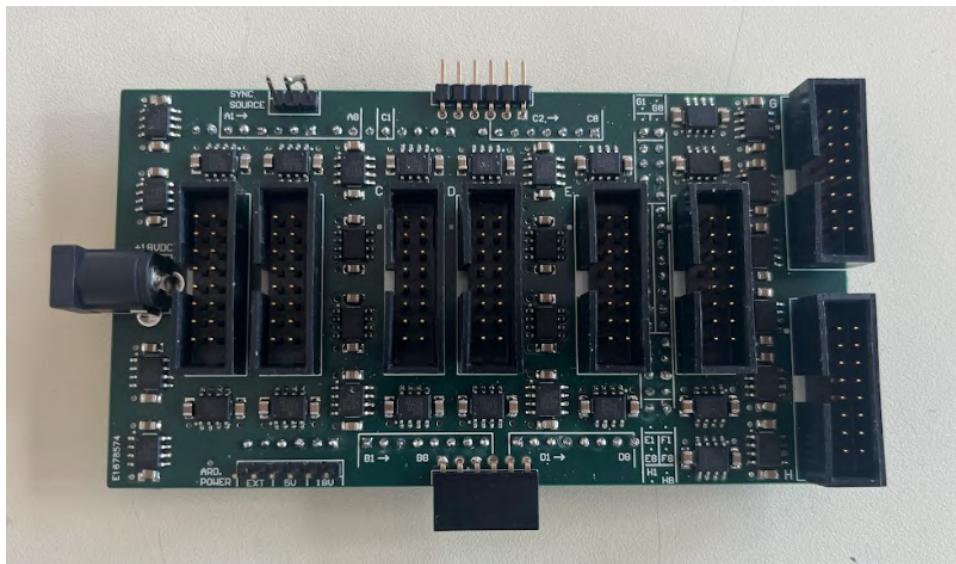


FIGURE 26 – Carte finalisée

## Sertissage des câbles en nappe

- Découpe de 8 bandes de 60 cm de câble en nappe ;
- Sertissage des connecteurs femelles à 16 pôles d'un côté.



FIGURE 27 – Connecteur femelle serti

- Séparation des fils et sertissage des connecteurs IDC :

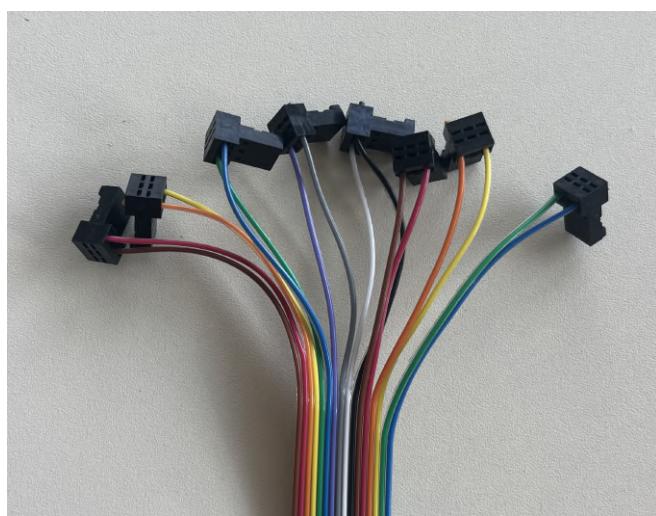


FIGURE 28 – Séparation des fils

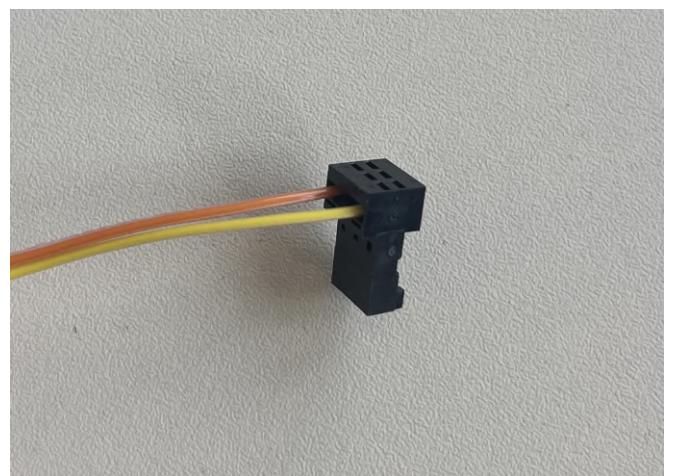
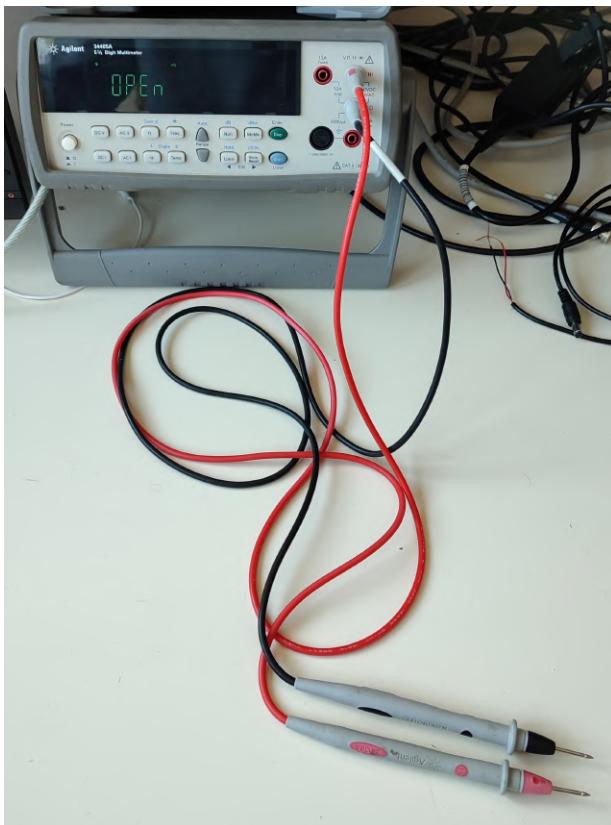


FIGURE 29 – Connecteur IDC

### 3.3.5 Vérification de la continuité du courant dans les fils avec un multimètre

L'objectif de cette étape est de vérifier le continuité du courant aux bornes des 2 connecteurs d'un câble en nappe afin de s'assurer de la bonne réalisation des connexions. Pour ce faire, il convient d'utiliser un multimètre en mode continuité :



(a) Branchement pour vérifier la continuité



(b) Mode continuité sur l'oscilloscope

FIGURE 30 – Montage de vérification de la continuité

### 3.3.6 Montage des transducteurs sur la plaque

Tout d'abord, il est nécessaire de tester la polarité de chacun des transducteurs utilisé.

#### Repérage de la polarité

##### Matériel nécessaire :

- Multimètre ;
- Feuilles de cuivre ;
- Transducteurs ;
- Câbles et pinces crocodiles.

#### Étape 1

Connecter à l'aide de fils, une feuille de cuivre à la sortie positive et une autre au zéro du multimètre (comme nous pouvons le voir sur la photo ci-dessous).

**Instruction :** Mettre le multimètre en mode *tension continue* au niveau le plus précis.

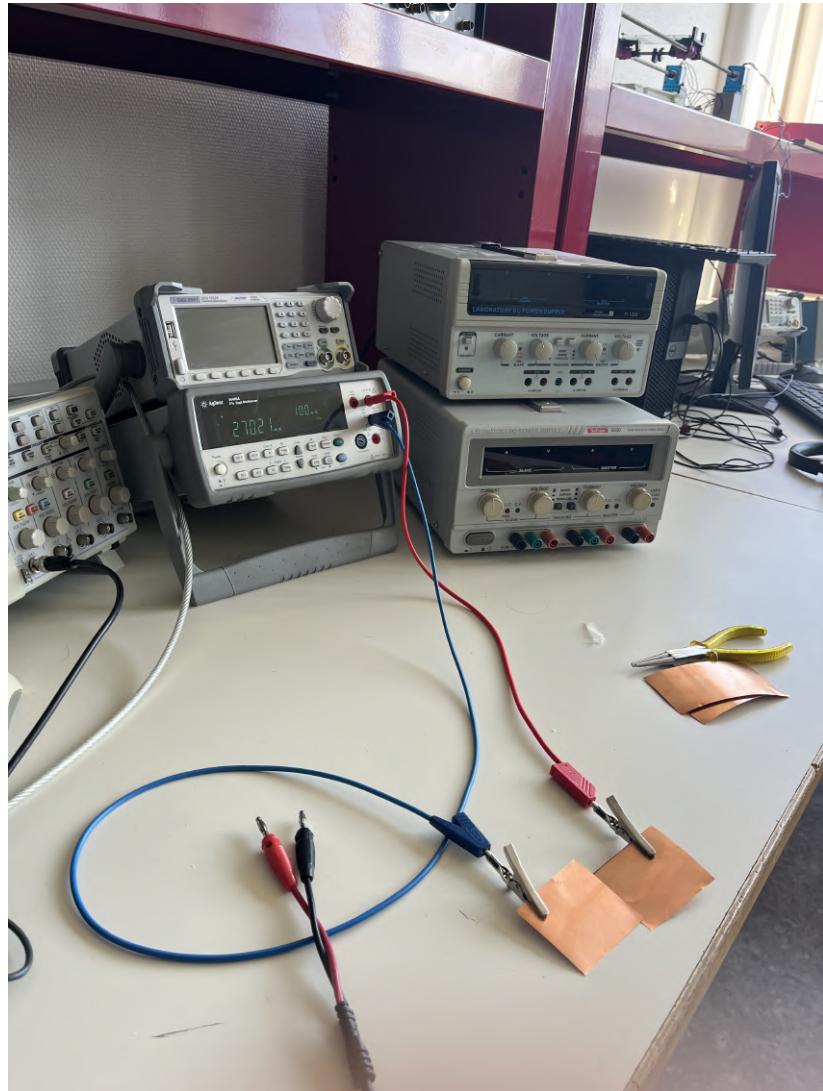


FIGURE 31 – Test de la polarité des transducteurs

### Étape 2

Poser les pattes du transducteur sur une feuille de cuivre chacune.

**Attention :** Veillez à ne pas toucher les pattes avec vos doigts durant la manipulation.

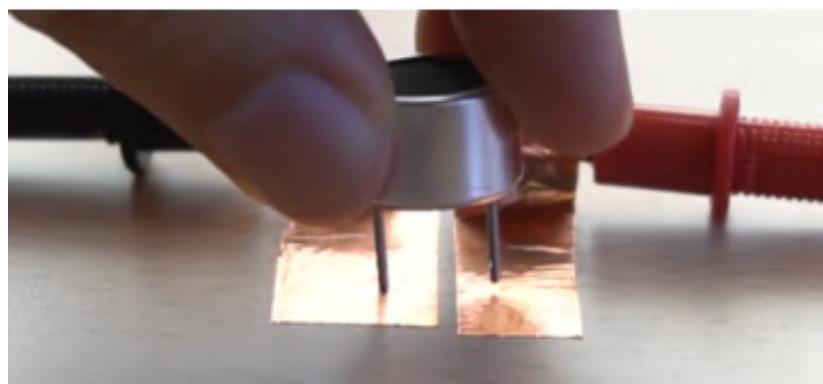


FIGURE 32 – Pose du transducteur à tester

### Étape 3

Regarder l'affichage du multimètre :

- **Si la tension est négative** : Mettre une marque du côté ou sur la patte qui touche la feuille reliée au **zéro** du multimètre ;
  - **Si la tension affichée est positive** : Mettre une marque sur la patte qui touche la feuille de cuivre reliée à la sortie **positive** du multimètre.
- 
- Mettre en position les transducteurs sur le support et les clipser aux connecteurs IDC.



FIGURE 33 – Clipser les transducteurs

- Il est alors possible d'ajouter les pieds pour poser la structure

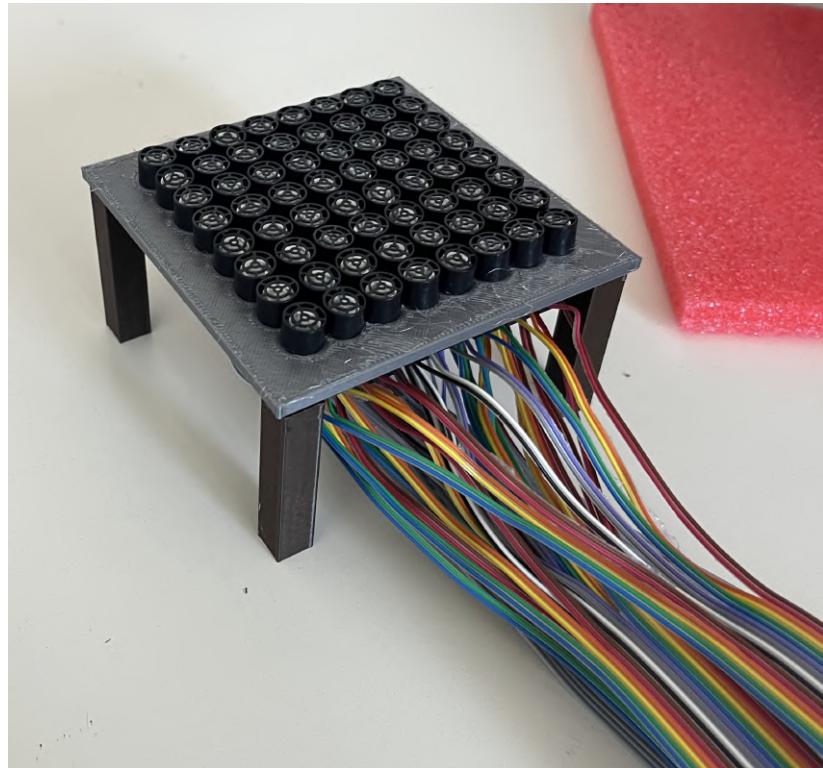


FIGURE 34 – Ajout des pieds du support

- Connecter l'autre extrémité des câbles en nappe à la carte d'amplification précédemment réalisée.

### 3.3.7 Assignation des pins

Le pilotage individualisé des transducteurs suppose d'être capable d'envoyer le bon signal au bon transducteur. Pour ce faire, il est nécessaire de réaliser un adressage (ou *pin assignment*) afin de faire le lien entre un pin de la carte d'amplification, et donc un transducteur, et le software. Nous présentons ici deux méthodes permettant de réaliser cette étape.

#### Méthode des déphasages

L'idée de cette méthode est d'envoyer un signal déphasé sur un pin de la carte Arduino (par exemple le 1) et trouver, en sortie de la carte d'amplification, l'endroit où le signal est déphasé :

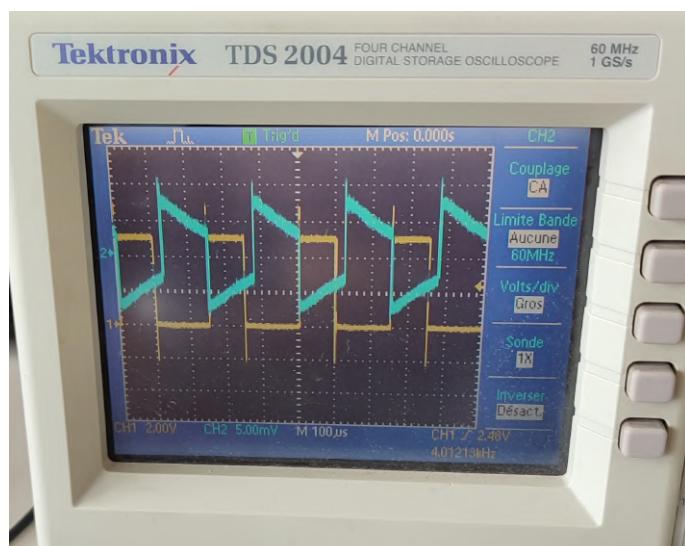


FIGURE 35 – Pin Assignment

Bien que fonctionnelle, cette étape est assez fastidieuse, répétitive et peut être source d'erreurs.

## Utilisation du hardware

Une autre méthode assez intuitive consiste à partir du principe que l'on connaît, grâce aux plans du PCB, les connexions entre l'Arduino et les différentes sorties de la carte d'amplification. Il suffit alors d'associer à chaque sortie de la carte un pin physique de l'Arduino.

Une fois cette étape réalisée, il ne reste qu'à faire le lien entre les pins physiques de l'Arduino et ses ports informatiques à l'aide de la datasheet dont un extrait est ci-après présenté en figure 36.

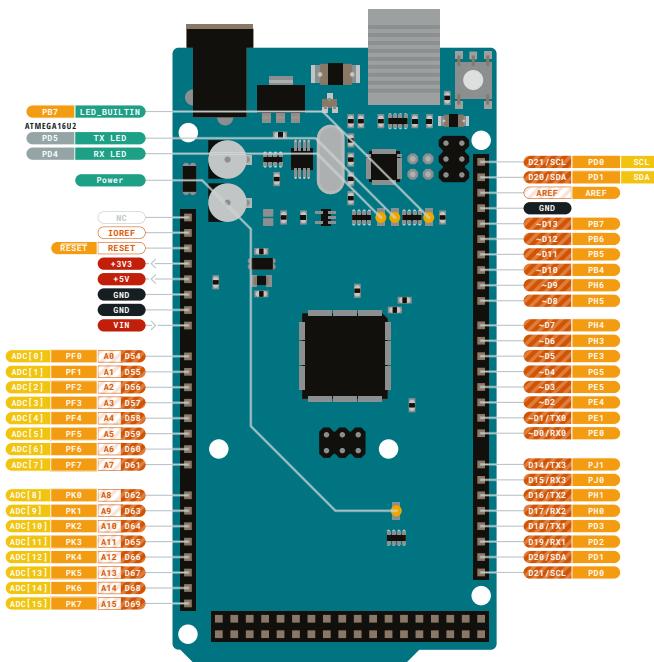


FIGURE 36 – Ports de l'Arduino Mega - *Source : Documentation Arduino*

Les relations obtenues sont données dans le document suivant : [PinAssignment](#)

## 3.4 Notice d'utilisation du lévitateur à une plaque

### 3.4.1 Codes Arduino

Utiliser le prototype 2 nécessite de bien comprendre les codes Arduino nécessaires à l'envoi des phases. Notre code NewDriverMega étant largement issu du code DriverMega disponible dans la littérature de Marzo, nous consacrons un paragraphe au code DriverMega.

#### Code DriverMega

Nous avons réalisé une documentation détaillée sur le fonctionnement du code DriverMega utilisé pour la communication entre l'application Java et notre modèle matériel.

Voici le lien de consultation : [Explication DriverMega](#)

Vous trouverez également une copie de ce document en Annexe 7.1.

## Code NewDriverMega

Le code DriverMega ayant été rédigé afin d'être utilisé avec le logiciel Java du projet [Ultraino](#) de Marzo, de nombreuses fonctions / variables y sont superflues. En effet, notre utilisation de ce code ne nécessite pas d'envoyer les phases en temps réel, ce pourquoi il était précédemment optimisé.

Nous avons donc décidé de le recoder dans son intièreté et en voici les *explications* à comparer avec l'ancien code. Nous avons tout recopié ici afin d'éviter la perte d'informations.

Voici le code étudié :

```
1 #include <avr/sleep.h>
2 #include <avr/power.h>
3
4 #define N_PATTERNS 2
5 #define N_PORTS 10
6 #define N_DIVS 10
7 #define PATTERN_PERIODS 400000 // nombre de periodes 40 kHz avant de passer au
     pattern suivant (env. 10s)
8
9 #define WAIT() __asm__ __volatile__("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t"
     "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t")
10
11 #define OUTPUT_WAVE(pointer, d) \
12 PORTA = pointer[d * N_PORTS + 0]; \
13 PORTC = pointer[d * N_PORTS + 1]; \
14 PORTL = pointer[d * N_PORTS + 2]; \
15 PORTB = pointer[d * N_PORTS + 3]; \
16 PORTK = pointer[d * N_PORTS + 4]; \
17 PORTF = pointer[d * N_PORTS + 5]; \
18 PORTH = pointer[d * N_PORTS + 6]; \
19 PORTD = pointer[d * N_PORTS + 7]; \
20 PORTG = pointer[d * N_PORTS + 8]; \
21 PORTJ = pointer[d * N_PORTS + 9];
22
23 static const byte patterns[N_PATTERNS * N_DIVS * N_PORTS] = {
24     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
25     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
26     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
27     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
28     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
29     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
30     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
31     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
32     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
33     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
34
35     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
36     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
37     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
38     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
39     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
40     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
41     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
42     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
43     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
44     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
45 };
46
```

```

47 static byte bufferA[N_DIVS * N_PORTS];
48 static byte bufferB[N_DIVS * N_PORTS];

```

Listing 1 – Programme NewDriverMega - Partie 1

```

1 void setup() {
2     DDRA = DDRC = DDRD = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ = 0xFF;
3     PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD = PORTG = PORTJ =
4         0x00;
5
6     for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {
7         bufferA[i] = 0;
8         bufferB[i] = 0;
9     }
10
11    for (int i = 0; i < (N_PORTS * N_DIVS/2); ++i) {
12        bufferA[i] = 0xFF;
13    }
14
15    pinMode(2, OUTPUT);
16    noInterrupts();
17    TCCR3A = bit(WGM10) | bit(WGM11) | bit(COM1B1);
18    TCCR3B = bit(WGM12) | bit(WGM13) | bit(CS10);
19    OCR3A = (F_CPU / 40000L) - 3;
20    OCR3B = (F_CPU / 40000L) / 2;
21    interrupts();
22
23    pinMode(3, INPUT_PULLUP);
24
25    ADCSRA = 0;
26    power_adc_disable();
27    power_spi_disable();
28    power_twi_disable();
29    power_timer0_disable();
30    power_usart1_disable();
31    power_usart2_disable();
32    power_usart3_disable();
33
34    bool useA = true;
35    unsigned long currentPeriods = 0;
36    byte currentPatternIndex = 0;
37
38    byte* activeBuffer = bufferA;
39    byte* nextBuffer = bufferB;
40
41    for(int i=0; i < N_PORTS * N_DIVS; i++)
42        bufferA[i] = patterns[i];
43
44    for(int i=0; i < N_PORTS * N_DIVS; i++)
45        bufferB[i] = patterns[(currentPatternIndex + 1) * N_PORTS * N_DIVS + i];
46
47    byte* emittingPointerH = &activeBuffer[0];
48    byte* emittingPointerL = &activeBuffer[N_PORTS * N_DIVS / 2];

```

Listing 2 – Programme NewDriverMega - Partie 2 (setup)

```

1 LOOP:
2     while(PINE & 0b00100000);
3
4     OUTPUT_WAVE(emittingPointerH, 0); WAIT();
5     OUTPUT_WAVE(emittingPointerH, 1); WAIT();
6     OUTPUT_WAVE(emittingPointerH, 2); WAIT();
7     OUTPUT_WAVE(emittingPointerH, 3); WAIT();
8     OUTPUT_WAVE(emittingPointerL, 0); WAIT();

```

```

9  OUTPUT_WAVE(emittingPointerL, 1); WAIT();
10 OUTPUT_WAVE(emittingPointerL, 2); WAIT();
11 OUTPUT_WAVE(emittingPointerL, 3); WAIT();
12
13 currentPeriods++;
14 if(currentPeriods >= PATTERN_PERIODS) {
15     currentPeriods = 0;
16     useA = !useA;
17     activeBuffer = useA ? bufferA : bufferB;
18     nextBuffer = useA ? bufferB : bufferA;
19     currentPatternIndex++;
20     if(currentPatternIndex >= N_PATTERNS) currentPatternIndex = 0;
21     emittingPointerH = &activeBuffer[0];
22     emittingPointerL = &activeBuffer[N_PORTS * N_DIVS / 2];
23     int nextPatternIndex = currentPatternIndex + 1;
24     if(nextPatternIndex >= N_PATTERNS) nextPatternIndex = 0;
25     for(int i=0; i < N_PORTS * N_DIVS; i++)
26         nextBuffer[i] = patterns[nextPatternIndex * N_PORTS * N_DIVS + i];
27 }
28 goto LOOP;
29 }
30 void loop() {}

```

Listing 3 – Programme NewDriverMega - Partie 3 (boucle principale)

Pour la lisibilité du présent document, vous trouverez les explications des différents changements entre DriverMega et NewDriverMega en Annexe.

### 3.4.2 Mise en Route et Utilisation

#### 1. Préparation et Notice

Avant toute manipulation, consultez la [notice de montage](#).

#### 2. Connexions initiales

Branchez les câbles de l'Arduino (câble bleu) et l'alimentation de la carte (câble noir).

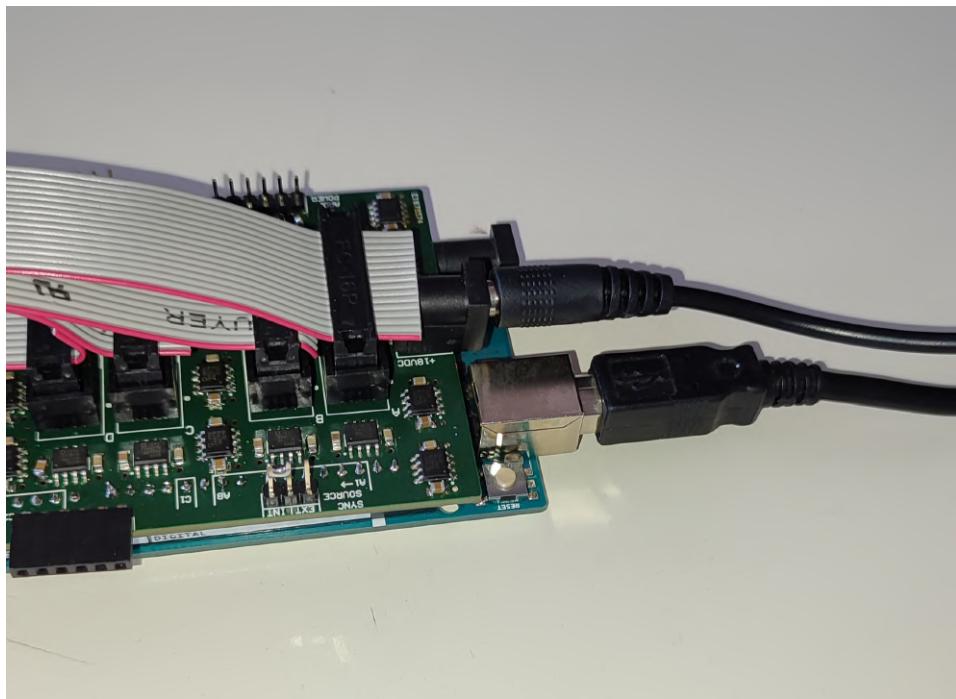


FIGURE 37 – Branchements de la carte Arduino et de la carte d'amplification

### 3. Mise sous tension

Allumez l'alimentation et branchez le fil noir sur CH2.

- L'intensité doit être basse automatiquement (sans intervention).
- **Attention :** Ne jamais dépasser 29V.
- Le bouton du courant ne doit pas dépasser la position “12h”.



FIGURE 38 – Configuration de l'alimentation de la carte d'amplification

#### 4. Récupération et mise à jour des phases

- Choix des phases : deux méthodes sont possibles :
  - (a) Utiliser les phases pré-calculées avec la méthode vortex pour 1 piège dans ce *Document Excel* pour différentes valeurs de x et z = 2.5cm. Dans ce document, 1 est l'ordre du vortex.
  - (b) Utiliser nos programmes de calcul si le point souhaité n'est pas dans le fichier Excel :
    - Méthode Optimizer pour 1 piège : [Lien Google Colab Optimizer 1 pièce](#) ;
    - Méthode Optimizer pour 2 pièges : [Lien Google Colab Optimizer 2 pièges](#) ;
    - Méthode vortex pour 1 piège : [Lien Google Colab Vortex 1 pièce](#) ;
    - Méthode vortex pour 2 pièges : [Lien Google Colab Vortex 1 pièce](#)
- Ouvrez sur Google Colab le fichier *Passage\_Phases\_python\_pin.ipynb* également disponible en Annexe 7.3.

5. Lancez le programme de passage des phases réelles en octets et copiez les octets générés (bit buffer).

6. Ouvrez le sketch Arduino disponible en Annexe 7.4; et remplacez les octets à la ligne 24.

Vous nous direz alors en essayant qu'il y a une erreur, et vous aurez en partie raison : en effet, le code Arduino fourni ici est codé pour 2 patterns !!

Donc deux options :

- (a) Vous voulez faire voler la bille à un endroit précis sans déplacement et alors vous copiez-collez le même set de phase deux fois (vous entendez un bruit toutes les 10 secondes si vous ne modifiez pas *PATTERN\_PERIODS* signifiant que vous passez au pattern suivant, mais comme vous avez mis deux fois le même, aucun changement visuel) ;
- (b) Vous souhaitez faire déplacer la bille et vous n'avez alors qu'à mettre deux set de phases différents ! Il convient tout de même de faire attention à ce que les deux sets de phase permettent à la bille de voler à deux endroits proches, sans quoi elle tombera.

## 7. Téléversement

Dans l'IDE Arduino, sélectionnez la carte Arduino Mega, puis téléversez et lancez le programme.

## 8. Mise en lévitation

À l'aide de la pince, placez délicatement la bille à l'endroit précis correspondant à la position choisie pour le piège acoustique dans vos phases.

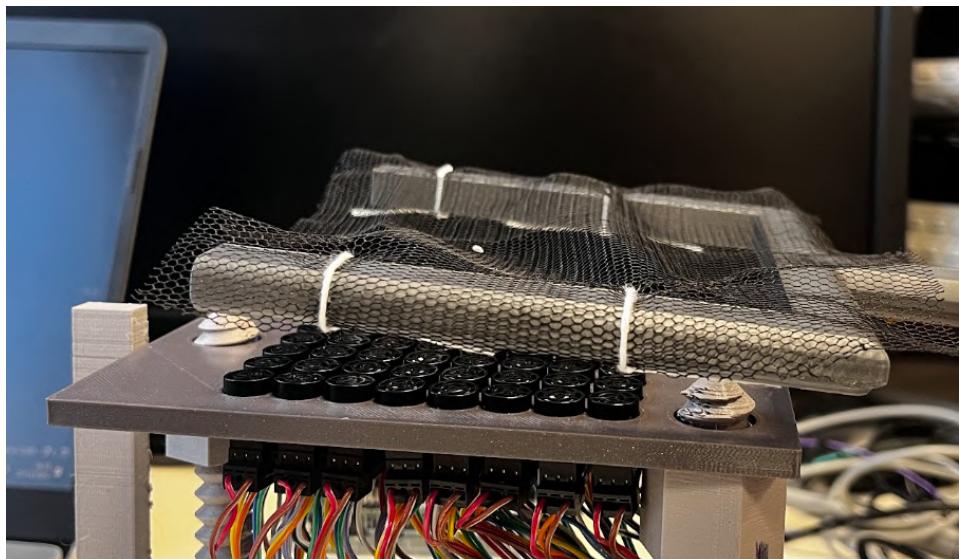


FIGURE 39 – Mise en lévitation grâce à la pince

## 9. Procédure d'arrêt

Pour éteindre le système : éteignez d'abord l'alimentation, puis débranchez les fils.

### Connexion de l'alimentation sur DriverBoard

Une fois que c'est fait, vous pouvez à priori utiliser votre carte en l'alimentant comme indiqué sur la photo ci-dessous :

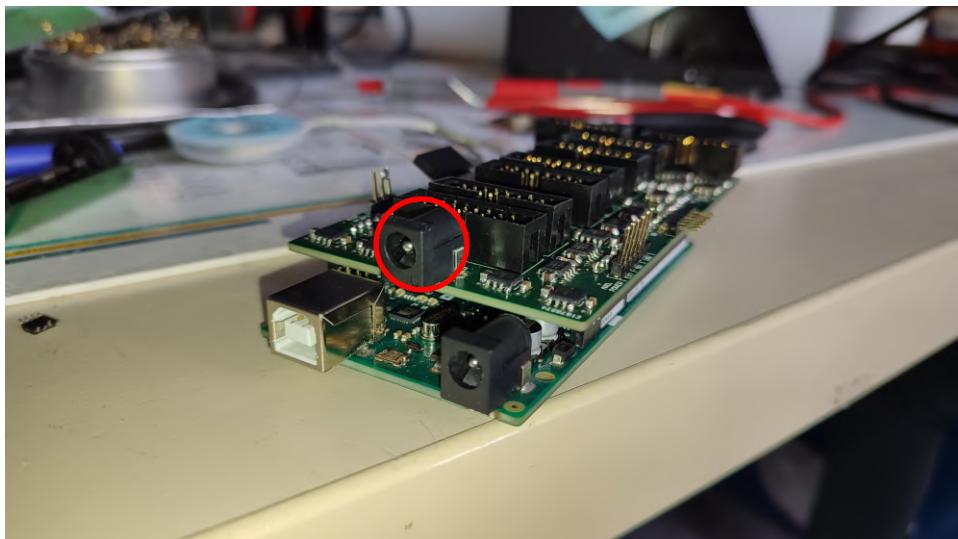


FIGURE 40 – Alimentation du DriverBoard (rond rouge)

### 3.5 Résultats

Au terme de cette phase nous avons réussi à faire léviter une particule à l'aide d'une seule plaque dont chaque transducteur est contrôler séparément, que ce soit avec la méthode de calcul des phases optimiser ou vortex.

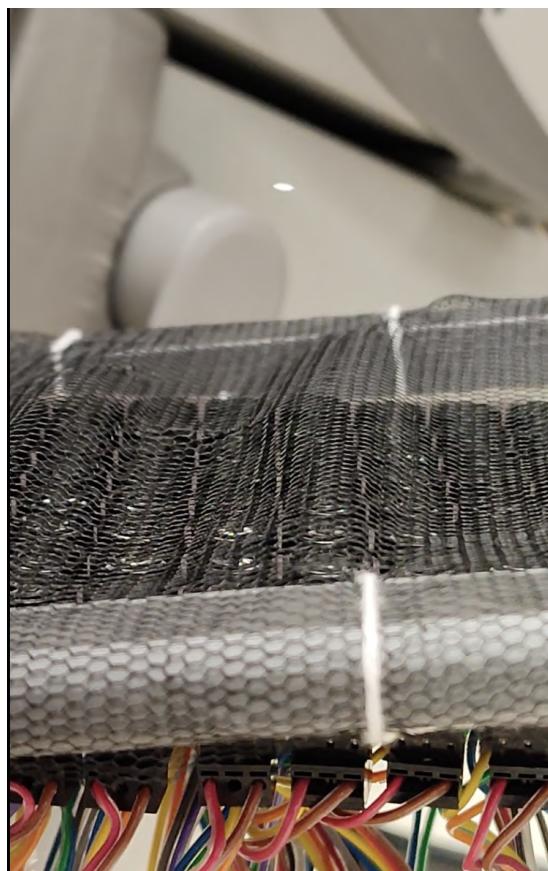


FIGURE 41 – Lévation d'une particule

De plus, nous avons souhaiter confronter nos résultats théoriques aux résultats réels et nous assurer que les programmes permettant de calculer les champs de pression que nous produisons soit exact. Pour ce faire nous nous sommes rendu dans les laboratoires de l'IEMN afin de mesurer le champs de pression que nous produisons réellement et ensuite pouvoir le comparer aux résultats théorique.

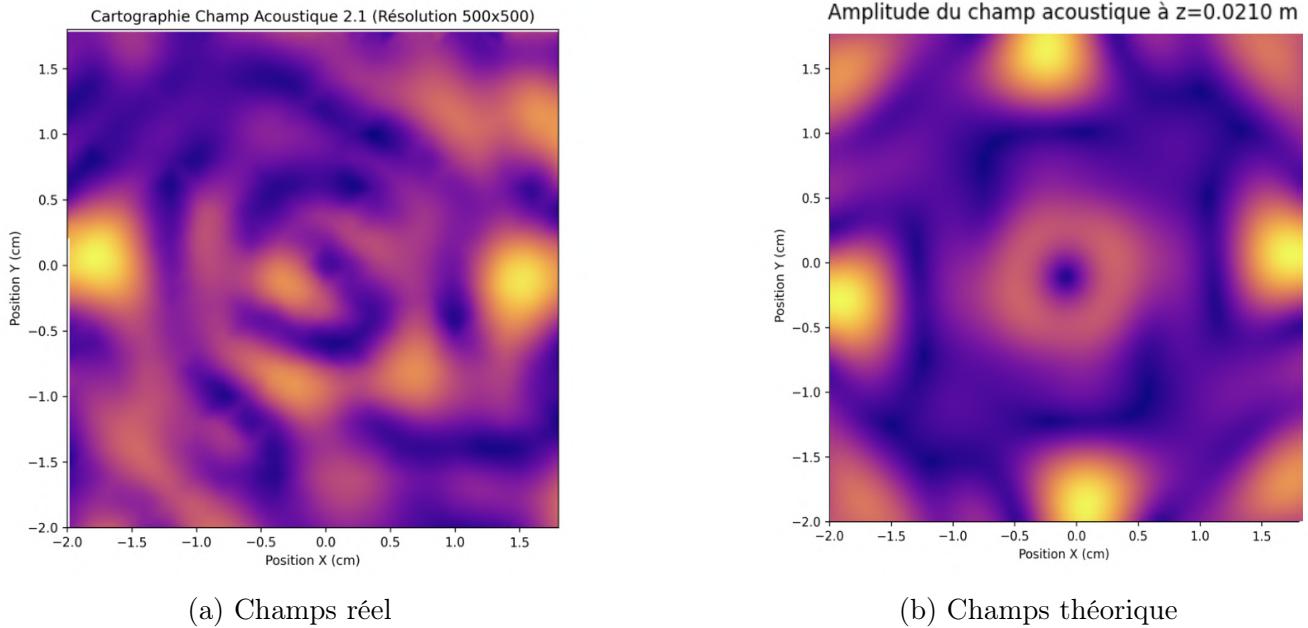


FIGURE 42 – Champ de pression acoustique sur un plan horizontal à hauteur  $z=0.02$  m

On observe que les deux champs de pression sont effectivement similaire. Pour autant on observe également des différences notables, on remarque notamment l'absence de symétrie par rotation de  $90^\circ$  dans les mesures réel. Au premier abord cela est particulièrement étonnant car le problème possède une symétrie par rotation. Cependant il est à noté que les phases calculées théoriquement sont ensuite discrétisé (à un pas de  $\pi/5$ ) cela peut expliquer les différences observer. Un autre explication est l'impact des instruments de mesures sur le champs de pression. En effet nos instruments de mesure (micro et pied de celui-ci) perturbe le champ de pression.

### 3.6 Problèmes rencontrés durant la phase 2

#### Problème 1

Le premier problème rencontré concerne la validité des connexions du driver board. En effet, le programme DriverMega permet d'envoyer un signal de  $40\text{kHz}$  sur tous les pins du driver board.

Lors du premier test, nous avons remarqué que de nombreux transducteurs n'envoyaient aucun signal. Après avoir tout re-vérifié, il s'est avéré que le problème venait de problèmes dans les soudures du driver board ce qui empêchait le passage de l'information.

Afin de vérifier le bon fonctionnement de la carte, voici les étapes à suivre :

- Vérifier que le signal de la sortie  $SYN$  est un créneau de  $40\text{kHz}$  ;
- Vérifier que toutes les masses du driver board sont bien connectées entre elles (vous devriez observer un signal de valeur négatif pour toutes les broches sur l'oscillo) ;
- Vérifier ensuite que chaque broche reçoit le même signal que celui envoyé sur la sortie  $SYN$ .

Pour mener à bien le dernier point :

- Téléverser le code DriverMega sur l'arduino Mega ;
- Ne pas oublier d'alimenter le driverboard ;
- Faire les test avec une sonde comme sur l'image ci-dessous.

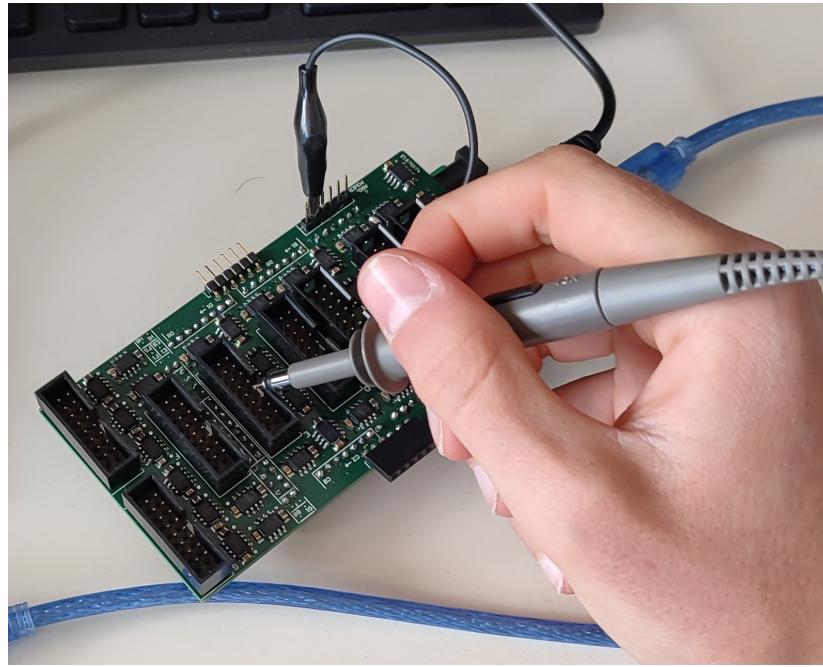


FIGURE 43 – Test avec sonde oscilloscope

## Problème 2

Le signal ne passait pas systématiquement dans les nappes de câbles qui font le lien entre le driver board et les transducteurs. Le problème était plus fréquent pour les fils de couleur grise que les fils multicolores. En effet, les gris étaient trop fins pour être bien découpés par les Connecteur IDC (ceux à 3 contacts sont donnés en dessous mais même problème avec les 16 entrées) :

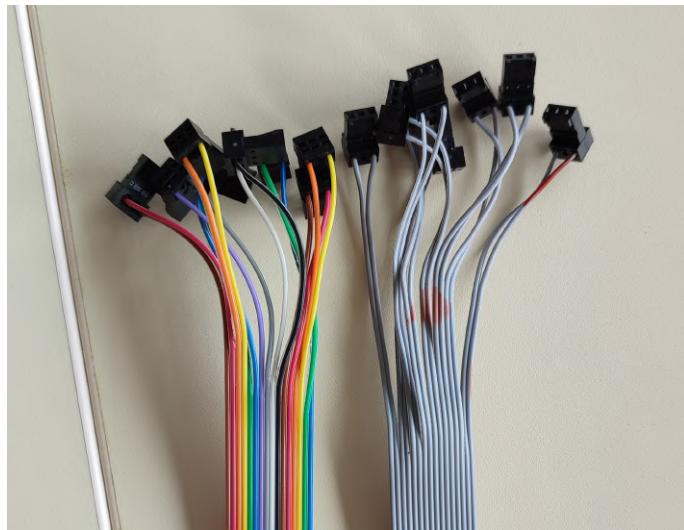


FIGURE 44 – Connecteur IDC problématique

**Conclusion :** veillez à choisir des câbles de diamètre conséquent afin d'éviter ces problèmes de connexion.

## Problème 3

Les connexions du DriverBoard étaient mal soudées : il faut bien vérifier que l'étain épouse au mieux les trous VIA (trous faisant le lien entre chaque couche du PCB) afin d'assurer une connexion optimale.

## Problème 4

Ce problème est en pratique un souci que nous avons rencontré durant la phase 2. Nous l'évoquons ici car nous avons fait le choix de ne plus du tout parler du programme Java dans notre documentation, simplement car nous ne l'utilisons plus et que nous n'avons rien repris des différents codes associés.

Le programme Java que nous utilisions (issu de [Ultraino](#)) était en pratique bien trop lourd pour l'utilisation que nous souhaitions en faire. De plus, nous ne comprenions pas bien son fonctionnement.

Nous avons donc décidé de tout recoder nous-mêmes pour avoir un contrôle total sur ce que nous faisons.

## 3.7 Vers la phase 3

### 3.7.1 Résumé de l'étude de brevets

La technologie de lévitation acoustique a donné lieu à plusieurs dépôts de brevets, notamment le brevet FR3144022 (Source 5) déposé par le CNRS. L'analyse montre que la majorité des dispositifs brevetés reposent sur l'utilisation d'ondes ultrasonores stationnaires. Ces ondes, créées entre un réseau de transducteurs et un réflecteur, génèrent des nœuds de pression dans l'air. Ces noeuds permettent de stabiliser des gouttelettes liquides en lévitation sans contact physique.

Les dispositifs présentent plusieurs caractéristiques remarquables :

- Ils permettent la manipulation sans contact de gouttelettes, évitant tout risque de contamination par un récipient.
- Ils autorisent la fusion contrôlée de deux gouttes, déclenchant une réaction chimique en suspension.
- L'absence de paroi facilite l'observation en temps réel des réactions par des techniques optiques telles que la spectroscopie infrarouge, UV-Visible ou Raman.
- Certains dispositifs intègrent une automatisation des opérations : injection, fusion, évaporation ou expulsion de gouttes, ouvrant la voie à des applications de microfluidique aérienne.
- Ils peuvent fonctionner dans une enceinte à atmosphère contrôlée, permettant l'étude de réactions sensibles à l'humidité ou à l'oxygène.

Ces systèmes ouvrent des perspectives dans des domaines variés comme la chimie fine, la formulation pharmaceutique, ou encore l'analyse de substances réactives ou instables dans un environnement propre.

### 3.7.2 Justification du choix d'application

L'application retenue pour la phase 3 concerne l'utilisation de la lévitation acoustique pour effectuer des micro-réactions chimiques sans contact, notamment dans les secteurs de la pharmacie et de l'analyse chimique.

Ce choix repose sur plusieurs constats issus de l'étude de marché :

- La demande industrielle est forte dans les domaines pharmaceutiques et analytiques. Dans ces secteurs, la miniaturisation et la stérilité des procédés sont des priorités.
- La lévitation acoustique présente des avantages techniques clairs : suppression du contact avec les parois, précision de la manipulation de volumes très faibles, possibilité d'observer les réactions directement sans transfert ni dilution.

- Elle permet également de réduire les consommables et de limiter les risques liés à la manipulation de substances toxiques ou stériles.
- Le contexte de marché est favorable avec une dynamique de recherche et d'investissement en Europe, en Amérique du Nord et en Asie. Des institutions comme le CNRS, la NASA, ou des entreprises de biotechnologie s'y intéressent activement.

Cette application est directement compatible avec les brevets existants utilisant des ondes stationnaires. Elle répond à des besoins industriels concrets et représente une solution innovante et viable pour des secteurs en forte demande.

### 3.7.3 Ondes progressives périodiques

On appelle onde progressive le phénomène de propagation d'une perturbation, de proche en proche, sans transport de matière mais avec transport d'énergie. La perturbation consiste en une déformation (au sens large) de son milieu de propagation.

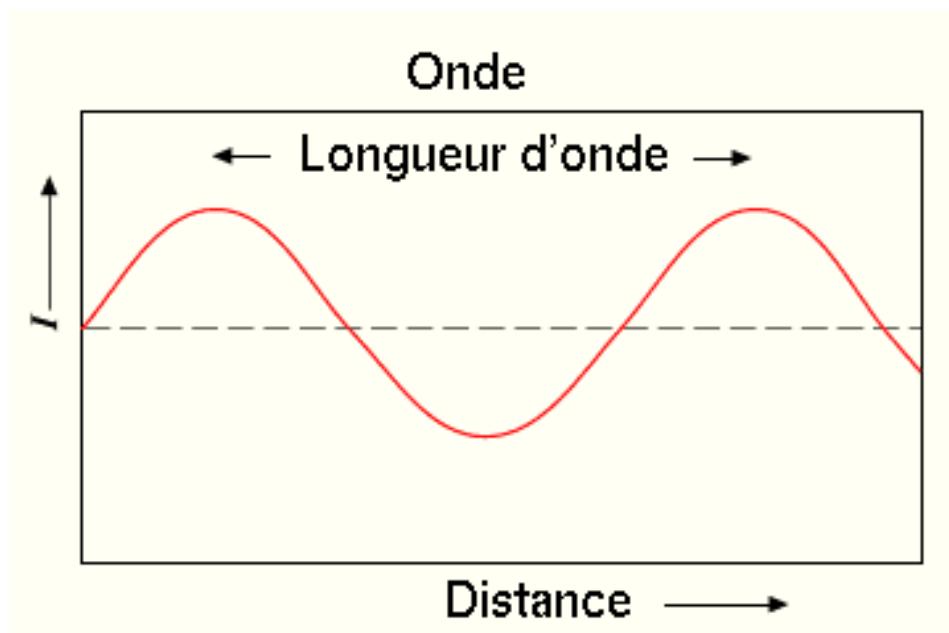


FIGURE 45 – Schéma de principe d'une onde progressive - Source 7

Exemple d'ondes : vagues, onde sonore, lumière, ...

La perturbation est produite initialement par une source, comme par exemple des baffles de haut parleur pour une onde sonore. Dans notre cas, la source sera chacun des transducteurs. L'onde résultante sera la somme des ondes créées par chaque transducteur.

Quand la source impose une perturbation périodique, l'onde progressive ainsi générée est elle-même périodique. Cette onde est caractérisée par une double périodicité : temporelle et spatiale.

### 3.7.4 Ondes stationnaires

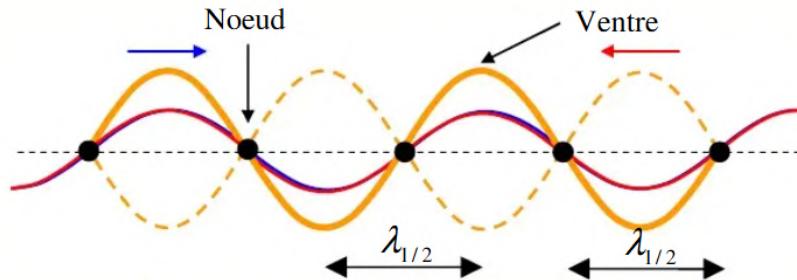


FIGURE 46 – Schéma de principe d'une onde stationnaire - Source 6

La figure précédente montre le principe d'ondes stationnaires, qui seront émises dans notre cas par la superposition d'ondes ultrasonores émises en face à face par un groupement de transducteurs (voir Proposition de modèle pour la phase 3), générant des nœuds et des ventres de pression. Les particules ou gouttelettes restent stables dans les nœuds de pression, là où la force de radiation acoustique contrebalance la gravité.

### 3.7.5 Pourquoi privilégier les ondes stationnaires ?

Voici les points d'intérêt principaux :

- **Intensité** : L'intérêt principal de cette utilisation est la sommation des contributions de chacune des plaques. En effet, les valeurs de pression sont égales à la somme des valeurs émises de chaque côté, donc le phénomène permet "d'enfermer" beaucoup plus efficacement les particules.
- **Stabilité** : les nœuds fixes offrent des points de suspension précisément placés, contrairement aux ondes progressives où les particules seraient entraînées hors du foyer.
- **Contrôle précis** : en ajustant la fréquence ou la distance transducteur/réflecteur, on règle la position des nœuds de demi-longueur d'onde  $\lambda/2$ .

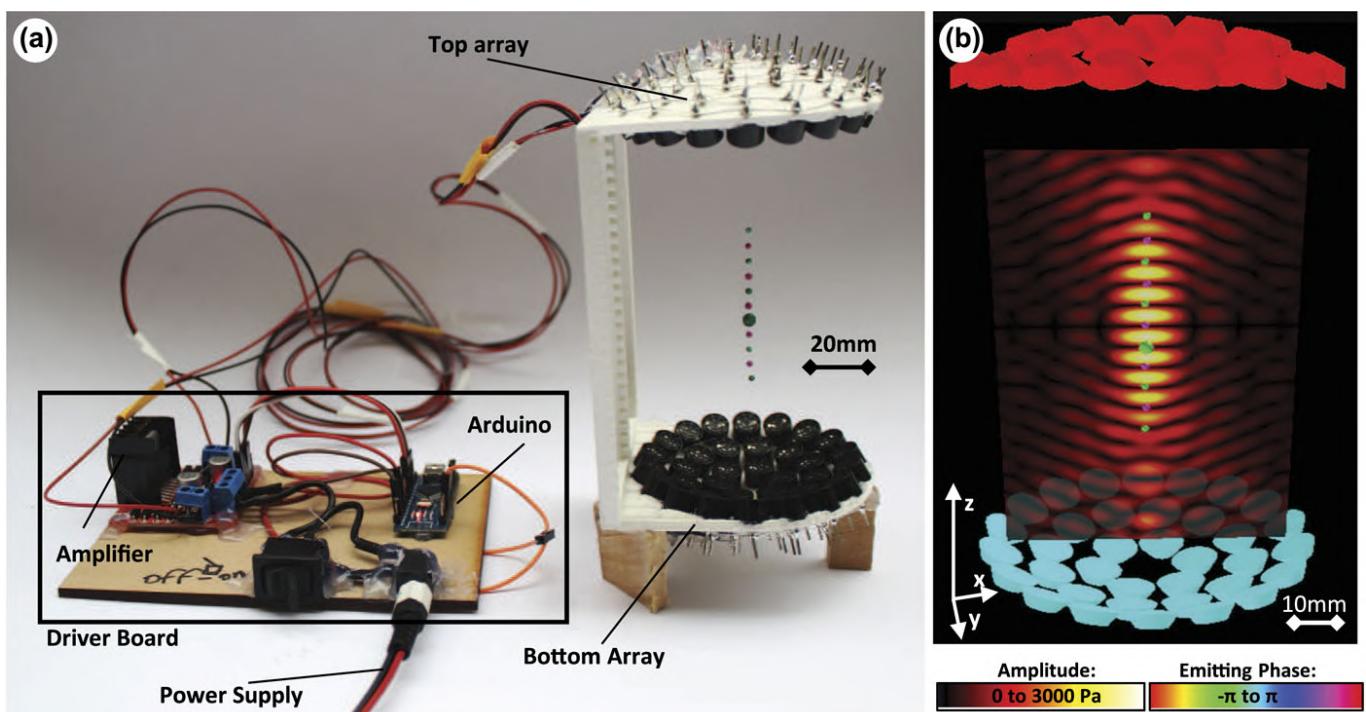


FIGURE 47 – Photo du Tinylev - Source 1

Pour prouver la différence précédente, nous allons comparer les valeurs des forces avec et sans deuxième plaque.

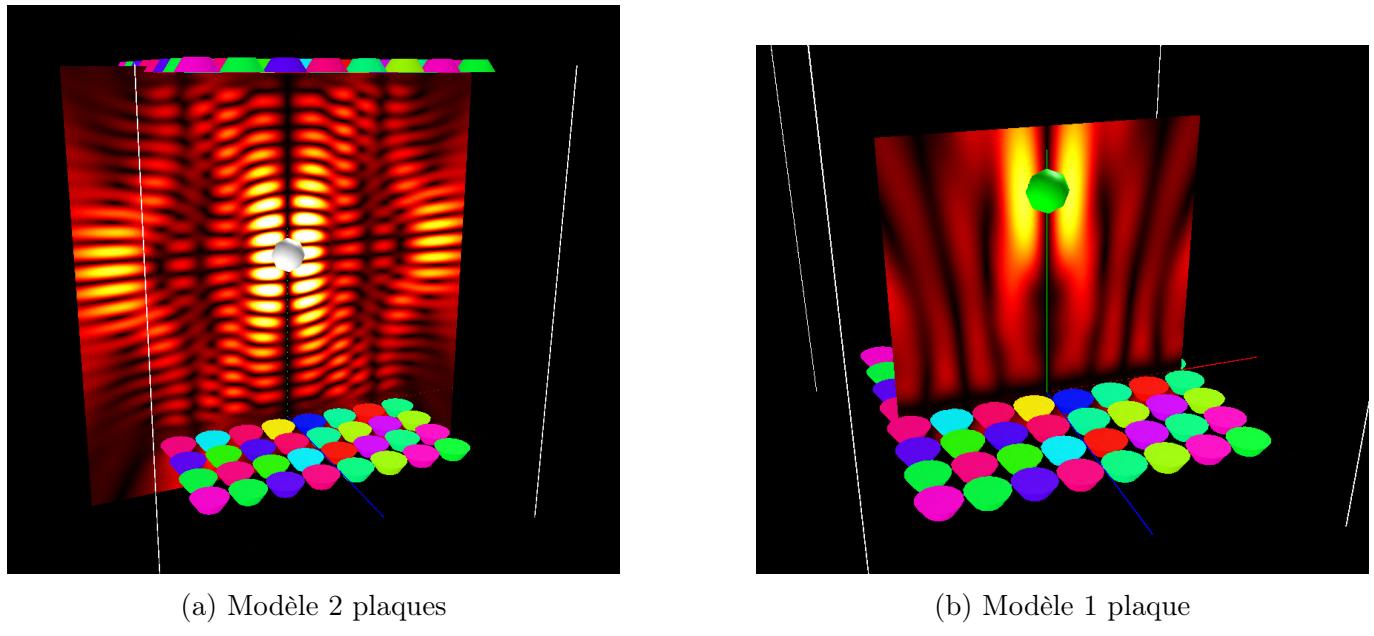


FIGURE 48 – Comparaison modèles avec 1 ou 2 plaques

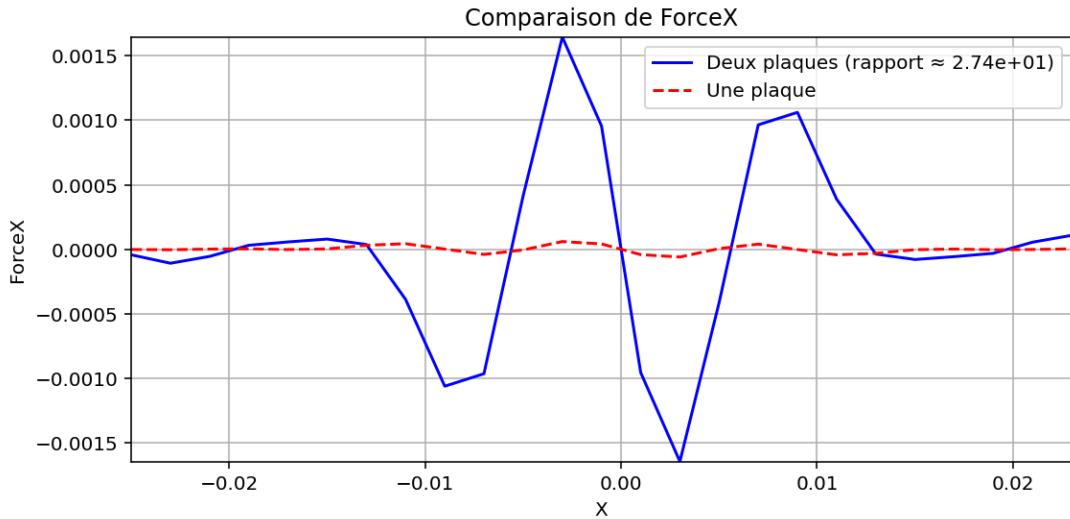


FIGURE 49 – Comparaison des forces selon X pour 1 et 2 plaques. Dans les deux cas : z = 0 cm et y = 5 cm.

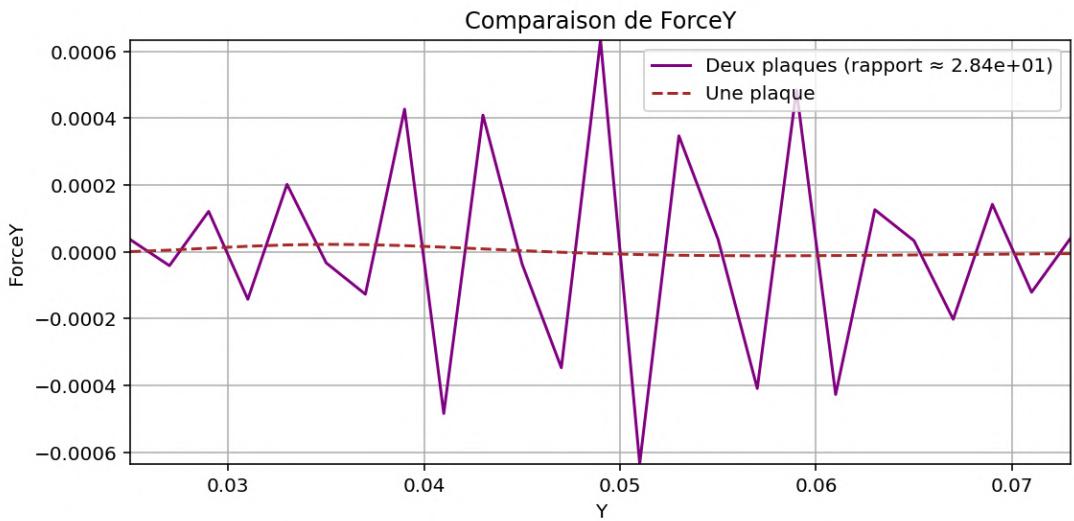


FIGURE 50 – Comparaison des forces selon Y pour 1 et 2 plaques. Dans les deux cas :  $x = 0$  cm et  $z = 0$  cm.

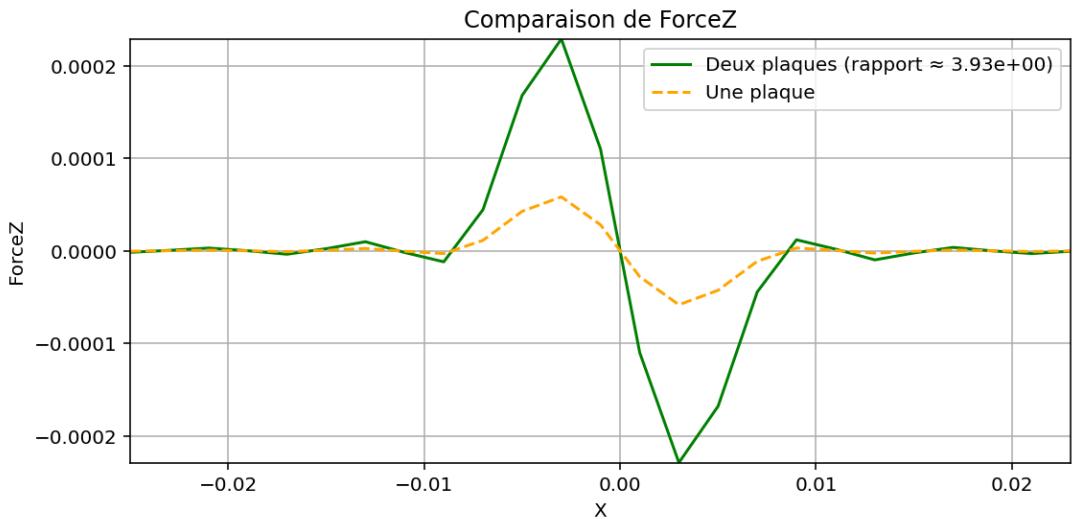


FIGURE 51 – Comparaison des forces selon Z pour 1 et 2 plaques. Dans les deux cas :  $x = 0$  cm et  $y = 5$  cm.

### 3.7.6 Conclusion

Le cas avec deux plaques montre bien que la force exercée sur la particule dans les zones d’importance (dans le cas d’un twin, ce sont les forces latérales, donc selon X et Z) sont supérieures d’un facteur 30.

Ce facteur est contestable et montre les limites théoriques du modèle physique donné par Marzo dans son article de 2015. Nous avons réalisé notre propre partie théorique et obtenons un facteur 2 (voir figure suivante) entre les situations, ce qui nous conforte malgré tout à passer en ondes stationnaires.

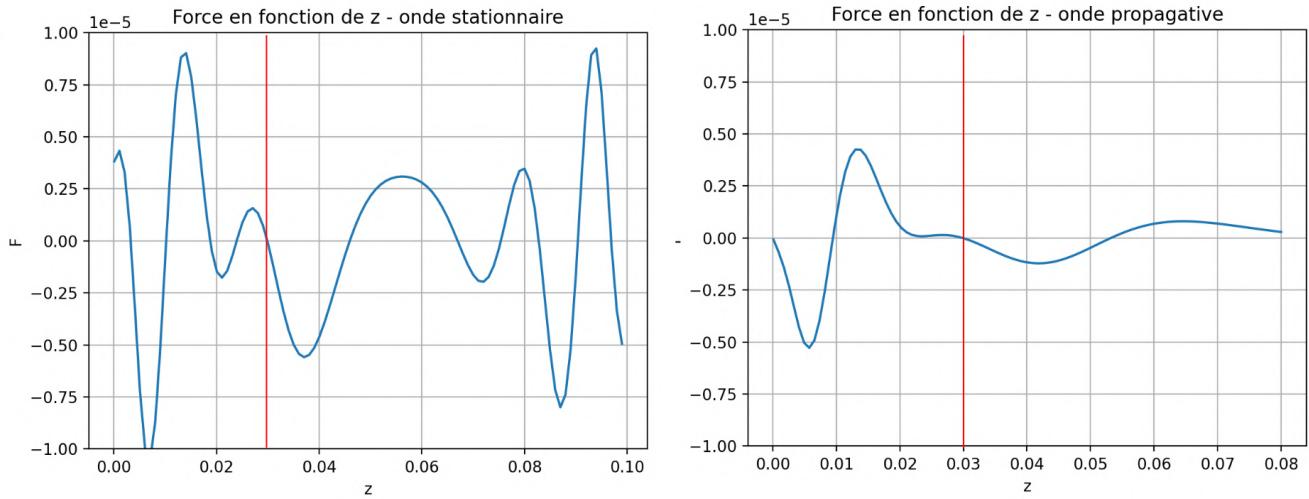


FIGURE 52 – Comparaison des forces selon Z pour 1 et 2 plaques (théorie propre).

### 3.7.7 Proposition de modèle pour la phase 3

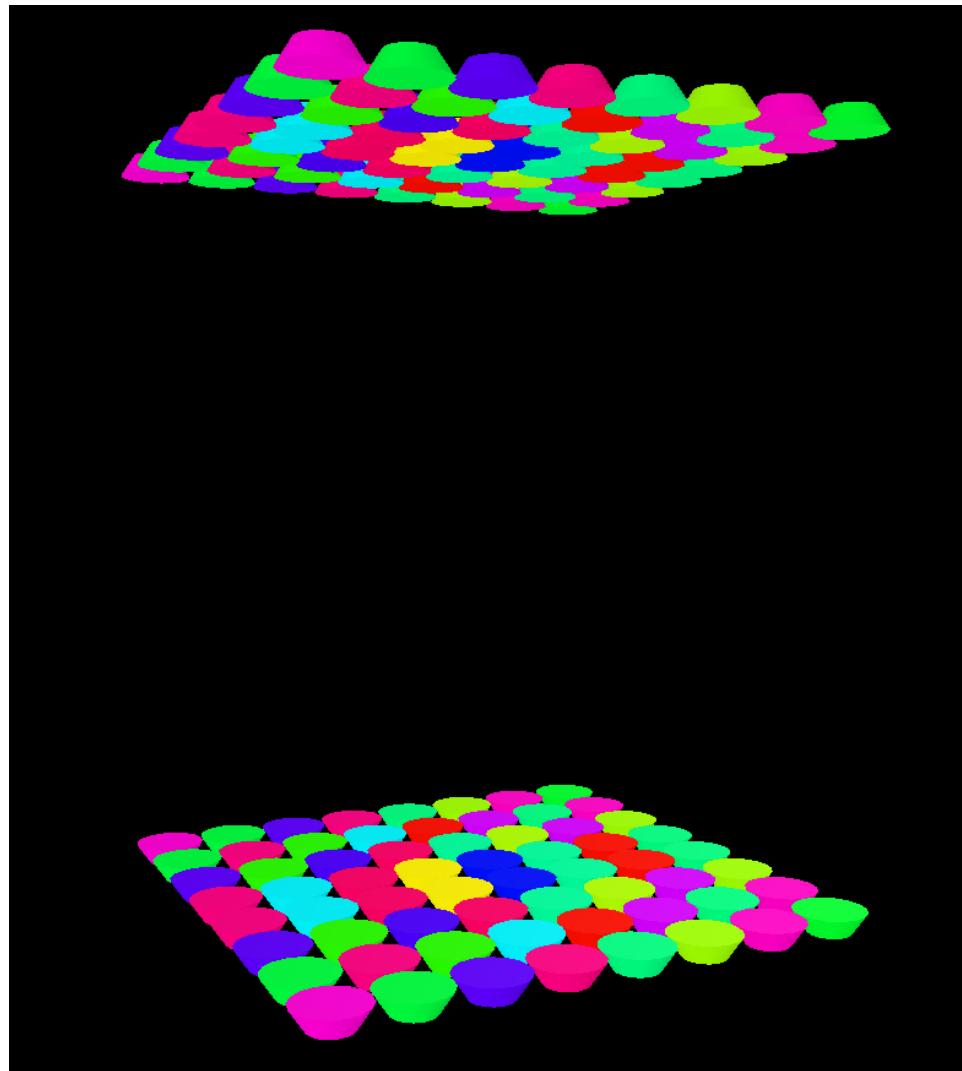


FIGURE 53 – Modèle avec deux plaques - Screen issu de la simulation AcousticField3D

Le système précédent est le regroupement des deux premières phases :

- L'utilisation du principe d'ondes stationnaires afin d'augmenter significativement la pression

(donc les forces) appliquée sur l'objet en lévitation. Ceci permet de faire léviter des liquides plus difficile à maintenir dans les airs qu'une bille de polystyrène ;

- L'utilisation de la plaque étudiée durant la phase 2 afin de pouvoir bouger librement différents objets en lévitation.

Les points précédents sont essentiels dans la volonté de mener à bien notre application : réunir deux gouttes de liquide sans contact aucun avec des matériaux externes.

## 4 Troisième phase : lévitateur à deux plaques

### 4.1 Pourquoi ce prototype ?

Comme évoqué dans la partie [Vers la phase 3](#), nous avons pensé et conçu ce nouveau prototype pour augmenter autant que faire se peut la force appliquée sur la particule en lévitation.

Nous l'avons montré, ajouter une plaque est la solution que nous avons trouvée comme la plus évidente. Nous n'avons finalement qu'à copier le prototype 2 et travailler sur la manière d'associer deux Arduino pour l'envoi des phases et le tour est joué.

### 4.2 Evolution des théories physiques et nouveaux programmes

#### 4.2.1 Méthode optimizer

##### Modifications réalisées et pistes étudiées

Pour améliorer la qualité des pièges et les forces générées nous avons souhaité augmenter le nombre de transducteur. Nous avons choisi une architecture à deux plaques de 64 transducteurs chacun, placées en vis-à-vis à une distance réglable. Nous avons donc adapté nos programmes pour que ceux-ci fonctionne avec cette nouvelle architecture. Pour ce faire il suffit de chercher nous plus 64 mais 128 phases optimale, tout en veillant à prendre en compte la position de ces nouveaux transducteur qui ne sont plus en dessous mais au dessus du piège.

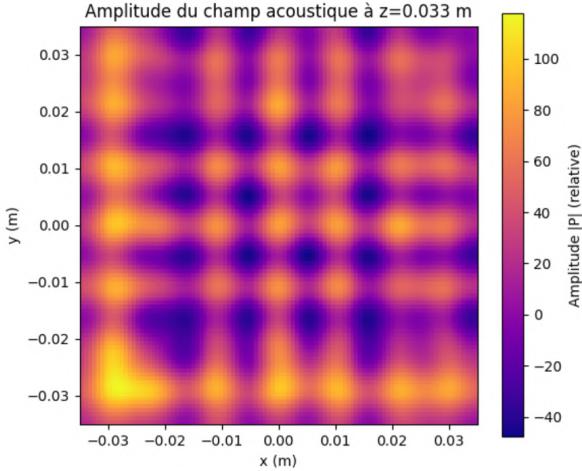
Nous avons ensuite souhaité piéger deux particules simultanément. Pour ce faire nous avons reproduit la réflexion ayant mené à la création de la méthode optimizer, nous avons donc cherché à minimiser la force non plus en un point mais en deux, tout en minimisant la pression en ces deux points. Nous pouvons déjà calculer la grandeur : pression - gradient (force) en un point. En calculant cette grandeur aux deux positions des pièges considérés, il suffit alors de minimiser le résultat.

Suite à la création de ces programmes créant deux pièges, nous avons essayé des variations dans les grandeurs à minimiser. Nous avons par exemple essayé de minimiser non pas la somme des deux grandeurs mais le maximum des deux. Nous avons également, dans le but d'éliminer la possibilité d'obtenir en résultat un minimum local, de réaliser l'opération de minimisation avec de multiples listes de phases initiales, tirées au hasard. Cependant, toutes ces méthodes ne semblent ni accélérer les programmes ni aboutir à des résultats différents.

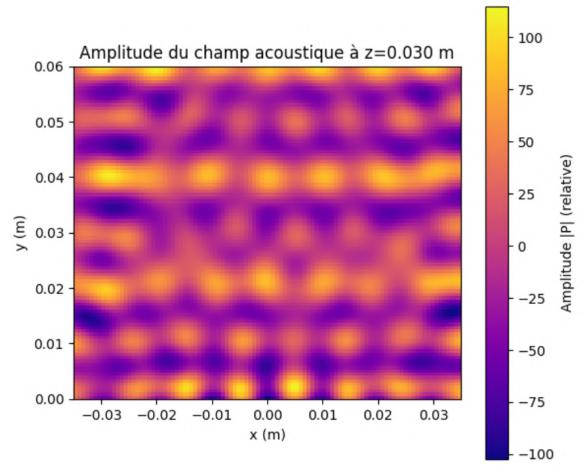
Enfin, les expérimentations nous ont poussé à nous demander si les pièges produits par la méthode optimizer n'étaient pas trop petits. Cette réflexion est pertinente puisque tous nos calculs ne se font qu'en un point, nous n'avions alors aucun moyen de maîtriser le diamètre du piège. Pour ce faire, nous avons testé un certain nombre de solutions potentielles. Nous avons par exemple cherché à minimiser la moyenne de la grandeur  $F_{opt}$  dans un rayon choisi autour du piège. Nous avons également cherché à ne moyennner que la pression, cela est moins restrictif et permet malgré tout d'agrandir le piège. Nous avons également fait des tests en appliquant des contraintes sur des géométries plus complexes autour du piège (uniquement sur la sphère, pondéré selon la distance au piège). Nous avons également essayé de contraindre directement le potentiel de Gorkov notamment dans une sphère autour du piège et sur les abords de cette sphère. Cependant ces méthodes d'élargissement du piège ne semblent pas très concluantes, en effet soit elles dégradent trop fortement la qualité du piège soit elles sont sans effet.

## Résultats

Nous avons commencé par chercher à augmenter la force produite en utilisant deux plaques pour produire notre piège. Cela à effectivement produit des pièges plus stables. Cependant on observe que cette configuration à tendance à créer de multiples pièges ce qui rend difficile le placement de la particule à l'emplacement souhaité.



(a) champs de pression - methode Optimizer



(b) champs de pression - methode Optimizer

FIGURE 54 – Champs de pression produit par la méthode Optimizer

Nous avons également cherché la distance interplaque la plus optimale. Pour ce faire nous avons tracé la grande pression-gradient de la force, que nous obtenions en fin d'optimisation, pour chaque distance intraplaque, en piégeant la particule toujours au centre des deux plaques. Dans le graphique obtenu l'altitude optimal est donc celle de la valeur minimale soit  $z = 0.03$

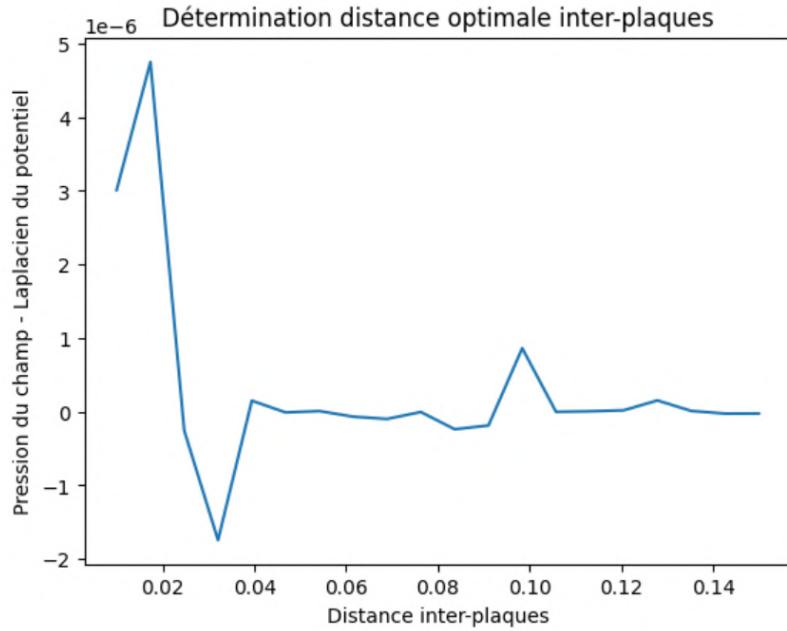


FIGURE 55 – Distance optimale entre deux plaque

Afin d'élargir le piège et de le rendre plus stable nous avons essayé de changer les paramètres de l'optimisation. On observe ici les résultats, ou plutôt l'absence d'effet, de l'optimisation cherchant à minimiser la pression moyenne dans une boule autour du piège.

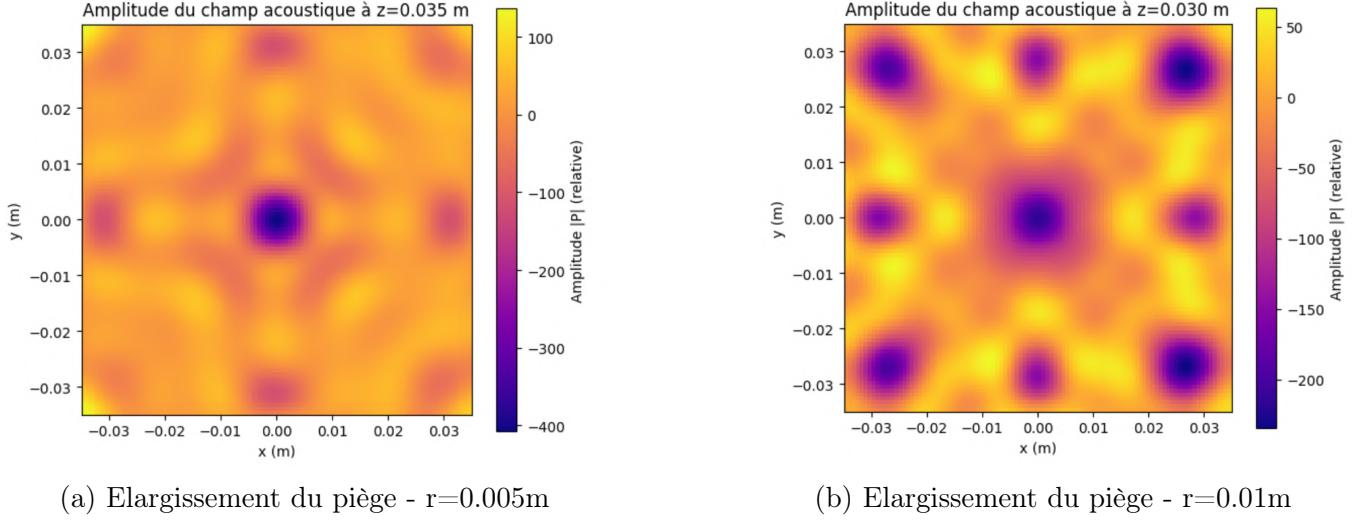


FIGURE 56 – Inefficacité des tentatives d’élargissement des pièges

Parmi les résultats obtenus, on observe que certaines positions de pièges sont plus facilement réalisables que d’autres. En particulier, la génération de deux pièges situés à  $x=\pm 0,01\text{m}$  converge de manière robuste et reproductible, tandis que des configurations similaires à  $x=\pm 0,015\text{m}$  s’avèrent nettement plus difficiles à obtenir. Cette différence de comportement suggère que la faisabilité et la qualité des pièges dépendent de leur position spatiale. Une explication plausible réside dans la position relative de ces pièges par rapport à la grille discrète des transducteurs, qui peut influencer localement la capacité du réseau à reproduire fidèlement le champ acoustique cible.

Enfin nous avons essayé de réaliser 2 piège simultanément. Cela semble assez bien fonctionner. On observe toujours les mêmes limites qu’avec un seul piège et les pièges créés sont légèrement moins qualitatifs que dans le cas à une seule particule. Cependant cela reste suffisant pour faire élever deux particules simultanément

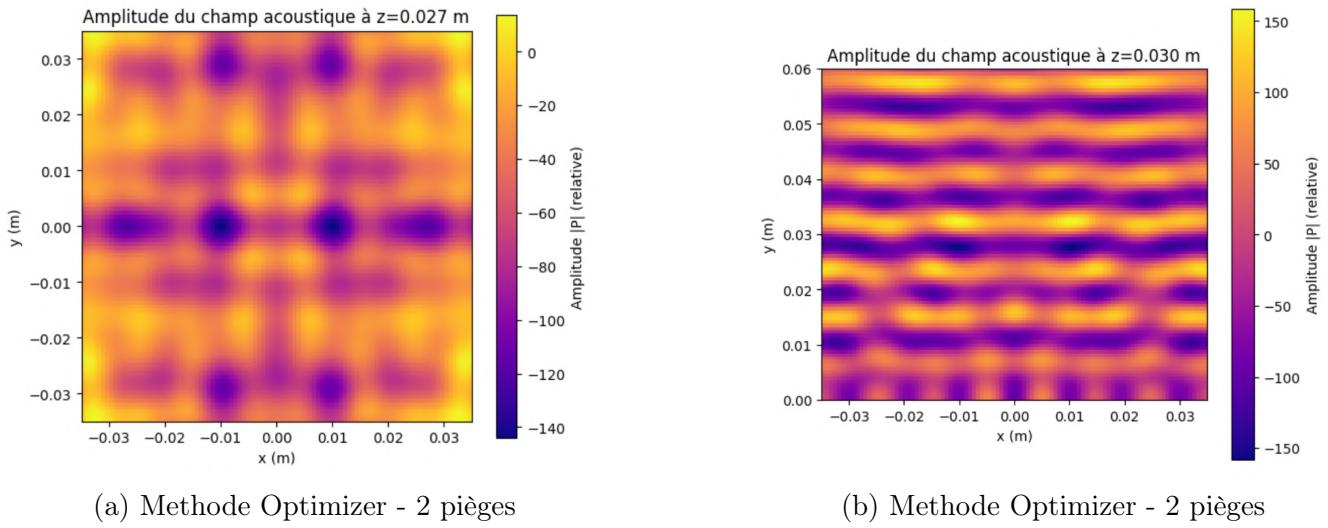


FIGURE 57 – Crédit de 2 pièges via la méthode Optimizer

#### 4.2.2 Méthode vortex

##### Modifications réalisées et pistes étudiées

Afin d'améliorer la qualité des pièges et les forces générées nous avons souhaité augmenter le nombre de transducteur. Nous avons choisi une architecture à deux plaques de 64 transducteurs chacune, placées en vis-à-vis à une distance réglable. Nous avons donc adapté nos programmes

pour que ceux si fonctionne avec cette nouvelle architecture. Pour ce faire il suffit de calculer les valeurs de phases et d'amplitude prise par le champs de pression pour tout les transducteurs. La méthode reste strictement la même qu'avec une seule plaque de transducteur.

Dans l'optique de faire léviter deux particule simultanément nous avons adapté la méthode vortex. Pour ce faire nous avons simplement calculer la somme des champs de pression formé par deux vortex centré au deux points voulu. Cependant cette solution simple peut créer des problèmes lorsque les deux vortex sont très proches. Nous avons donc essayé d'ajouter un coefficient dépendant de la distance au centre du vortex dans la somme (nous avons notamment essayé différentes puissances pour ces coefficients).

## Résultats

Lorsque l'on chercher à créer deux pièges on observe que si certe ceux-ci apparaissent, lorsqu'il sont trop proche il on tendance à laisser une zone de basse pression entre eux, ceux qui réduit considérablement la qualité du confinement des deux particules. Cependant, nous avons obeserver experimentalement que ces pièges étaient d'une qualité suffisante pour assurer la lévation de deux particules.

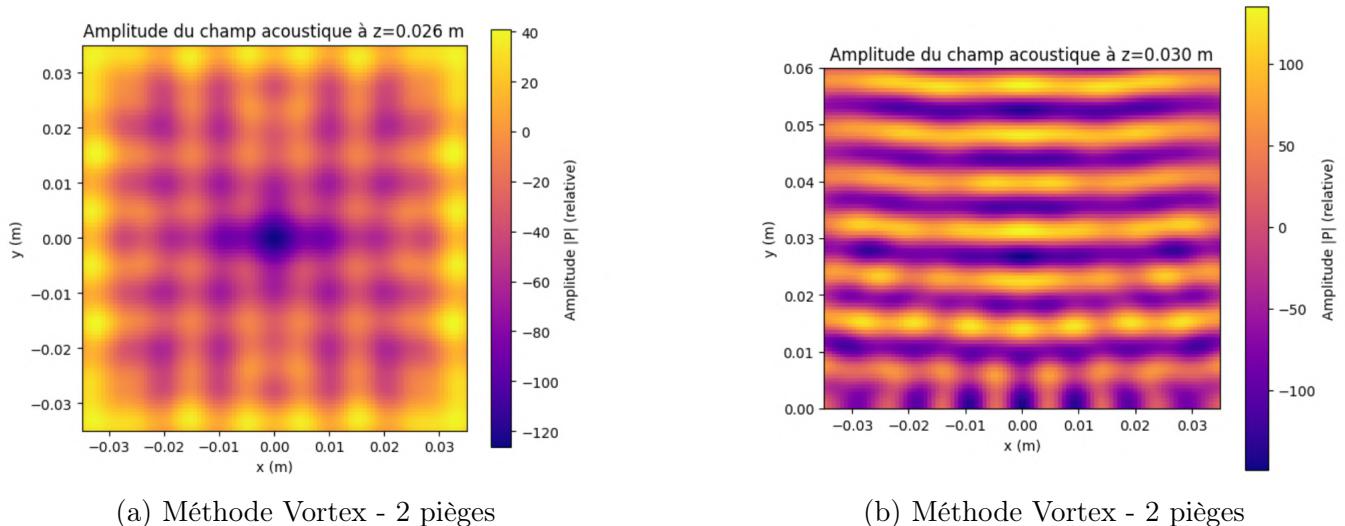
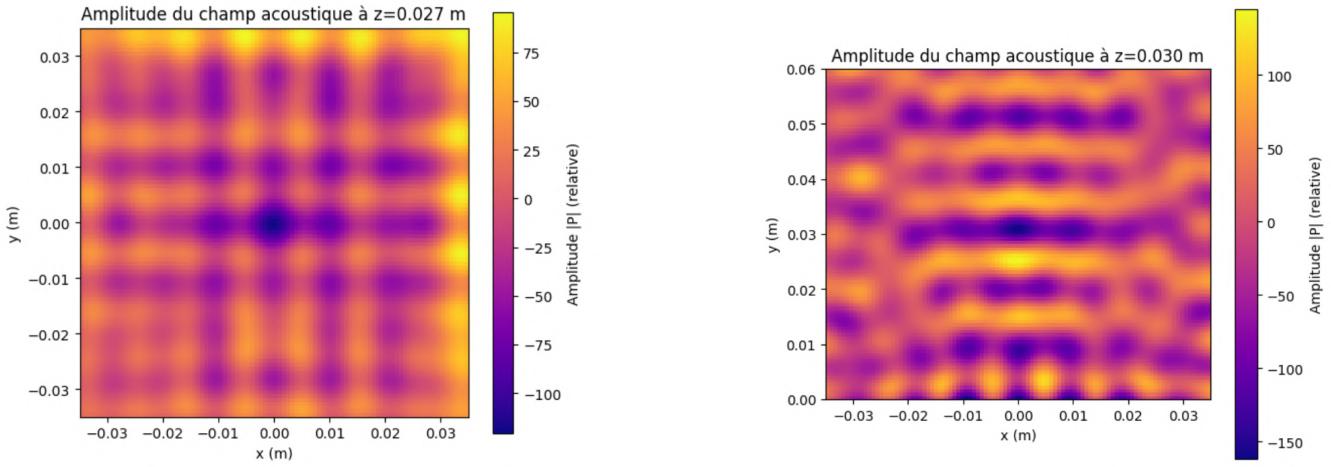


FIGURE 58 – Crédit de 2 pièges via la méthode Vortex

L'ajout d'une deuxième plaque, en plus d'aumenter les forces en jeux, augmente nettement la qualité des pièges formées et par la même le confinement des particules.



(a) Méthode Optimizer - 2 pièges

(b) Méthode Optimizer - 2 pièges

FIGURE 59 – Crédit de 2 pièges via la méthode Vortex avec 2 plaques

#### 4.2.3 Pistes de développements

Ce travail repose sur un certain nombre d'hypothèses et de choix de modélisation qui ouvrent naturellement la voie à plusieurs pistes d'amélioration. Tout d'abord, la taille finie des transducteurs constitue une première limite importante. Cette approximation peut affecter la qualité du champ généré, en particulier pour des structures fines comme les vortex acoustiques. De manière plus générale, l'effet de la discréétisation, tant en phase qu'en amplitude, mériterait une étude approfondie afin d'évaluer son impact sur la stabilité et la profondeur du piège acoustique.

Par ailleurs, le dispositif expérimental se limite ici à une ou deux plaques de transducteurs. Une extension naturelle consisterait à envisager une configuration entourant complètement la zone de piégeage, par exemple une distribution quasi sphérique de transducteurs autour de la particule. Une telle géométrie pourrait renforcer la symétrie du champ et améliorer significativement les performances du piège, notamment dans le cadre de la méthode des vortex acoustiques.

Un autre point important concerne l'écart entre le modèle théorique et la réalité expérimentale. Les transducteurs réels ne produisent pas des ondes parfaitement sphériques ou idéales, et présentent des imperfections (directivité, non-linéarités, variations d'amplitude) qui ne sont pas prises en compte dans le modèle vortex actuel. Revenir de manière critique sur ces hypothèses permettrait d'affiner le modèle et de mieux prédire les performances expérimentales.

De plus, l'utilisation du potentiel de Gor'kov repose sur une approximation valable dans un cadre précis (particules petites devant la longueur d'onde, champ faiblement perturbé). Des modèles plus complets existent et permettraient une description plus fidèle des forces acoustiques, au prix toutefois d'un coût de calcul bien plus élevé. La question de la pertinence de ces modèles avancés se pose alors : le gain en précision justifie-t-il la complexité et le temps de calcul supplémentaires ?

Enfin, l'optimisation des paramètres du vortex, notamment les indices  $l$  et  $m$ , constitue une autre piste prometteuse. Un ajustement plus fin de ces paramètres pourrait permettre d'améliorer la stabilité du piège ou de mieux contrôler la position et la dynamique de la particule piégée.

### 4.3 Notice de montage du lévitateur à deux plaques

Le Prototype 3 est en quelques sortes une version dédoublée du Prototype 2, avec 2 plaques en opposition. Ainsi, les étapes de montage identiques à celles du prototype 2 ne sont pas reprises dans cette sous-section.

#### 4.3.1 Étapes clés

1. Liste du matériel et commande
2. Conception et montage du prototype 3
3. Soudure des composants et montage des fils
4. Montage des transducteurs sur les plaques

#### 4.3.2 Liste du matériel et commande

Les composants nécessaires à la réalisation de ce prototype sont, à une différence près, les mêmes que ceux utilisés pour le Prototype 2, en quantité double.

Afin de disposer d'une amplitude de sortie d'environ 35 Vpp, les drivers TC4427CPA ont été remplacés par des transistors MOSFET IX4427NTR.

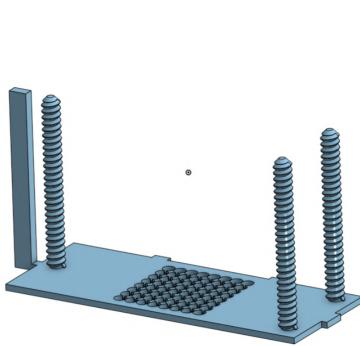
#### 4.3.3 Conception et montage du prototype 3

Le concept du prototype 3 est donc de mettre deux plaques en vis-à-vis et de pouvoir régler la distance entre elles. C'est ce point qui a valu le plus de réflexion pour mettre au point un prototype qui règle la distance entre les deux plaques de manière continue.

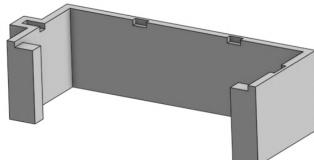
Nous avons d'abord repris la plaque du prototype 2 que nous avons allongée et sur laquelle nous avons ajouté trois tiges filetées et une tige qui servira de règle.

Nous ajoutons aussi un support avec des encoches pour maintenir la plaque et un espace pour laisser passer les câbles électroniques. Nous avons choisi ce support pour avoir plus de stabilité qu'avec des pieds, car ce prototype va être plus long et plus lourd que le précédent.

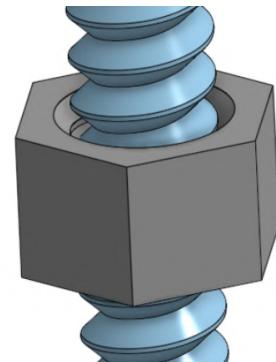
Nous ajoutons également trois écrous qui se vissent sur les tiges filetées à n'importe quelle hauteur et qui vont assurer le fait qu'on peut régler la distance entre les plaques de manière continue.



(a) Plaque basse - Onshape



(b) Support - Onshape



(c) Écrou - Onshape

FIGURE 60 – Pièces de la partie basse

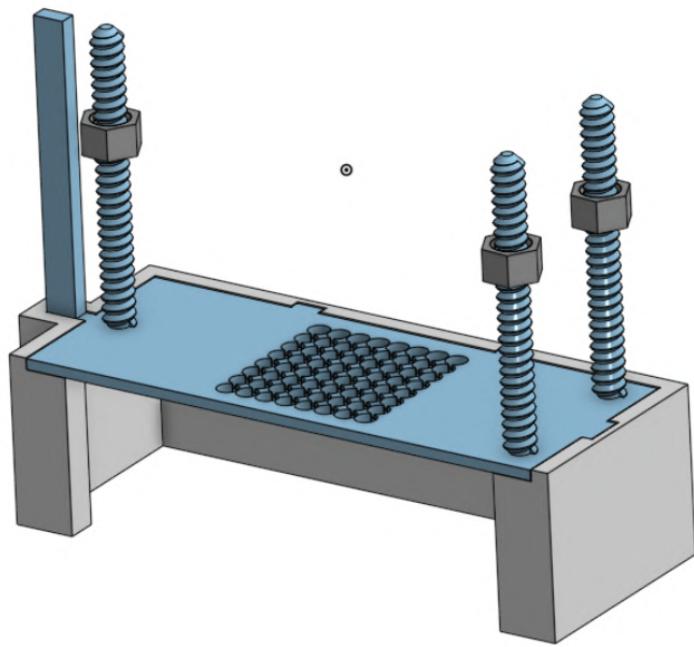


FIGURE 61 – Conception de la partie basse - Onshape

Pour la plaque en vis-à-vis, nous reprenons la plaque du bas et nous remplaçons les tiges filetées par des trous de diamètre légèrement plus grand.

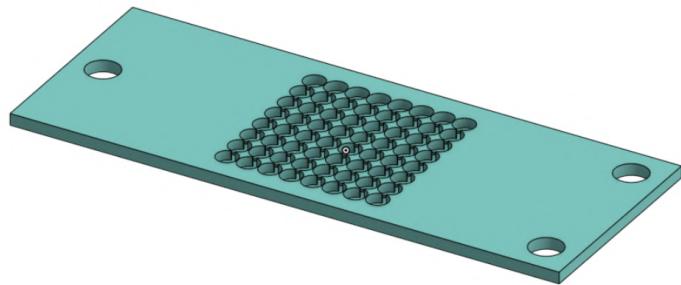
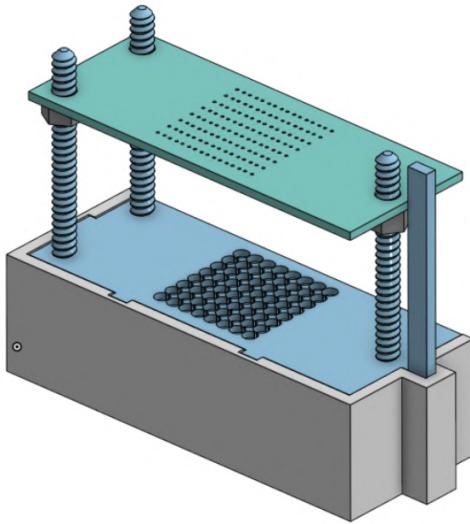


FIGURE 62 – Plaque haute - Onshape

Pour le montage, nous plaçons la plaque du bas sur le support, les écrous sur les tiges à la hauteur souhaitée, et la plaque du haut en vis-à-vis repose sur les écrous.



(a) Conception prototype 3 - Onshape



(b) Impression prototype 3

FIGURE 63 – Montage du prototype 3

## 4.4 Notice d'utilisation du lévitateur à deux plaques

### 4.4.1 Étapes clés

1. Prise en main des codes Arduino
2. Prise en main pratique

### 4.4.2 Prise en main des codes Arduino

Ce troisième prototype nécessitant l'utilisation de 2 plaques de lévitation contrôlables de façon indépendantes, il est alors nécessaire d'utiliser 2 cartes Arduino et donc d'être capable de les faire fonctionner de façon synchrone. Pour ce faire, nous avons mis en place un système *maître-esclave*. Les codes utilisés sont présentés en Annexe [7.6](#)

Le fonctionnement de ce système est assez basique : les phases à émettre sur chacune des deux plaques sont stockées "en dur" (dans le code) dans chacune des deux cartes Arduino, ce qui permet de ne pas avoir à gérer le transfert des octets de l'Arduino *maître* vers l'Arduino *esclave*. La communication entre les 2 deux cartes Arduino se résume donc à leur synchronisation sur le signal créneau du pin 2 de la carte *maître* permettant de cadencer l'émission des signaux pour les 64 transducteurs. Le fonctionnement schématisé de la communication entre les 2 programmes est repris par la figure [64](#).



FIGURE 64 – Schéma explicatif de la communication entre les 2 cartes Arduino

## Initialisation

Cette première partie des codes est tout à fait similaire à celle du code *NewDriverMega* présenté en 3.4.1 et est commune aux deux cartes Arduino.

Dans cette phase d'initialisation, les deux cartes Arduino configurent les constantes, les différentes variables qui seront utilisées ainsi que l'ensemble des ports parallèles (A, C, L, B, K, F, H, D, G et J) en sortie et les placent à l'état bas afin de garantir un état initial stable pour les transducteurs. Elles effacent ensuite le buffer contenant les données d'émission pour éviter toute valeur résiduelle.

La configuration de l'interruption INT5 (sur la broche 3) est mise en place pour détecter chaque front descendant du signal de synchronisation envoyé par la carte maître ; cette interruption alimente un compteur `sync_cnt` utilisé comme horloge externe.

Enfin, les modules inutiles du microcontrôleur (ADC, SPI, TWI, certains timers et ports série) sont désactivés afin de réduire la charge et d'assurer un fonctionnement précis et déterministe, ce qui est essentiel pour la génération synchrone des signaux.

Il est à noter que le code *maître* possède une étape qui est absente dans le code *esclave* : l'émission du signal de synchronisation à 80 kHz. En effet, la carte *esclave* n'a pas besoin d'émettre ce signal puisque sa broche 3 est directement connectée à la broche 2 de l'Arduino *maître*, ce qui lui permet de se synchroniser sur le signal du *maître*.

**Remarque :** il est à noter que nous avons fait le choix d'utiliser un signal de synchronisation à 80 kHz et non plus à 40 kHz comme précédemment. Cela résulte uniquement de contraintes informatiques liées au cadencement du processeur de la carte Arduino et permet d'obtenir, en utilisant la méthode ci-après présentée, un signal à 40 kHz pour les 64 transducteurs

## Synchronisation

La synchronisation des deux cartes début par un court protocole d'amorçage sur la liaison série. Avant tout échange automatique, la carte maître attend qu'un caractère (quelconque) soit envoyé depuis le moniteur série de l'utilisateur ; ce premier envoi agit comme un déclencheur manuel permettant de signaler que l'opérateur est prêt et que le système peut s'amorcer.

Lorsque ce caractère est reçu, le maître transmet alors l'octet 0x55 à l'esclave. De son côté, l'esclave reste bloqué tant qu'il n'a pas reçu cette valeur, ce qui garantit qu'il ne démarre pas prematurely. Une fois 0x55 reconnu, l'esclave confirme implicitement sa disponibilité en allumant sa LED 13 et les deux cartes basculent dans leur boucle synchronisée.

À partir de ce point, l'horloge commune est fournie par le maître sous forme d'un signal carré de 80 kHz généré par le Timer 3. Chaque front descendant de ce signal, détecté sur INT5, incrémenté sur les deux cartes le compteur `sync_cnt`, qui sert alors de référence temporelle partagée. Grâce à ce double mécanisme — déclenchement manuel, handshake maître-esclave via 0x55, puis synchronisation par horloge matérielle — les deux systèmes démarrent exactement en même temps et exécutent ensuite leurs mises à jour de sorties en parfaite simultanéité.

## Emission

Une fois la synchronisation établie, les deux cartes peuvent commencer la phase d'émission. Chacune dispose d'un tableau `buffer` contenant les états logiques à appliquer successivement aux dix ports parallèles raccordés aux transducteurs. Ces valeurs représentent les motifs d'excitation ultrasonore en haute résolution temporelle.

Lorsqu'un front descendant du signal de synchronisation est détecté, la fonction `wait_ticks()` assure que chaque mise à jour survient exactement au bon instant, garantissant un timing strictement identique entre le maître et l'esclave. À chaque tick, les cartes envoient simultanément les tranches successives du motif grâce à la macro `OUTPUT_WAVE()`, qui écrit en une seule instruction l'ensemble des dix octets sur les ports matériels.

L'organisation en pointeurs `emittingPointerH` et `emittingPointerL` permet de structurer les données en deux blocs correspondant aux étapes hautes et basses du motif. Ce mécanisme d'émission, basé sur l'écriture directe dans les registres et synchronisé par une horloge externe, assure une génération parfaitement alignée des ondes ultrasonores, condition indispensable pour obtenir un champ acoustique cohérent et stable.

### 4.4.3 Prise en main pratique

L'utilisation des deux Arduino n'étant pas à cette étape du projet optimale, il faut absolument veiller à réaliser les étapes suivantes dans l'ordre et en respectant précisément les recommandations à venir.

## Téléversement des codes sur les 2 Arduino

La première étape consiste à téléverser les codes sur les 2 Arduino. Pour ce faire, il faut tout d'abord avoir repéré quel Arduino est connecté à quel port COM de l'ordinateur. Pour obtenir cette information, il suffit de connecter séparément chaque Arduino afin d'identifier à quel port il est associé. Une fois cette information obtenue, vous pouvez téléverser le code Arduino Maître sur l'Arduino contrôlant la plaque du bas et le code Esclave sur l'Arduino contrôlant la plaque du haut.

## Connexion des deux Arduino

Il est maintenant temps de connecter l'Arduino maître (contrôlant la plaque du bas) à l'Arduino esclave (contrôlant la plaque du haut). Pour ce faire, il suffit de faire correspondre le connecteur mâle à 6 pins de la carte maître sur le connecteur femelle à 6 pins de la carte esclave en s'assurant que l'on connecte bien le port TX de la carte maître au port RX de la carte esclave (cf. gravures à l'arrière du PCB). Il est également possible d'utiliser un câble en nappe disposant de 6 connecteurs mâle et 6 connecteur femelle, ce qui permet d'augmenter la flexibilité de l'installation en réduisant les contraintes liées à la connexion des deux cartes. Cette solution est présentée en figure 66.

**Remarque :** il est impossible de téléverser le code sur l'Arduino esclave lorsque ce dernier a été connecté à l'Arduino maître via son port RX car l'utilisation de ce dernier bloque l'accès à l'ordinateur au port série UART.

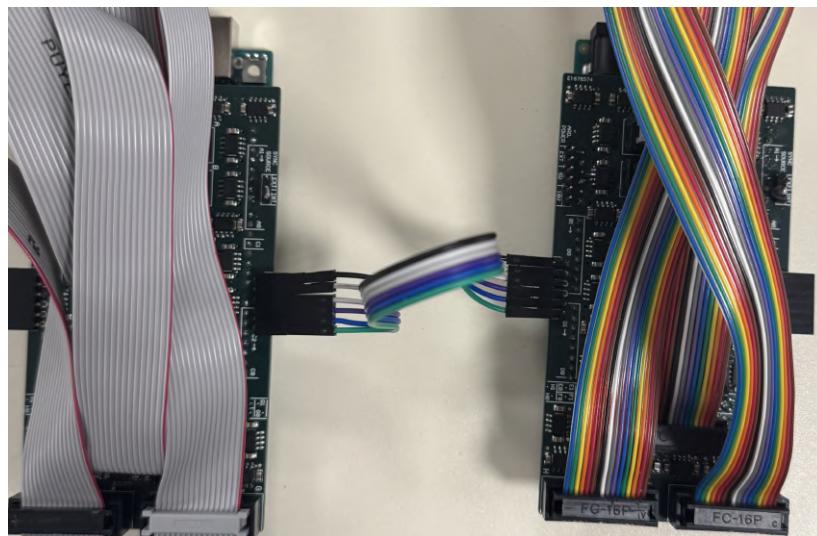


FIGURE 65 – Connexion des deux cartes Arduino

## Lancement de l'émission sur les deux matrices

Une fois les connexions réalisées, il ne reste qu'à lancer l'émission des signaux acoustiques sur les 2 matrices de 64 transducteurs. Pour cela, il suffit d'envoyer via le moniteur série de l'Arduino maître un caractère quelconque (lettre ou chiffre) à l'Arduino esclave. L'envoi de ce caractère permet de synchroniser les 2 Arduino et d'initier l'émission des signaux.

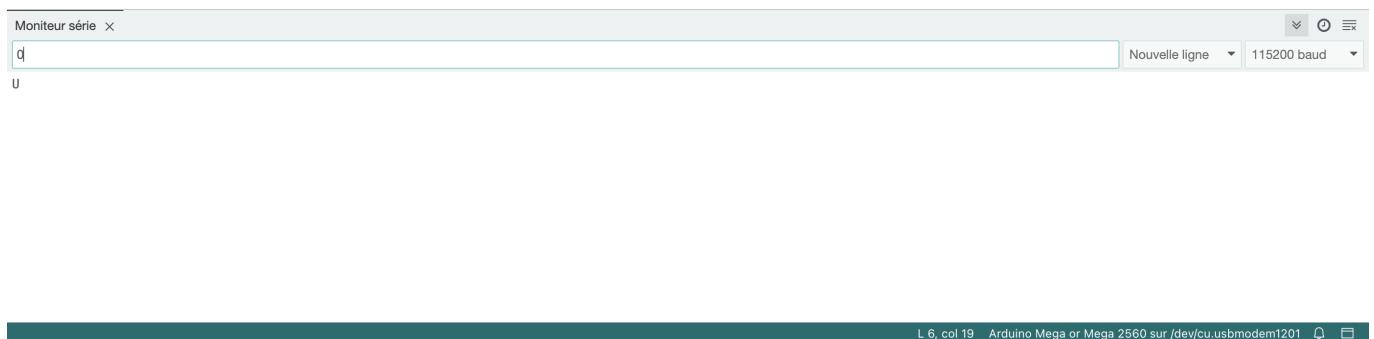


FIGURE 66 – Envoi d'un caractère via le moniteur série

## Alimentation du dispositif

L'étape finale consiste à alimenter le dispositif avec une tension continue comprise entre 25 et 35 Vpp. Pour cela, connecter d'abord les fiches banane noire et rouge du câble d'alimentation respectivement sur la borne – et la borne + de l'alimentation continue. Ensuite, connecter la fiche jack du câble d'alimentation à l'un des deux shield d'amplification (maître ou esclave, les deux étant interconnectés). Pour cette étape, se référer au paragraphe 3.4.2 et particulièrement à la figure 40. Enfin, allumer l'alimentation, régler l'intensité environ au tiers du maximum et enfin régler la tension à environ 25 V.

**Remarque :** par expérience, il est possible qu'un court-circuit apparaisse à la suite d'une mauvaise manipulation du matériel. Ce dernier est souvent repérable à l'aide de l'alimentation continue : si le voyant "CC" de cette dernière est allumé ou l'on entend les calibres de cette dernière passer les uns à la suite des autres, l'éteindre immédiatement afin de préserver le matériel.

Il ne reste plus qu'à faire léviter !

## Procédure d'arrêt

Pour éteindre le système, il suffit de commencer par éteindre l'alimentation continue, puis de déconnecter les Arduino de l'ordinateur.

## 4.5 Résultats

Comme vous pourrez le constater en comparant une vidéo à une plaque et une vidéo à deux plaques disponibles sur notre github, la stabilité est grandement améliorée avec le prototype à deux plaques.

Il est même possible de sélectionner des billes de différentes tailles, cela n'affectant plus tellement la qualité de la lévitation du à la nette augmentation de la force appliquée sur l'objet en lévitation.

Voici une image d'un objet en lévitation avec le prototype final :



FIGURE 67 – Lévitration d'une bille de polystyrène avec le prototype à deux plaques

## 5 Conclusion

Ce projet de 18 mois mené aux côtés d'enseignants de l'IEMN nous a permis de finement explorer le phénomène de la lévitation acoustique, depuis la compréhension des principes physiques fondamentaux jusqu'à la réalisation de prototypes capables de manipuler de la matière sans contact.

Notre démarche a été structurée autour de trois grandes phases :

- **La Phase 1 (TinyLev)** a validé la faisabilité du piégeage acoustique par ondes stationnaires générée par la géométrie du prototype. Bien que limité à des positions fixes, ce prototype a posé les bases de notre projet en nous permettant de nous familiariser avec les outils de lévitation acoustique.
- **La Phase 2 (Lévitateur à une plaque)** a marqué un tournant logiciel et théorique. Grâce à l'implémentation de théories développées par notre équipe projet (*Vortex* et *Optimizer*), nous sommes passés d'un piège statique à un contrôle dynamique des phases, ouvrant la voie au déplacement de particules.
- **La Phase 3 (Lévitateur à deux plaques)** a permis l'aboutissement technique du projet. En augmentant significativement la puissance acoustique tout en optimisant la géométrie des champs grâce au contrôle indépendant des transducteurs, nous avons obtenu une stabilité de piégeage supérieure, permettant d'envisager la manipulation d'objets plus denses ou de plusieurs particules simultanément.

Au-delà de la réussite technique, ce travail a mis en lumière les défis inhérents à la discrétisation des sources acoustiques et à la précision de la synchronisation électronique. Les algorithmes d'optimisation que nous avons développés offrent un compromis robuste entre temps de calcul et qualité du confinement.

L'étude de marché et l'analyse des brevets confirment que cette technologie ne se limite pas à une curiosité de laboratoire. Les applications dans le secteur **pharmaceutique** (analyse de principes actifs sans contamination) et en **micro-chimie** (fusion de gouttes en lévitation) représentent des vecteurs de développement concrets. C'est d'ailleurs pour cette raison que nous avons choisi comme application de notre projet le rapprochement sans contact de deux gouttes de liquide.

En conclusion, ce projet a été une expérience formatrice complète, alliant gestion de projet, modélisation informatique, théorie acoustique, et conception mécanique (CAO), offrant ainsi un début de solution viable pour les besoins de manipulation stérile de demain.

## 6 Bibliographie

### Références

- [1] Baudoin, M. et Thomas, J.-L. (2019). *Acoustic Tweezers for Particle and Fluid Micromanipulation*. Annual Reviews.
- [2] Gong, Z. (2019). *Particle Assembly with Synchronized Acc Tweezers*. Physical Review Applied.
- [3] Marzo, A. (2015). *Holographic acoustic elements for manipulation of levitated objects*. Nature communications.
- [4] Marzo, A. (2018). *Ultraino : An Open Phased-Array System for Narrowband Airborne Ultrasound Transmission*. IEEE TRANSACTIONS ON ULTRASONICS.
- [5] Morales R. (2021). *Generating Airborne Ultrasonic Amplitude Patterns Using an Open Hardware Phased Array*. Applied sciences.
- [6] John, F. (2022). *Particle-Size Effect in Airborne Standing-Wave Acoustic Levitation : Trapping Particles at Pressure Antinodes*. Physical review applied.

#### 6.1 Notre GitHub

Github : [Lien vers github](#)

#### 6.2 Vers la phase 3

- Source 1 : [TinyLev](#)
- Source 2 : [Applications pharmaceutiques](#)
- Source 3 : [Lévitation de liquides](#)
- Source 4 : [Réactions chimiques « sans contact »](#)
- Source 5 : [Brevet : FR3144022 - Dispositif de lévitation acoustique](#)
- Source 6 : [Onde stationnaire](#)
- Source 7 : [Onde progressive](#)

## 7 Annexes

### 7.1 ArduinoMega

Cette documentation détaille le fonctionnement du programme DriverMega.ino. Ce code est optimisé pour piloter des transducteurs acoustiques via un Arduino Mega, en utilisant une gestion directe des ports et une synchronisation matérielle précise.

Voici le code étudié :

```
1 #include <avr/sleep.h>
2 #include <avr/power.h>
3
4 #define N_PATTERNS 32
5
6 #define N_PORTS 10
7 #define N_DIVS 10
8
9 // ports A C L B K F H D G J
10
11 #define COMMAND_SWITCH 0b00000000
12 #define COMMAND_DURATION 0b00110000
13 #define MASK_DURATION 0b00111111
14 #define COMMAND_COMMITDURATIONS 0b00010000
15
16 #define WAIT(a) __asm__ __volatile__ ("nop")
17 #define OUTPUT_WAVE(pointer, d) PORTA = pointer[d * N_PORTS + 0]; PORTC = pointer
18 [d * N_PORTS + 1]; PORTL = pointer[d * N_PORTS + 2]; PORTB = pointer[d *
19 N_PORTS + 3]; PORTK = pointer[d * N_PORTS + 4]; PORTF = pointer[d * N_PORTS +
20 5]; PORTH = pointer[d * N_PORTS + 6]; PORTD = pointer[d * N_PORTS + 7]; PORTG
21 = pointer[d * N_PORTS + 8]; PORTJ = pointer[d * N_PORTS + 9]
22
23 static byte bufferA[N_PATTERNS * N_DIVS * N_PORTS];
24 static byte bufferB[N_PATTERNS * N_DIVS * N_PORTS];
25
26 void setup()
27 {
28     // set as output ports A C L B K F H D G J
29     DDRA = DDRC = DDRL = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ = 0xFF;
30     // low signal on all of them
31     PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD = PORTG = PORTJ =
32     0x00;
33
34     // clear the buffers
35     for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {
36         bufferA[i] = bufferB[i] = 0;
37     }
38
39     // initial pattern
40     for (int i = 0; i < (N_PORTS * N_DIVS / 2); ++i) {
41         bufferA[i] = 0xFF;
42     }
43
44     // a patterns of 01s at pin 22, for debugging and adjusting times
45     for (int i = 0; i < N_DIVS; ++i) {
46         if (i % 2 == 0) {
47             bufferA[i * N_PORTS] |= 0b00001000; // HIGH
48             bufferA[(N_PORTS * N_DIVS / 2) + i * N_PORTS] |= 0b00001000; // HIGH in
49             lower half
50         } else {
51             bufferA[i * N_PORTS] &= 0b11110111; // LOW
52             bufferA[(N_PORTS * N_DIVS / 2) + i * N_PORTS] &= 0b11110111; // LOW in
53         }
54     }
55 }
```

```

    lower half
47 }
48 }

// generate a sync signal of 40 khz in pin 2
50 pinMode(2, OUTPUT);
51 noInterrupts(); // disable all interrupts
52 TCCR3A = bit(WGM10) | bit(WGM11) | bit(COM1B1); // fast PWM, clear OC1B on
53   compare
54 TCCR3B = bit(WGM12) | bit(WGM13) | bit(CS10); // fast PWM, no prescaler
55 OCR3A = (F_CPU / 40000L) - 5; // should only be -1 but fine tuning with the
56   scope determined that -5 gave 40 kHz almost exactly
57 OCR3B = (F_CPU / 40000L) / 2;
58 interrupts(); // enable all interrupts

59 // sync in signal at pin 3
60 pinMode(3, INPUT_PULLUP); // please connect pin 3 to pin 2
61
62 // disable everything that we do not need
63 ADCSRA = 0; // ADC
64 power_adc_disable();
65 power_spi_disable();
66 power_twi_disable();
67 power_timer0_disable();
68 power_usart1_disable();
69 power_usart2_disable();
70 power_usart3_disable();
71 // power_usart0_disable();

72 Serial.begin(115200);

73 byte bReceived = 0; // octet recu via Serial
74 bool byteReady = false; // drapeau : un octet est disponible ?
75 bool isSwitch = false; // drapeau : commande SWITCH recue ?
76 bool isPatternForMe = false; // drapeau : cet octet fait partie du pattern a
77   charger ?
78 bool isDuration = false; // drapeau : cet octet contient un fragment de duree ?
79 bool isCommitDurations = false; // drapeau : commande de validation des durees
80   recue ?
81 byte nextMsg = 0; // valeur a renvoyer par defaut si commande inconnue
82 int writtingIndex = 0; // index pour l'ecriture progressive d'un pattern

83
84 bool emittingA = true; // indique quel buffer (A ou B) est actuellement emis
85 // Pointeurs vers le debut (H) et le milieu (L) du buffer A
86 byte * emittingPointerH = &bufferA[0];
87 byte * emittingPointerL = &bufferA[N_PORTS * N_DIVS / 2];
88 // Sauvegarde des pointeurs "a zero" pour pouvoir reinitialiser apres un switch
89 byte * emittingPointerZeroH = &bufferA[0];
90 byte * emittingPointerZeroL = &bufferA[N_PORTS * N_DIVS / 2];
91 // Pointeurs pour ecrire dans le buffer de reception (l'autre buffer)
92 byte * readingPointerH = &bufferB[0];
93 byte * readingPointerL = &bufferB[N_PORTS * N_DIVS / 2];

94
95 byte durations[N_PATTERNS];
96 byte durationsBuffer[N_PATTERNS];
97 // On met tout a zero
98 for (int i = 0; i < N_PATTERNS; ++i) {
99   durations[i] = durationsBuffer[i] = 0;
}
100 // On force la duree du pattern 0 a 1 division temporelle par defaut
101 durations[0] = durationsBuffer[0] = 1;

102 byte currentPattern = 0; // motif courant dans la sequence

```

```

105 byte currentPeriods = 0; // compte les divisions emises dans ce motif
106 byte durationsPointer = 0; // index bit-a-bit lors de la reception des durees
107 byte currentDuration = 0; // duree lue pour le motif en cours
108 bool patternComplete = false; // drapeau : motif completement emis ?
109 bool lastPattern = false; // drapeau : on est sur le dernier motif (31) ?
110 byte nextPattern = 0; // motif suivant (courant +1)
111 byte nextDuration = 0; // duree du motif suivant
112 bool returnToFirstPattern = false; // drapeau : la sequence doit revenir au
   motif 0 ?
113
114 LOOP:
115 // Attendre front descendant
116 while (!(PINE & 0b00100000)); // puis attendre que ca redescende (0)
117
118 OUTPUT_WAVE(emittingPointerH, 0); // emission des 8 bits de chaque port a la
   division 0
119 byteReady = Serial._dataAvailable(); // on verifie si un octet est deja arrive
   dans le buffer materiel du Serial
120 OUTPUT_WAVE(emittingPointerH, 1); // emission de la division 1
121 bReceived = Serial._peekData(); // on lit l'octet en tete si present, mais on
   ne le retire pas du buffer.
122 OUTPUT_WAVE(emittingPointerH, 2); // emission de la division 2
123 isSwitch = bReceived == COMMAND_SWITCH; // si l'octet cumule vaut exactement 0
   x00, on prepare le flag "switch buffer".
124 isCommitDurations = bReceived == COMMAND_COMMITDURATIONS; // meme principe pour
   la validation des durees
125 OUTPUT_WAVE(emittingPointerH, 3); // emission de la division 3
126 isPatternForMe = (bReceived & 0b00001111) == 1; // on regarde les 4 LSB pour
   voir si ce byte contient un fragment du pattern
127 ++currentPeriods; // on incremente le compteur de periodes deja emises pour le
   motif courant.
128 OUTPUT_WAVE(emittingPointerH, 4); // emission de la division 4
129 nextMsg = bReceived - 1; // pre-calcule la reponse generique (ack ou message d'
   erreur minimaliste.
130 OUTPUT_WAVE(emittingPointerL, 0); // emission de la division 0 (seconde moitie)
131 isDuration = (bReceived & MASK_DURATION) == COMMAND_DURATION;
132 nextPattern = currentPattern + 1;
133 OUTPUT_WAVE(emittingPointerL, 1);
134 nextDuration = durations[nextPattern];
135 OUTPUT_WAVE(emittingPointerL, 2);
136 patternComplete = (currentPeriods == durations[currentPattern]);
137 OUTPUT_WAVE(emittingPointerL, 3);
138 lastPattern = (currentPattern + 1 == N_PATTERNS);
139 returnToFirstPattern = nextDuration == 0;
140 OUTPUT_WAVE(emittingPointerL, 4);
141
142 if (patternComplete) {
143     currentPeriods = 0;
144     ++currentPattern;
145     if (lastPattern || returnToFirstPattern) {
146         currentPattern = 0;
147         emittingPointerH = emittingPointerZeroH;
148         emittingPointerL = emittingPointerZeroL;
149     } else {
150         emittingPointerH += (N_DIVS * N_PORTS);
151         emittingPointerL += (N_DIVS * N_PORTS);
152     }
153 }
154
155 if (byteReady) {
156     if (isSwitch) {
157         // --- SWITCH BUFFER ---
158         Serial.write(COMMAND_SWITCH);

```

```

159     emittingA = !emittingA;
160     // On swap A<->B pour emission vs. reception
161     if (emittingA) {
162         emittingPointerH = &bufferA[0];
163         emittingPointerL = &bufferA[N_PORTS * N_DIVS / 2];
164         readingPointerH = &bufferB[0];
165         readingPointerL = &bufferB[N_PORTS * N_DIVS / 2];
166     } else {
167         emittingPointerH = &bufferB[0];
168         emittingPointerL = &bufferB[N_PORTS * N_DIVS / 2];
169         readingPointerH = &bufferA[0];
170         readingPointerL = &bufferA[N_PORTS * N_DIVS / 2];
171     }
172     emittingPointerZeroH = emittingPointerH;
173     emittingPointerZeroL = emittingPointerL;
174
175     writtingIndex = 0;
176     durationsPointer = 0;
177 } else if (isPatternForMe) {
178     // --- RECEPTION D'UN OCTET DE PATTERN ---
179     if (writtingIndex % 2 == 0)
180         readingPointerH[writtingIndex / 2] = bReceived & 0xFO;
181     else {
182         readingPointerH[writtingIndex / 2] |= (bReceived >> 4);
183     }
184     ++writtingIndex;
185 } else if (isDuration) {
186     // --- RECEPTION D'UN OCTET DE DUREE ---
187     Serial.write(bReceived);
188     if (durationsPointer % 4 == 0) {
189         durationsBuffer[durationsPointer / 4] = bReceived & 0b11000000;
190     } else {
191         durationsBuffer[durationsPointer / 4] |= (bReceived & 0b11000000) >> (
durationsPointer % 4 * 2);
192     }
193     ++durationsPointer;
194 } else if (isCommitDurations) {
195     // --- COMMIT DES DUREES ---
196     Serial.write(bReceived);
197     for (int i = 0; i < N_PATTERNS; ++i) {
198         durations[i] = durationsBuffer[i];
199     }
200     durationsPointer = 0;
201 } else {
202     // --- COMMANDE INCONNUE ---
203     Serial.write(nextMsg);
204 }
205 // On consomme enfin l'octet
206 Serial._discardByte();
207 }
208
209 goto LOOP;
210 }
211
212 void loop() {}

```

Listing 4 – Source : DriverMega.ino

## 7.2 NewDriverMega

Le code suivant assure la génération des signaux PWM à 40 kHz et le pilotage des 10 ports de l'Arduino Mega pour la gestion des phases des transducteurs.

```

1 #include <avr/sleep.h>
2 #include <avr/power.h>
3
4 #define N_PATTERNS 2
5 #define N_PORTS 10
6
7 // Nombre de periodes 40 kHz avant de passer au pattern suivant (ex: 400000 = 10s
8 )
9
10#define PATTERN_PERIODS 400000
11
12#define OUTPUT_WAVE(pointer, d) \
13PORTA = pointer[d * N_PORTS + 0]; \
14PORTC = pointer[d * N_PORTS + 1]; \
15PORTL = pointer[d * N_PORTS + 2]; \
16PORTB = pointer[d * N_PORTS + 3]; \
17PORTK = pointer[d * N_PORTS + 4]; \
18PORTF = pointer[d * N_PORTS + 5]; \
19PORTH = pointer[d * N_PORTS + 6]; \
20PORTD = pointer[d * N_PORTS + 7]; \
21PORTG = pointer[d * N_PORTS + 8]; \
22PORTJ = pointer[d * N_PORTS + 9];
23
24 static const byte patterns[N_PATTERNS * N_DIVS * N_PORTS] = {
25     // Pattern 1
26     0x42, 0xEE, 0x42, 0xB5, 0x07, 0xDA, 0x69, 0x8A, 0x01, 0x02,
27     0x45, 0xE9, 0x44, 0xB5, 0x17, 0xBA, 0x51, 0x86, 0x01, 0x02,
28     0x5D, 0xE9, 0x45, 0xAD, 0x17, 0xA2, 0x11, 0x86, 0x01, 0x02,
29     0xBD, 0xE9, 0x45, 0x4B, 0x1F, 0xA4, 0x12, 0x06, 0x04, 0x02,
30     0xBD, 0xE9, 0xB5, 0x4A, 0x18, 0xA5, 0x12, 0x06, 0x06, 0x01,
31     0xBD, 0x11, 0xBD, 0x4A, 0xF8, 0x25, 0x12, 0x05, 0x06, 0x01,
32     0xBA, 0x16, 0xBB, 0x4A, 0xE8, 0x45, 0x2A, 0x09, 0x06, 0x01,
33     0xA2, 0x16, 0xBA, 0x52, 0xE8, 0x5D, 0x6A, 0x09, 0x06, 0x01,
34     0x42, 0x16, 0xBA, 0xB4, 0xE0, 0x5B, 0x69, 0x89, 0x03, 0x01,
35     0x42, 0x16, 0x4A, 0xB5, 0xE7, 0x5A, 0x69, 0x89, 0x01, 0x02,
36
37     // Pattern 2
38     0x42, 0xEE, 0x42, 0xB5, 0x07, 0xDA, 0x69, 0x8A, 0x01, 0x02,
39     0x45, 0xE9, 0x44, 0xB5, 0x17, 0xBA, 0x51, 0x86, 0x01, 0x02,
40     0x5D, 0xE9, 0x45, 0xAD, 0x17, 0xA2, 0x11, 0x86, 0x01, 0x02,
41     0xBD, 0xE9, 0x45, 0x4B, 0x1F, 0xA4, 0x12, 0x06, 0x04, 0x02,
42     0xBD, 0xE9, 0xB5, 0x4A, 0x18, 0xA5, 0x12, 0x06, 0x06, 0x01,
43     0x11, 0xBD, 0x4A, 0xF8, 0x25, 0x12, 0x05, 0x06, 0x01,
44     0xBA, 0x16, 0xBB, 0x4A, 0xE8, 0x45, 0x2A, 0x09, 0x06, 0x01,
45     0xA2, 0x16, 0xBA, 0x52, 0xE8, 0x5D, 0x6A, 0x09, 0x06, 0x01,
46     0x42, 0x16, 0xBA, 0xB4, 0xE0, 0x5B, 0x69, 0x89, 0x03, 0x01,
47     0x42, 0x16, 0x4A, 0xB5, 0xE7, 0x5A, 0x69, 0x89, 0x01, 0x02
48 };
49
50 static byte bufferA[N_DIVS * N_PORTS];
51 static byte bufferB[N_DIVS * N_PORTS];
52
53 void setup() {
54     // Configuration des ports en sortie
55     DDRA = DDRC = DDRL = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ = 0xFF;
56     PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD = PORTG = PORTJ =
57     0x00;
58     for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {

```

```

59     bufferA[i] = 0;
60     bufferB[i] = 0;
61 }
62
63 // Pattern initial (sinus en phase)
64 for (int i = 0; i < (N_PORTS * N_DIVS / 2); ++i) {
65     bufferA[i] = 0xFF;
66 }
67
68 // PWM 40 kHz sur pin 2
69 pinMode(2, OUTPUT);
70 noInterrupts();
71 TCCR3A = bit(WGM10) | bit(WGM11) | bit(COM1B1);
72 TCCR3B = bit(WGM12) | bit(WGM13) | bit(CS10);
73 OCR3A = (F_CPU / 40000L) - 3;
74 OCR3B = (F_CPU / 40000L) / 2;
75 interrupts();
76
77 pinMode(3, INPUT_PULLUP); // Sync in (connectee a pin 2)
78
79 // Desactivation peripheriques inutiles pour optimiser la boucle
80 ADCSRA = 0;
81 power_adc_disable();
82 power_spi_disable();
83 power_twi_disable();
84 power_timer0_disable();
85 power_usart1_disable();
86 power_usart2_disable();
87 power_usart3_disable();
88
89 bool useA = true;
90 unsigned long currentPeriods = 0;
91 byte currentPatternIndex = 0;
92
93 byte * activeBuffer = bufferA;
94 byte * nextBuffer = bufferB;
95
96 for (int i = 0; i < N_PORTS * N_DIVS; i++) bufferA[i] = patterns[i];
97 for (int i = 0; i < N_PORTS * N_DIVS; i++) bufferB[i] = patterns[(currentPatternIndex + 1) * N_PORTS * N_DIVS + i];
98
99 byte * emittingPointerH = &activeBuffer[0];
100 byte * emittingPointerL = &activeBuffer[N_PORTS * N_DIVS / 2];
101
102 LOOP:
103 while (PINE & 0b00100000); // Attente synchro
104
105 OUTPUT_WAVE(emittingPointerH, 0); WAIT();
106 OUTPUT_WAVE(emittingPointerH, 1); WAIT();
107 OUTPUT_WAVE(emittingPointerH, 2); WAIT();
108 OUTPUT_WAVE(emittingPointerH, 3); WAIT();
109 OUTPUT_WAVE(emittingPointerL, 0); WAIT();
110 OUTPUT_WAVE(emittingPointerL, 1); WAIT();
111 OUTPUT_WAVE(emittingPointerL, 2); WAIT();
112 OUTPUT_WAVE(emittingPointerL, 3); WAIT();
113
114 currentPeriods++;
115 if (currentPeriods >= PATTERN_PERIODS) {
116     currentPeriods = 0;
117     useA = !useA;
118     activeBuffer = useA ? bufferA : bufferB;
119     nextBuffer = useA ? bufferB : bufferA;
120 }
```

```

121     currentPatternIndex++;
122     if (currentPatternIndex >= N_PATTERNS) currentPatternIndex = 0;
123
124     emittingPointerH = &activeBuffer[0];
125     emittingPointerL = &activeBuffer[N_PORTS * N_DIVS / 2];
126
127     int nextPatternIndex = currentPatternIndex + 1;
128     if (nextPatternIndex >= N_PATTERNS) nextPatternIndex = 0;
129
130     for (int i = 0; i < N_PORTS * N_DIVS; i++)
131         nextBuffer[i] = patterns[nextPatternIndex * N_PORTS * N_DIVS + i];
132     }
133     goto LOOP;
134 }
135
136 void loop() {}

```

Listing 5 – Programme de gestion des patterns et des phases (Lévitation Acoustique)

### 7.2.1 Explications des modifications

#### Suppression des fonctions superflues

Notre volonté première en modifiant ce code est de simplifier au maximum le code de départ. Cela passe par la suppression des commandes superflues pour notre application :

```

1 #define COMMAND_SWITCH 0b00000000
2 #define COMMAND_DURATION 0b00110000
3 #define MASK_DURATION 0b00111111
4 #define COMMAND_COMMITDURATIONS 0b00010000

```

Ces variables étaient précédemment utilisées dans le but de permettre un envoi en temps réel des phases alors calculées avec le logiciel Java. Or nous souhaitons nous détacher de ce logiciel et n'avons pas besoin de faire du temps réel. Nous pouvons donc nous permettre de supprimer ces commandes.

Il en va de même pour ces variables-ci :

```

1 byte bReceived = 0;
2 bool byteReady = false;
3 bool isSwitch = false;
4 bool isPatternForMe = false;
5 bool isDuration = false;
6 bool isCommitDurations = false;
7 byte nextMsg = 0;
8 int writingIndex = 0;
9
10 byte* emittingPointerZeroH = &bufferA[0];
11 byte* emittingPointerZeroL = &bufferA[N_PORTS * N_DIVS / 2];
12 byte* readingPointerH = &bufferB[0];
13 byte* readingPointerL = &bufferB[N_PORTS * N_DIVS / 2];
14
15 byte durations[N_PATTERNS];
16 byte durationsBuffer[N_PATTERNS];
17
18 byte currentPeriods = 0;
19 byte durationsPointer = 0;
20 byte currentDuration = 0;
21 bool patternComplete = false;
22 bool lastPattern = false;

```

```
23 byte nextDuration = 0;
24 bool returnToFirstPattern = false;
```

Certaines variables sont cependant similaires au premier code :

### I- Le booléen permettant de savoir quel buffer est actuellement utilisé

Avant on utilisait :

```
1 bool emittingA = true;
```

Maintenant on utilise :

```
1 bool useA = true;
```

### II- La variable permettant d'indiquer le pattern actuellement émis

Avant on utilisait :

```
1 byte currentPattern = 0;
```

Maintenant on utilise :

```
1 byte currentPatternIndex = 0;
```

### III- Le buffer suivant à traiter

Avant on utilisait :

```
1 byte nextPattern = 0;
```

Maintenant on utilise :

```
1 byte* activeBuffer = bufferA;
2 byte* nextBuffer = bufferB;
```

## Choix de la durée du wait en fonction du processeur Arduino

Le code DriverMega utilise une fonction permettant d'attendre à chaque envoi de pattern afin de combler les latences intrinsèques du processeur Arduino, permettant ainsi d'assurer l'envoi synchrone des informations. La fonction utilisée est la suivante :

```
1 #define WAIT(a) __asm__ __volatile__("nop")
```

### Pourquoi utiliser cette fonction ?

Afin de remplacer les fonctions/variables du programme initial qui entraînaient des tours de processeur :

```
1 OUTPUT_WAVE(emittingPointerH, 0);
2 byteReady = Serial._dataAvailable();
3 OUTPUT_WAVE(emittingPointerH, 1);
4 bReceived = Serial._peekData();
5 OUTPUT_WAVE(emittingPointerH, 2);
6 isSwitch = bReceived == COMMAND_SWITCH;
7 isCommitDurations = bReceived == COMMAND_COMMITDURATIONS;
8 OUTPUT_WAVE(emittingPointerH, 3);
9 isPatternForMe = (bReceived & 0b00001111) == 1;
10 ++currentPeriods;
11 OUTPUT_WAVE(emittingPointerH, 4);
12 nextMsg = bReceived - 1;
13 OUTPUT_WAVE(emittingPointerL, 0);
14 isDuration = (bReceived & MASK_DURATION) == COMMAND_DURATION;
15 nextPattern = currentPattern + 1;
16 OUTPUT_WAVE(emittingPointerL, 1);
```

```

17 nextDuration = durations[nextPattern];
18 OUTPUT_WAVE(emittingPointerL, 2);
19 patternComplete = (currentPeriods == durations[currentPattern]);
20 OUTPUT_WAVE(emittingPointerL, 3);
21 lastPattern = (currentPattern+1 == N_PATTERNS);
22 returnToFirstPattern = nextDuration == 0;
23 OUTPUT_WAVE(emittingPointerL, 4);

```

Le code précédent est ainsi remplacé par cette partie du nouveau code :

```

1 LOOP:
2 while(PINE & 0b00100000);
3 OUTPUT_WAVE(emittingPointerH, 0); WAIT();
4 OUTPUT_WAVE(emittingPointerH, 1); WAIT();
5 OUTPUT_WAVE(emittingPointerH, 2); WAIT();
6 OUTPUT_WAVE(emittingPointerH, 3); WAIT();
7 OUTPUT_WAVE(emittingPointerL, 0); WAIT();
8 OUTPUT_WAVE(emittingPointerL, 1); WAIT();
9 OUTPUT_WAVE(emittingPointerL, 2); WAIT();
10 OUTPUT_WAVE(emittingPointerL, 3); WAIT();

```

**Choix de la durée du wait en fonction du processeur Arduino** En fonction du code et des fonctions/variables utilisées, le nombre de “nop” en entrée de la fonction WAIT doit être modifié. **I-** Le cas où l'on souhaite envoyer deux patterns La fonction WAIT() est alors définie comme :

```

1 #define WAIT() asm volatile("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t"
                           "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t")

```

## II- Autre cas de figure

La fonction WAIT() est alors définie comme :

```

1 #define WAIT() asm volatile("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t")

```

Notons pour finir que le nombre de “nop” à appliquer est entièrement empirique et doit être déterminé expérimentalement :

- Brancher l'arduino à l'ordinateur ;
- Relier les pins 2 et 3 entre elles pour la synchronisation ;
- Récupérer le signal d'un pin à l'oscilloscope ;
- Vérifier que le créneau est stable à 40kHz et symétrique.

## Liste des variables essentielles qui ont été ajoutées

- `bool useA` : indique quel Buffer remplir ;
- `#define PATTERN_PERIODS 400000` : durée d'un pattern ;
- `unsigned long currentPeriods` : compteur pour le changement de pattern.

## 7.3 Code python : Passage des phases réelles aux phases en bit

```

1 import math as m
2
3 # Constantes
4 N_TRANSDUCTEUR = 80
5 N_DIVS = 10
6 N_PORTS = 10

```

```

7 N_T = N_TRANSDUCTEUR // N_PORTS
8
9 # Initialisation des phases
10
11 Phi_64 = [3.824784098879988, 3.9563941051912463, 0.9894161429899966,
12   4.35675309608393, 4.626883686706485, 4.90978454023457, 2.022950189080672,
13   5.369463916096343, 0.5266272714337508, 0.6483319890872609, 3.9686333960858295,
14   1.0912770348023004, 1.4513674007765582, 1.8438050298816073,
15   2.167972984887574, 5.536229733803326, 3.4772312039174023, 3.568220146716669,
16   0.5779019369622455, 3.9982982817725317, 4.514993420534809, 5.149016140198231,
17   2.418613300188302, 2.6344941491974563, 0.115759217720811, 3.2919640815969275,
18   0.21406068356382157, 3.5087664874080122, 4.171969480114106, 5.662935821196764,
19   5.997133865462268, 2.958481836327309, 6.167426089458775, 2.9912212255826587,
20   6.069124623615765, 2.7744188197715745, 2.1112158270654815, 0.6202494859828213,
21   0.28605144171731817, 3.324703470852277, 2.805954103262184, 2.714965160462917,
22   5.705283370217341, 2.284887025407055, 1.7681918866447777, 1.1341691669813554,
23   3.864572006991284, 3.64869115798213, 5.756558035745836, 5.6348533180923255,
24   2.3145519110937567, 5.191908272377287, 4.831817906403028, 4.439380277297979,
25   4.115212322292012, 0.7469555733762602, 2.4584012082995983, 2.32679120198834,
26   5.293769164189589, 1.9264322110956564, 1.6563016204731014, 1.373400766945016,
27   4.260235118098914, 0.9137213910832431]
28
29
30
31
32
33 # Creation de la matrice binaire oct_T
34 oct_T = [[0 for i in range(N_DIVS)] for j in range(N_TRANSDUCTEUR)]
35
36 for i in range(N_TRANSDUCTEUR):
37     for j in range(N_DIVS):
38         angle = 2 * m.pi * j / N_DIVS + Phi[i]
39         value = m.sin(angle)
40         oct_T[i][j] = 0 if value <= 0 else 1
41
42 pins_inutiles=[
43     53, 48, 59, 58, 57, 68, 67, 66,
44     65, 64, 77, 76, 75, 74, 73, 72
45 ]
46
47 pins_inutiles=pins_inutiles[::-1]
48 for i in pins_inutiles :
49     for j in range(N_DIVS):
50         oct_T[i][j]=0
51
52 # Creation de la structure oct_P
53 oct_P = [[[[] for i in range(N_PORTS)] for j in range(N_DIVS)]]
```

```

54
55     for j in range(N_DIVS):
56         for i in range(N_PORTS):
57             for k in range(N_T):
58                 oct_P[j][i].append(oct_T[i * N_T + k][j])
59
60 # Construction du buffer
61 buffer = [0 for i in range(N_DIVS * N_PORTS)]
62
63 for i in range(N_DIVS):
64     for j in range(N_PORTS):
65         value = 0
66         for k in range(8):
67             value = (value << 1) | (oct_P[i][j][k] & 1)
68         buffer[((i - 1) % N_DIVS) * N_PORTS + j] = value

```

## 7.4 Sketch Arduino - prototype à une plaque

```

1
2 #include <avr/sleep.h>
3 #include <avr/power.h>
4
5 #define N_PATTERNS 2
6 #define N_PORTS 10
7 #define N_DIVS 10
8 #define PATTERN_PERIODS 400000 // nombre de periodes 40 kHz avant de passer au
      pattern suivant
9
10#define WAIT() __asm__ __volatile__("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t"
11    "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t")
12
13#define OUTPUT_WAVE(pointer, d) \
14    PORTA = pointer[d*N_PORTS + 0]; \
15    PORTC = pointer[d*N_PORTS + 1]; \
16    PORTL = pointer[d*N_PORTS + 2]; \
17    PORTB = pointer[d*N_PORTS + 3]; \
18    PORTK = pointer[d*N_PORTS + 4]; \
19    PORTF = pointer[d*N_PORTS + 5]; \
20    PORTH = pointer[d*N_PORTS + 6]; \
21    PORTD = pointer[d*N_PORTS + 7]; \
22    PORTG = pointer[d*N_PORTS + 8]; \
23    PORTJ = pointer[d*N_PORTS + 9];
24
25 static const byte patterns[N_PATTERNS * N_DIVS * N_PORTS] = {
26     0x42, 0xEE, 0x42, 0xB5, 0x07, 0xDA, 0x69, 0x8A, 0x01, 0x02,
27     0x45, 0xE9, 0x44, 0xB5, 0x17, 0xBA, 0x51, 0x86, 0x01, 0x02,
28     0x5D, 0xE9, 0x45, 0xAD, 0x17, 0xA2, 0x11, 0x86, 0x01, 0x02,
29     0xBD, 0xE9, 0x45, 0x4B, 0x1F, 0xA4, 0x12, 0x06, 0x04, 0x02,
30     0xBD, 0xE9, 0xB5, 0x4A, 0x18, 0xA5, 0x12, 0x06, 0x06, 0x01,
31     0xBD, 0x11, 0xBD, 0x4A, 0xF8, 0x25, 0x12, 0x05, 0x06, 0x01,
32     0xBA, 0x16, 0xBB, 0x4A, 0xE8, 0x45, 0x2A, 0x09, 0x06, 0x01,
33     0xA2, 0x16, 0xBA, 0x52, 0xE8, 0x5D, 0x6A, 0x09, 0x06, 0x01,
34     0x42, 0x16, 0xBA, 0xB4, 0xE0, 0x5B, 0x69, 0x89, 0x03, 0x01,
35     0x42, 0x16, 0x4A, 0xB5, 0xE7, 0x5A, 0x69, 0x89, 0x01, 0x02,
36
37     0x42, 0xEE, 0x42, 0xB5, 0x07, 0xDA, 0x69, 0x8A, 0x01, 0x02,
38     0x45, 0xE9, 0x44, 0xB5, 0x17, 0xBA, 0x51, 0x86, 0x01, 0x02,

```

```

39 0x5D, 0xE9, 0x45, 0xAD, 0x17, 0xA2, 0x11, 0x86, 0x01, 0x02,
40 0xBD, 0xE9, 0x45, 0x4B, 0x1F, 0xA4, 0x12, 0x06, 0x04, 0x02,
41 0xBD, 0xE9, 0xB5, 0x4A, 0x18, 0xA5, 0x12, 0x06, 0x06, 0x01,
42 0xBD, 0x11, 0xBD, 0x4A, 0xF8, 0x25, 0x12, 0x05, 0x06, 0x01,
43 0xBA, 0x16, 0xBB, 0x4A, 0xE8, 0x45, 0x2A, 0x09, 0x06, 0x01,
44 0xA2, 0x16, 0xBA, 0x52, 0xE8, 0x5D, 0x6A, 0x09, 0x06, 0x01,
45 0x42, 0x16, 0xBA, 0xB4, 0xE0, 0x5B, 0x69, 0x89, 0x03, 0x01,
46 0x42, 0x16, 0x4A, 0xB5, 0xE7, 0x5A, 0x69, 0x89, 0x01, 0x02
47 };
48
49 static byte bufferA[N_DIVS * N_PORTS];
50 static byte bufferB[N_DIVS * N_PORTS];
51
52 void setup() {
53   DDRA = DDRC = DDRL = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ = 0xFF;
54   PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD = PORTG = PORTJ =
55   0x00;
56
57   for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {
58     bufferA[i] = 0;
59     bufferB[i] = 0;
60   }
61
62 // --- PWM 40 kHz sur pin 2 ---
63 pinMode(2, OUTPUT);
64 noInterrupts();
65 TCCR3A = bit(WGM10) | bit(WGM11) | bit(COM1B1);
66 TCCR3B = bit(WGM12) | bit(WGM13) | bit(CS10);
67 OCR3A = (F_CPU / 40000L) - 3;
68 OCR3B = (F_CPU / 40000L) / 2;
69 interrupts();
70
71 pinMode(3, INPUT_PULLUP);
72
73 // Desactivation peripheriques
74 ADCSRA = 0;
75 power_adc_disable();
76 power_spi_disable();
77
78 bool useA = true;
79 unsigned long currentPeriods = 0;
80 byte currentPatternIndex = 0;
81
82 byte* activeBuffer = bufferA;
83 byte* nextBuffer = bufferB;
84
85 for(int i=0; i<N_PORTS*N_DIVS; i++) bufferA[i] = patterns[i];
86
87 byte* emittingPointerH = &activeBuffer[0];
88 byte* emittingPointerL = &activeBuffer[N_PORTS * N_DIVS / 2];
89
90 LOOP:
91   while (PINE & 0b00100000); // attente synchro
92
93   OUTPUT_WAVE(emittingPointerH, 0); WAIT();
94   OUTPUT_WAVE(emittingPointerH, 1); WAIT();
95   OUTPUT_WAVE(emittingPointerH, 2); WAIT();
96   OUTPUT_WAVE(emittingPointerH, 3); WAIT();
97   OUTPUT_WAVE(emittingPointerL, 0); WAIT();
98   OUTPUT_WAVE(emittingPointerL, 1); WAIT();
99   OUTPUT_WAVE(emittingPointerL, 2); WAIT();
100  OUTPUT_WAVE(emittingPointerL, 3); WAIT();

```

```

101    currentPeriods++;
102    if(currentPeriods >= PATTERN_PERIODS) {
103        currentPeriods = 0;
104        useA = !useA;
105        activeBuffer = useA ? bufferA : bufferB;
106        nextBuffer = useA ? bufferB : bufferA;
107        // ... (suite logique switch)
108    }
109    goto LOOP;
110}
111
112void loop() {}

```

## 7.5 Communication à deux cartes Arduino

### 7.5.1 Carte *maître*

Voici le code Arduino utilisé pour le contrôle de la carte *maître* :

```

1 #include <avr/sleep.h>
2 #include <avr/power.h>
3
4 #define N_PATTERNS 32
5 #define N_TRANSDUCTEUR 80
6 #define N_PORTS 10
7 #define N_DIVS 10
8 #define N_T 8
9
10 //ports A C L B K F H D G J
11
12 #define WAIT() __asm__ __volatile__("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t"
13     "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t")
14 #define OUTPUT_WAVE(pointer, d) PORTA = pointer[d*N_PORTS + 0]; PORTC = pointer[d*N_PORTS +
15     d*N_PORTS + 1]; PORTL = pointer[d*N_PORTS + 2]; PORTB = pointer[d*N_PORTS +
16     3]; PORTK = pointer[d*N_PORTS + 4]; PORTF = pointer[d*N_PORTS + 5]; PORTH =
17     pointer[d*N_PORTS + 6]; PORTD = pointer[d*N_PORTS + 7]; PORTG = pointer[d*N_PORTS +
18     8]; PORTJ = pointer[d*N_PORTS + 9]
19
20 static byte buffer[ N_DIVS * N_PORTS];
21 volatile uint8_t sync_cnt = 0; // compteur de fronts sur pin 3 (INT5)
22
23 // Attente robuste de N ticks (gère le wrap 0..255 et les ticks manqués)
24 static inline void wait_ticks(uint8_t n, volatile uint8_t* last_seen) {
25     uint8_t start = *last_seen;
26     while ((uint8_t)(sync_cnt - start) < n) { /* spin, interruptions actives */ }
27     *last_seen = sync_cnt;
28 }
29
30 static inline void wait_substep(volatile uint16_t* tref, uint16_t ticks)
31 {
32     uint16_t start = *tref;
33     while ((uint16_t)(TCNT4 - start) < ticks) {
34         // attente active, base de temps = Timer4
35     }
36     *tref = start + ticks;
37 }
38
39 void setup()
40 {
41     //set as output ports A C L B K F H D G J
42     DDRA = DDRC = DDRL = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ = 0xFF;
43     //low signal on all of them

```

```

39 PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD = PORTG = PORTJ =
    0x00;
40
41 //clear the buffers
42 for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {
    buffer[i] = 0;
}
45
46 // INT5 (PE5 = pin 3 sur Mega) sur FRONT DESCENDANT
47 EICRB |= (1 << ISC51); // ISC51:50 = 10 => front descendant
48 EICRB &= ~(1 << ISC50);
49 EIFR = (1 << INTF5); // clear tout pending
50 EIMSK |= (1 << INT5); // enable INT5
51
52 // generate a sync signal of 40khz in pin 2
53 pinMode (2, OUTPUT);
54 noInterrupts(); // disable all interrupts
55 TCCR3A = bit (WGM10) | bit (WGM11) | bit (COM1B1); // fast PWM, clear OC1B on
    compare
56 TCCR3B = bit (WGM12) | bit (WGM13) | bit (CS10); // fast PWM, no prescaler
57 OCR3A = (F_CPU / 80000L) - 1; //should only be -1 but fine tuning with the
    scope determined that -5 gave 40kHz almost exactly
58 OCR3B = (F_CPU / 80000L) / 2;
59 TCCR4A=0;
60 TCCR4B=(1<<CS40);
61 TCNT4=0;
62
63 interrupts(); // enable all interrupts
64
65 //sync in signal at pin 3
66 pinMode(3, INPUT_PULLUP); //please connect pin3 to pin 2
67
68
69 // disable everything that we do not need
70 ADCSRA = 0; // ADC
71 power_adc_disable ();
72 power_spi_disable();
73 power_twi_disable();
74 power_timer0_disable();
75 power_usart1_disable();
76 power_usart2_disable();
77 power_usart3_disable();
78 //power_usart0_disable();
79
80
81 //creation d'une liste (de liste des valeur binaire) que chaque transducteur doit
    prendre 64*10
82 byte buffer[N_DIVS * N_PORTS] = {
83 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
84 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
85 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
86 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
87 0x42, 0x32, 0xCB, 0x4D, 0x5A, 0xBD, 0x10, 0x82, 0x00, 0x01,
88 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
89 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
90 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
91 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
92 0xBD, 0xCD, 0x34, 0xB2, 0xA5, 0x42, 0x6B, 0x0D, 0x07, 0x02
};
93
94
95
96 //initialisation des variables
97 byte* emittingPointerH = & buffer[0];

```

```

98 byte* emittingPointerL = & buffer[N_PORTS * N_DIVS / 2];
99
100 Serial.begin(115200);
101 while(!Serial.available());
102 //Serial.read();
103 Serial.write(0x55);
104
105 //-----boucle-----
106
107 LOOP:
108 {
109     uint8_t last = sync_cnt; // point de depart cale sur l horloge externe
110
111     // Exemple : 1 front (= 25 s si 40 kHz) entre chaque OUTPUT_WAVE
112     wait_ticks(1,&last);
113     OUTPUT_WAVE(emittingPointerH, 0); WAIT();
114     OUTPUT_WAVE(emittingPointerH, 1); WAIT();
115     OUTPUT_WAVE(emittingPointerH, 2); WAIT();
116     OUTPUT_WAVE(emittingPointerH, 3); WAIT();
117     OUTPUT_WAVE(emittingPointerH, 4); WAIT();
118     OUTPUT_WAVE(emittingPointerL, 0); WAIT();
119     OUTPUT_WAVE(emittingPointerL, 1); WAIT();
120     OUTPUT_WAVE(emittingPointerL, 2); WAIT();
121     OUTPUT_WAVE(emittingPointerL, 3); WAIT();
122     OUTPUT_WAVE(emittingPointerL, 4); WAIT();
123
124
125     goto LOOP;
126 }
127 }
128
129 void loop() {}
130
131 ISR(INT5_vect) {
132     sync_cnt++; // 1 tick = 1 front re u depuis le maitre
133 }
134

```

### 7.5.2 Carte *esclave*

De même, voici le code Arduino utilisé pour le contrôle de la carte *esclave* :

```

1 #include <avr/sleep.h>
2 #include <avr/power.h>
3
4 #define N_PATTERNS 32
5 #define N_TRANSDUCTEUR 80
6 #define N_PORTS 10
7 #define N_DIVS 10
8 #define N_T 8
9
10 //ports A C L B K F H D G J
11
12 #define WAIT() __asm__ __volatile__("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t"
13 "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t")
14 #define OUTPUT_WAVE(pointer, d) PORTA = pointer[d*N_PORTS + 0]; PORTC = pointer[d*N_PORTS + 1];
15 //PORTL = pointer[d*N_PORTS + 2]; PORTB = pointer[d*N_PORTS + 3]; PORTK = pointer[d*N_PORTS + 4];
16 //PORTF = pointer[d*N_PORTS + 5]; PORTH = pointer[d*N_PORTS + 6]; PORTD = pointer[d*N_PORTS + 7];
17 //PORTG = pointer[d*N_PORTS + 8]; PORTJ = pointer[d*N_PORTS + 9]
18
19 static byte buffer[ N_DIVS * N_PORTS ];
20 volatile uint8_t sync_cnt = 0; // compteur de fronts sur pin 3 (INT5)
21

```

```

18 // Attente robuste de N ticks (gère le wrap 0..255 et les ticks manqués)
19 static inline void wait_ticks(uint8_t n, volatile uint8_t* last_seen) {
20     uint8_t start = *last_seen;
21     while ((uint8_t)(sync_cnt - start) < n) { /* spin, interruptions actives */ }
22     *last_seen = sync_cnt;
23 }
24
25 static inline void wait_substep(volatile uint16_t* tref, uint16_t ticks)
26 {
27     uint16_t start = *tref;
28     while ((uint16_t)(TCNT4 - start) < ticks) {
29         // attente active, base de temps = Timer4
30     }
31     *tref = start + ticks;
32 }
33
34 void setup()
35 {
36     //set as output ports A C L B K F H D G J
37     DDRA = DDRC = DDRL = DDRB = DDRK = DDRF = DDRH = DDRD = DDRG = DDRJ = 0xFF;
38     //low signal on all of them
39     PORTA = PORTC = PORTL = PORTB = PORTK = PORTF = PORTH = PORTD = PORTG = PORTJ =
40         0x00;
41
42     //clear the buffers
43     for (int i = 0; i < (N_PATTERNS * N_DIVS * N_PORTS); ++i) {
44         buffer[i] = 0;
45     }
46
47     // INT5 (PE5 = pin 3 sur Mega) sur FRONT DESCENDANT
48     EICRB |= (1 << ISC51); // ISC51:50 = 10 => front descendant
49     EICRB &= ~(1 << ISC50);
50     EIFR = (1 << INTF5); // clear tout pending
51     EIMSK |= (1 << INT5); // enable INT5
52     //sync in signal at pin 3
53     pinMode(3, INPUT_PULLUP); //please connect pin3 to pin 2
54
55     // disable everything that we do not need
56     ADCSRA = 0; // ADC
57     power_adc_disable();
58     power_spi_disable();
59     power_twi_disable();
60     power_timer0_disable();
61     power_usart1_disable();
62     power_usart2_disable();
63     power_usart3_disable();
64     //power_usart0_disable();
65
66
67
68
69 //creation d'une liste (de liste des valeur binaire) que chaque transducteur doit
70 //prendre 64*10
71 byte buffer[N_DIVS * N_PORTS] = {
72     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
73     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
74     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
75     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x7B, 0x8F, 0x07, 0x03,
76     0xBD, 0xCD, 0x34, 0xB2, 0xA5, 0x42, 0x6B, 0x0D, 0x07, 0x02,
77     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
78     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```

78 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
79 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
80 0x42, 0x32, 0xCB, 0x4D, 0x5A, 0xBD, 0x10, 0x82, 0x00, 0x01
81 };
82
83
84 //initialisation des variables
85 byte* emittingPointerH = & buffer[0];
86 byte* emittingPointerL = & buffer[N_PORTS * N_DIVS / 2];
87 pinMode(13,OUTPUT);
88 Serial.begin(115200);
89 while(!Serial.available());
90 while(Serial.read()!=0x55);
91 digitalWrite(13,HIGH);
92
93 LOOP:
94 {
95     uint8_t last = sync_cnt; // point de d part cal sur l horloge externe
96
97 // Exemple : 1 front (= 25 s si 40 kHz) entre chaque OUTPUT_WAVE
98 wait_ticks(1,&last);
99 OUTPUT_WAVE(emittingPointerH, 0); WAIT();
100 OUTPUT_WAVE(emittingPointerH, 1); WAIT();
101 OUTPUT_WAVE(emittingPointerH, 2); WAIT();
102 OUTPUT_WAVE(emittingPointerH, 3); WAIT();
103 OUTPUT_WAVE(emittingPointerH, 4); WAIT();
104 OUTPUT_WAVE(emittingPointerL, 0); WAIT();
105 OUTPUT_WAVE(emittingPointerL, 1); WAIT();
106 OUTPUT_WAVE(emittingPointerL, 2); WAIT();
107 OUTPUT_WAVE(emittingPointerL, 3); WAIT();
108 OUTPUT_WAVE(emittingPointerL, 4); WAIT();
109
110
111     goto LOOP;
112 }
113 }
114
115 void loop() {}
116
117 ISR(INT5_vect) {
118     sync_cnt++; // 1 tick = 1 front re u depuis le ma tre
119 }
```

## 7.6 Programme de calcul de phase

### 7.6.1 Méthode optimiser

Voici le code phyton implémentant la méthode optimiser pour le calcul des phases optimale pour un piège dont on choisie la position, avec une seule plaque :

```

1 import cmath
2 import numpy as np
3 import math as m
4 from scipy.optimize import minimize
5 import matplotlib.pyplot as plt
6
7 # === Constantes physiques ===
8 k = 2 * m.pi * 40000 / 343
9 # R_p = 0.0005 ##rayon particule finale
10 R_p = 0.0015 # Rayon des objets actuellement utilisés
11 R_t=0.005 ##rayon transducteur
12 c_p = 2400
```

```

13 rho_p = 28.59
14 c_0 = 343
15 rho_0 = 1.204
16 eps = 1e-4
17
18 # === Grille de transducteurs ===
19 a = 0.01
20 n = 8
21 l = np.linspace(-(n // 2 - 0.5) * a, (n // 2 - 0.5) * a, n)
22 X_j, Y_j = np.meshgrid(l, l)
23 X_j = X_j.flatten()
24 Y_j = Y_j.flatten()
25 Z_j = np.zeros_like(X_j)
26
27 # === Fonctions utiles ===
28
29 def K1(R, c_p, rho_p):
30     V = 4 * m.pi / 3 * R**3
31     a0 = 1 / (c_0**2 * rho_0)
32     ap = 1 / (c_p**2 * rho_p)
33     return V / 4 * (a0 - ap)
34
35 def K2(R, omega, rho_p):
36     V = 4 * m.pi / 3 * R**3
37     return 3 * V / 4 * ((rho_0 - rho_p) / (omega**2 * rho_0 * (rho_0 + 2 * rho_p)))
38
39
40 def D_F_vectorized(xj, yj, zj, x, y, z):
41     theta = np.arctan(np.sqrt((x - xj)**2 + (y - yj)**2) / np.where(np.abs(z - zj) < 1e-12, 1e-12, np.abs(z - zj)))
42     df = np.sin(k * R_p * np.sin(theta)) / np.where(np.sin(k * R_p * np.sin(theta)) == 0, 1e-12, k * R_p * np.sin(theta))
43     df = np.where(np.isnan(df), 1, df)
44     return df
45
46 # === Pre-calculs (ne dependent pas des phases) ===
47 DJ={}
48 DF={}
49 def precompute_field_params(x, y, z):
50     # Plaque du bas
51     if (x,y,z) in DJ:
52         return (DJ[(x,y,z)],DF[(x,y,z)])
53
54     else :
55         dx = X_j - x
56         dy = Y_j - y
57         dz = Z_j - z
58         dj = np.sqrt(dx**2 + dy**2 + dz**2)
59         df = D_F_vectorized(X_j, Y_j, Z_j, x, y, z)
60         DJ[(x,y,z)]=dj
61         DF[(x,y,z)]=df
62
63     return (dj, df)
64
65 # === Calcul du champ total ===
66
67 def p_tot(x, y, z, phi_vals, dj, df):
68     exp_term = np.exp(1j * (k * dj + phi_vals))
69     return np.sum(df / dj * exp_term)
70
71 # === Gradient numerique du champ ===
72

```

```

73 def grad_p(x, y, z, phi_vals, dj0, df0):
74     def shifted_p(dx, dy, dz):
75         dj, df = precompute_field_params(x + dx, y + dy, z + dz)
76         return p_tot(x + dx, y + dy, z + dz, phi_vals, dj, df)
77
78     dp_dx = (shifted_p(eps, 0, 0) - shifted_p(0, 0, 0)) / eps
79     dp_dy = (shifted_p(0, eps, 0) - shifted_p(0, 0, 0)) / eps
80     dp_dz = (shifted_p(0, 0, eps) - shifted_p(0, 0, 0)) / eps
81     return np.array([dp_dx, dp_dy, dp_dz])
82
83 # === Potentiel de Gorkov ===
84
85 def gorkov_potential(x, y, z, phi_vals, dj, df):
86     p_complex = p_tot(x, y, z, phi_vals, dj, df)
87     gradp = grad_p(x, y, z, phi_vals, dj, df)
88
89     omega = 2 * np.pi * 40000
90     k1 = K1(R_p, c_p, rho_p)
91     k2 = K2(R_p, omega, rho_p)
92
93     abs_p2 = abs(p_complex)**2
94     abs_gradp2 = np.sum(np.abs(gradp)**2)
95     return k1 * abs_p2 - k2 * abs_gradp2
96
97 # === Laplacien du potentiel ===
98
99 def lap_U(x, y, z, phi_vals, dj0, df0):
100    def U_shift(dx, dy, dz):
101        dj, df = precompute_field_params(x + dx, y + dy, z + dz)
102        return gorkov_potential(x + dx, y + dy, z + dz, phi_vals, dj, df)
103
104    dU_dxx = (U_shift(eps, 0, 0) - 2*U_shift(0, 0, 0) + U_shift(-eps, 0, 0)) / eps**2
105    dU_dyy = (U_shift(0, eps, 0) - 2*U_shift(0, 0, 0) + U_shift(0, -eps, 0)) / eps**2
106    dU_dzz = (U_shift(0, 0, eps) - 2*U_shift(0, 0, 0) + U_shift(0, 0, -eps)) / eps**2
107    return dU_dxx + dU_dyy + dU_dzz
108
109 # === Fonction objectif ===
110
111 def F_opt(xf, yf, zf, phi_vals, dj, df):
112     ptot = p_tot(xf, yf, zf, phi_vals, dj, df)
113     lapU = lap_U(xf, yf, zf, phi_vals, dj, df)
114     return abs(ptot) - lapU
115
116 # === Optimiseur ===
117
118 def optimizer2(xf, yf, zf):
119     dj, df = precompute_field_params(xf, yf, zf)
120     def objective(phi_vals):
121         phi_mod = np.mod(phi_vals, 2*np.pi)
122         return F_opt(xf, yf, zf, phi_mod, dj, df)
123
124     phi0 = np.zeros(n**2)
125     result = minimize(objective, phi0, method='L-BFGS-B')
126     phases_mod = np.mod(result.x, 2*np.pi)
127     print("Phases optimales trouvées (mod 2π) :")
128     print(", ".join(f"{a:.4f}" for a in phases_mod))
129     return phases_mod
130
131 # === Point de focalisation ===
132 X_foc, Y_foc, Z_foc = 0, 0, 0.03

```

```

133
134 # === Lancement ===
135 optimizer2(X_foc, Y_foc, Z_foc)

```

### 7.6.2 Méthode vortex

Voici le code phyton implémentant la méthode vortex pour le calcul des phases optimale pour un piège dont on choisie la position, avec une seule plaque :

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.special import spherical_jn, lpmv
4
5
6 # -----
7 # Parametres physiques
8 #
9 f = 40000          # frequence (Hz)
10 c0 = 343           # vitesse du son dans l'air (m/s)
11 = 2 * np.pi * f
12 k = / c0           # nombre d'onde
13
14 A = 1.0            # amplitude du vortex
15 l = 6               # ordre du vortex (radial)
16 m = 1               # moment angulaire (nombre topologique)
17 t = 0               # temps instantane pour la visualisation
18
19
20 # -----
21 # Espace de simulation
22 #
23 N = 8 # nombre de transducteur par axe (8x8)
24 a=0.01
25
26 # bornes symetriques : +/- 5 cm lateralement, 8 cm en hauteur
27 Lx = 0.05
28 Ly = 0.05
29 DzT=0.08
30
31 x = np.linspace(-Lx, Lx, N)
32 y = np.linspace(-Ly, Ly, N)
33
34 X_T, Y_T = np.meshgrid(x, y)
35 Z_T=np.array([[0]*N]*N)
36
37 # on passe tout une dimension, dans le ref de la plaque :
38 X_T=X_T.flatten()
39 Y_T=Y_T.flatten()
40 Z_T=Z_T.flatten()
41
42 # ---- Constantes physiques ----
43 k = 2 * np.pi * 40000 / 343
44 R_p = 0.0005
45 c_p = 2400
46 rho_p = 28.59
47 c_0 = 343
48 rho_0 = 1.204
49 R_t = 0.005 # rayon transducteur
50
51 eps = 1e-4 # pas de derivation
52
53 def compute_vortex_phases(xf, yf, zf):

```

```

54
55     ## dans le ref du vortex :
56
57     Z_T_v=Z_T-zf
58     X_T_v=X_T-xf
59     Y_T_v=Y_T-yf
60
61     # -----
62     # Passage en coordonnees spheriques
63     #
64     r = np.sqrt(X_T_v**2 + Y_T_v**2 + Z_T_v**2)
65     = np.arccos(Z_T_v/ (r + 1e-12))  # angle polaire
66     = np.arctan2(Y_T_v, X_T_v)           # angle azimutal
67
68     # -----
69     # Calcul du champ acoustique vortex(r, , )
70     #
71     jl = spherical_jn(l, k * r)
72     Plm = lpmv(m, l, np.cos( ))
73
74     vortex = A * jl * Plm * np.exp(1j * (m * - * t))
75
76     # -----
77     # Amplitude et phase
78     #
79     ampl = np.abs(vortex)
80     phase = np.angle(vortex)
81     phase=np.mod(phase,2*np.pi)
82
83     # normalisation amplitude (comme tu fais deja)
84     ampl = ampl / np.max(ampl)
85
86     return ampl, phase
87
88 # === Point de focalisation ===
89 X_foc, Y_foc, Z_foc = 0, 0, 0.03
90
91 # === Lancement ===
92 compute_vortex_phases(X_foc, Y_foc, Z_foc)

```