Driving a stepper motor with Arduino

Basile Radisson, Yi Man, Eva Kanso
Department of Aerospace and Mechanical Engineering,
University of Southern California, Los Angeles, CA 90089-1191, USA
(Dated: August 6, 2021)

In this tutorial we provide the basic information needed to drive a stepper motor in a well controlled manner. We describe the components and the code necessary to control the stepper. The control part is based on the Arduino platform which is open source, well documented and provides an excellent hardware-software solution for electronic project.

I. INTRODUCTION

In the following we provide a short tutorial on how to drive a stepper motor in a well controlled manner (i.e controlling the speed of the stepper according to a given mathematical function) using Arduino. One of the main interest of using Arduino for this kind of project is that it provides a all-in-one hardware-software solution. As an open-source project, Arduino is backed-up by a very active community that provides a lot of helpful resources online. Moreover, all the software part is free and the Arduino compatible boards can be usually found at a very affordable price which makes Arduino a very low-cost solution compared to other proprietary ones.

II. HARDWARE AND COMPONENTS

The hardware required to realize this project is composed of four main components, and can be purchased for about one hundred dollars. It is composed of the stepper itself, a microstep driver, a power supply and a Arduino board. In this section we provide information and resources on the role of each of these components.

A. Stepper motor

The first component of the setup is the stepper motor itself. As suggested by the name, a stepper motor is a motor that rotates by precise increments called ”steps”. This makes stepper motors very desirable for application where accurate positioning is needed. They are used in 3D printers, CNC machines or in the autofocus in digital cameras for example. Stepper motors are controlled by applying pulses of DC electricity to their internal coils. Each pulses advances the motor by one step. The number of steps per full rotation (or equivalently the angle traveled at each steps) is a characteristic of the motor. The more steps per revolution the more precise the control. The most standard steppers on the market have 200 steps/revolution (1.8° per step). For more precision the number of steps per revolution can be increased using a procedure called microstepping. It allows to move the motor in fraction of the actual step by adjusting the current in the different coils of the stepper. Precise microstepping can be achieved easily using an industrial microstep driver (see next section).

The choice of the stepper motor depends on your application and a lot of helpful resources can be found online to help you select the right stepper for your project (e.g see [1]). One of the main characteristics you want to look at is the torque-speed curve in order to determine whether or not the motor is able to provide enough torque at the desired speed for your project.

In this project, we are going to use a NEMA 23 bi-polar stepper motor with a standard 200steps/rev.

B. Microstep driver

A microstep driver is an electronic device that precisely divide the current between the motor phases in order to position the rotor at smaller increments between full steps. Increasing the number of step per rotation increases the accuracy of the positionning and reduces vibrations. However, as more steps are needed to achieve a full revolution of the shaft, it decreases the maximal speed you can reach for a given setup.

Industrial microstepper drivers usually come with switches that allow to easily change the number of step per revolution. There are a large number of microstep drivers on the market. Here again you can find helpful resources online to help you choosing the right one for your project [1]. Generally speaking, the microstep driver must be chosen based on the characteristics of your stepper motor.

In this project, we are going to use a driver Wantai DQ542MA that can be adjusted from 400 steps/rev to 25,600 steps/rev.

C. Power supply

The stepper driver and the motor must be supplied with DC current with an appropriate voltage. For this project we are going to use a standard 24V DC power supply.

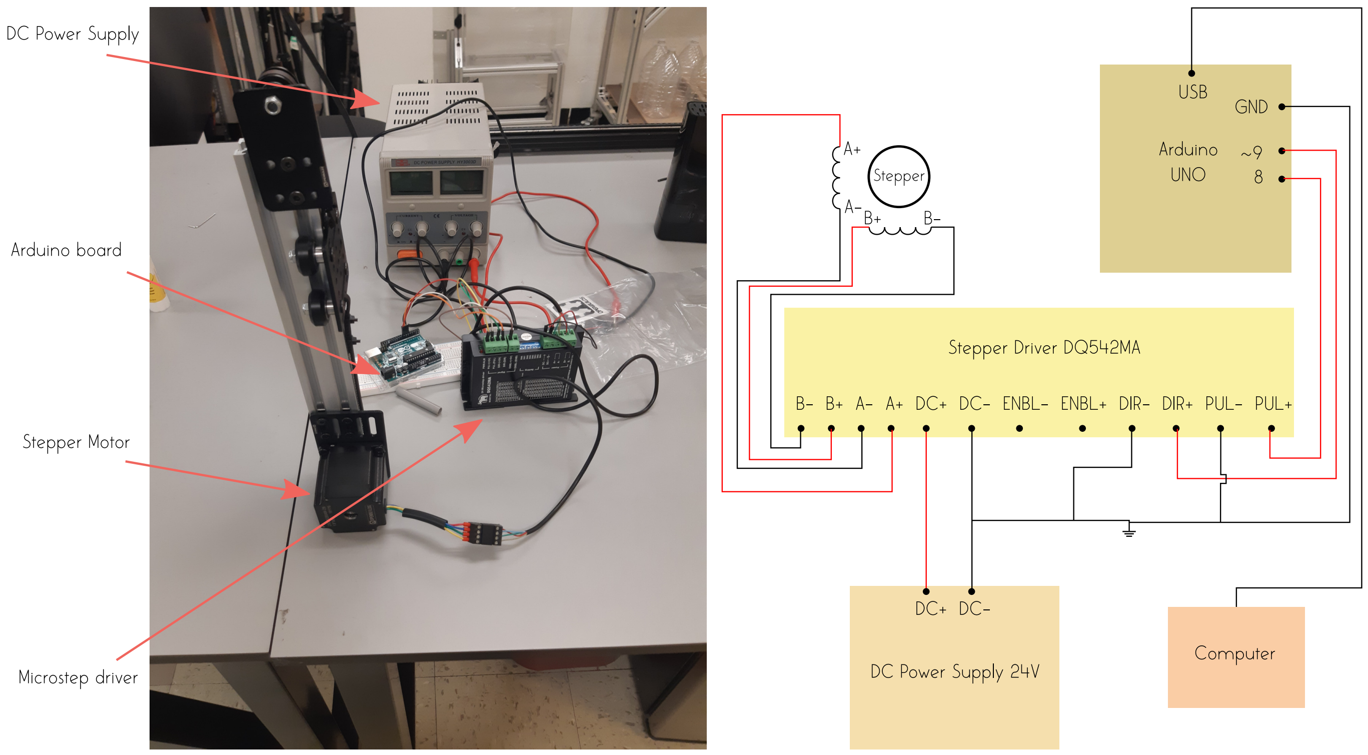

Figure 1. **Required hardware components for this project and wiring diagram**

D. Arduino board

The Arduino board is the main component of your setup. It is composed of a microcontroller and pins that can be used to acquire/send signal from/to external devices.

Theoretically, any standard Arduino board can be used to control a stepper motor, including the basic and most popular Arduino UNO board. However, the latter is very limited in term of memory which can become a problem when you want to drive a stepper at very high-speed, as we shall see.

In the following we will use both a Arduino UNO and a Arduino DUE board.

III. SOFTWARE AND CODE

The board must be told what to do by sending information to the microcontroller. This is done by using Arduino programming language that is send to the board through the Arduino IDE.

The Arduino programming language was specifically designed to communicate information to microcontrollers. The language itself is very close to the C programming language and a lot of the C libraries are available with Arduino. Arduino also includes a large number of additional libraries developed by the Arduino community for different applications. All these libraries can be easily installed using the Arduino IDE.

The Arduino IDE provides an all-in-one environment to write a program (Arduino programs are called ”sketches”) compile it and upload it to the board. It can be installed on any platform (Windows, Mac or Linux) following the instructions provided on the Arduino website [2].

IV. DRIVING A STEPPER MOTOR : BASICS

As mentioned earlier a stepper motor is driven by sending pulses of DC current that makes the motor to rotate by the angle corresponding to one step (or one fraction of a step if using microstepping).

A. Pulse and direction

There are two different kind of pulses that we want to send to the stepper motor through the driver. One is the directional pulse (DIR) that tells the motor in which direction we want to rotate and the other is the actual pulse (PUL) that tells the motor to rotate by one step. Therefore, we are going to use two pins as output on the Arduino board to send these two types of information to the motor. The wiring of the all setup is described on the wiring diagram on Fig. 1. Everything passes through the stepper driver. The two phases (here we are using a bipolar stepper motor) of the motor are wired to the corresponding pins on the driver. The DC power supply is wired to the two corresponding pins (DC+ and DC-) on the driver (no surprise here). Then comes the interesting part where the driver is wired to the board. We are using two digital pins of the Arduino UNO board here: pins 8 and 9. These two pins were chosen for speed consideration (see below) but technically any of the other digital pins can be used instead. Pin 8 is wired to the PUL+ pin on the driver and will be used to send the signal that tells the motor to rotate one step. Pin 9 is wired to the DIR+ pin on the driver and will be used to send the directional pulse. Finally, DIR- and PUL- on the driver and the GND pin on the Arduino board are all wired to the ground and the USB port on the Arduino board is connected to the computer. The latter both powers the board and allows to send information to the board.

B. Control of the time interval between two pulses

Now that everything is wired properly we can turn to the initial goal of this tutorial which is driving the motor in a well controlled manner. More precisely, we want to control the angular position θ of the motor so that the evolution in time of the position $\theta(t)$ follows a prescribed mathematical function. In the following we'll describe the procedure to drive the motor into a sinusoidal motion but the exact same procedure applies for any invertible function on a particular domain. As an example, here we want the position of the motor to be described by :

$$\theta(t) = -A\cos(\omega t) + A, \tag{1}$$

where A is the amplitude of the displacement and ω is the pulsation associated with the sinusoidal motion.
However, as mentioned earlier the only kind of action we can ask the motor to perform is : "rotate by one step in a given direction (i.e clockwise or counterclockwise)". This means that θ can take only discrete values that are multiple of the step angle $\Delta\theta$ such that

$$\theta_n = n\Delta\theta = n\frac{2\pi}{N_{ppr}} = -A\cos(\omega t_n) + A, \quad \{n \in \mathbb{N}, 0 \leq n \leq N\}, \tag{2}$$

where θ_n is the angular position at step n, N_{ppr} is the number of step per full revolution (determined by the setting of the microstep driver), $N = 2A/\Delta\theta$ is the number of step we want the motor to move on half a period and t_n is the time at which the stepper must be rotated from the angular position θ_{n-1} to the angular position θ_n. The latter is the quantity of interest as it determines the time at which we want to tell the motor to rotate by one step. It is simply obtained by inverting (2) such that :

$$t_n = \frac{1}{\omega}\arccos\left(1 - \frac{\theta_n}{A}\right), \quad \{n \in \mathbb{N}, 0 \leq n \leq N\}, \tag{3}$$

As we shall see, with Arduino it is easy to set a waiting time between two successive operations and it is therefore more convenient to compute the waiting time between two steps which is obtained using backward finite differences :

$$\Delta t_n = t_n - t_{n-1}, \quad \{n \in \mathbb{N}, 1 \leq n \leq N\}, \tag{4}$$

C. Basic Arduino implementation

Now that we have obtained the time to wait between two successive steps for the motor to achieve the desired sinusoidal motion, we can implement that in the Arduino IDE. The implementation is very simple, we simply need to define the pins we are using as the directional pin and the rotation pin on the Arduino board and then send the appropriate signal to these pins when we want the motor to rotate by one step. The code is given below :

```
#include <math.h> //import library math.h from C in order to use trigonometric function
#define stepPin 8 //we define stepPin as the pin 8
#define dirPin 9  //we define dirPin as the pin 9
```

```

float T=500.0; //period of the forcing in sec
float fForcing=1/T; //frequency of the forcing
float omega=2.*PI*fForcing; //pulsation of the forcing
unsigned int Nppr=800; //number of pulses per full rotation of the motor (this is set using the switches on the driver)
float nbRot=5.; //number of full rotations of the motor before reversing direction of rotation
unsigned int N=round(nbRot*Nppr/2); //amplitude of the oscillations (in term of motor steps)
int nb_oscillations=1;//number of periods
float tNext; //time at the next step
float tNow=0; //time now
long deltaT;

//the setup() function in Arduino is the function where we want to put the commands that are executed only one time (see Arduino documentation for more details)

void setup()
{
    Serial.begin(9600);
    pinMode (stepPin, OUTPUT); //set the pulse pin as output
    pinMode (dirPin, OUTPUT); //set the direction pin as output
    delay(150); //we wait 150ms before starting the loop


}

//the commands placed in the loop function in Arduino are repeated indefinitely until something breaks the loop or the board is reset.
void loop()
{
    for(int i=0; i<nb_oscillations; i++)
    {
        tNow=0;
        digitalWrite(dirPin, LOW); //set the Dir pin to low which corresponds to the counterclockwise direction
        for(unsigned long k=1; k<2*N+1; k++)
        {
            tNext=(1./omega)*acos(1.-float(k)/N)*pow(10,6);//compute the time at which the motor should rotate to the next position in microseconds
            deltaT=int(round(tNext-tNow)); //compute the waiting time
            delayMicroseconds(deltaT-10); //we wait for the computed deltaT -10microseconds
            digitalWrite(stepPin, HIGH); //set the Step pin to high
            delayMicroseconds(10); //wait for 10microseconds
            digitalWrite(stepPin, LOW); //set the step pin back to low
            tNow=tNext;  //update the current time
        }
        //when we get out of this loop we must revert the direction of rotation and redo the same thing to come back to the initial position

        tNow=0;
        digitalWrite(dirPin, HIGH); //set the Dir pin to high which corresponds to the clockwise direction
        for(unsigned long k=1; k<2*N+1; k++)
        {
            tNext=(1./omega)*acos(1.-float(k)/N)*pow(10,6);//compute the time at which the motor should rotate to the next position in microseconds
            deltaT=int(round(tNext-tNow)); //compute the waiting time
            delayMicroseconds(deltaT-10); //we wait for the computed deltaT -10microseconds
            digitalWrite(stepPin, HIGH); //set the Step pin to high
            delayMicroseconds(10); //wait for 10microseconds
            digitalWrite(stepPin, LOW); //set the step pin back to low
            tNow=tNext;  //update the current time
        }
    }

    exit(0);
}

```

The code is pretty straightforward, the only thing that need to be discussed in detail is the core loop in the function

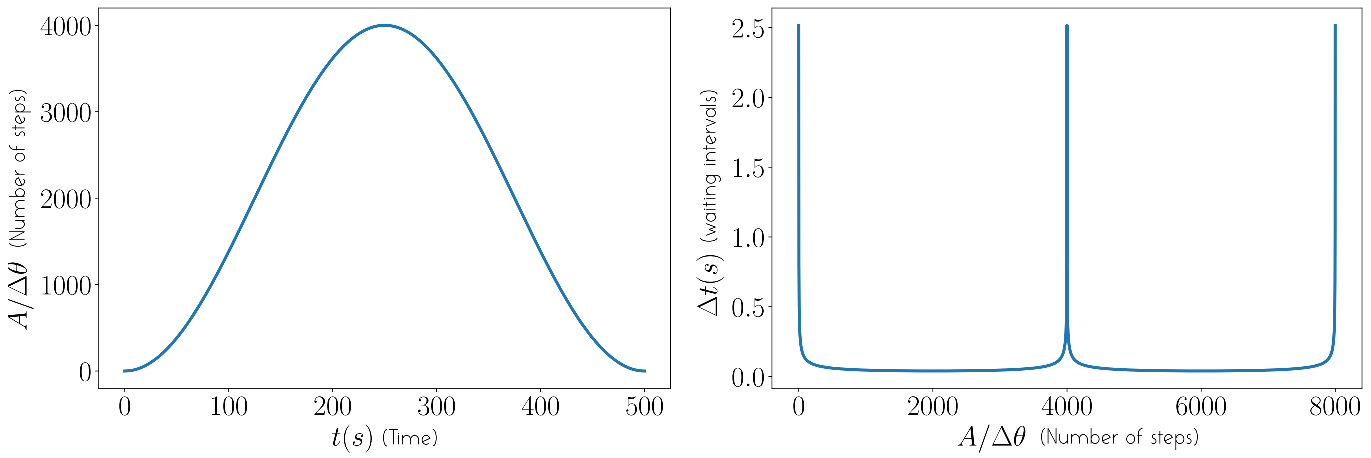

Figure 2. **Typical displacement and time intervals** Left: typical evolution of the displacement (in number of steps) of the shaft over one period of oscillations. Right : corresponding waiting time that must be applied between two successive steps for the motor to describe the motion associated with the left curve.

`loop()`. The `loop()` function is one of the main Arduino functions. All the commands in this function are repeated indefinitely until something breaks the program execution. Here, we don't want the motor to run indefinitely but for a given number of period (selected by the user by specifying the value of the variable `nb_oscillations`). Therefore, we use a first for loop that is executed `nb_oscillations` times before the command `exit(0)` breaks the execution. Inside this for loop, are the commands for the motor to perform one period of oscillations. This section is composed of two identical for loops - each one corresponding to half a period of oscillations - where only the direction of rotation is reversed. The two main Arduino functions we are using in this code are the functions `digitalWrite()` and `delayMicroseconds()`.

The function `digitalWrite()` allows to assign the value `HIGH` or `LOW` to a given pin. For a pin configured as `OUTPUT`, setting the pin to `HIGH` will set the current to 5V whereas setting the pin to `LOW` will set the current to the ground value (0V) [3]. As mentioned earlier, we have two pins to deal with here : the directional and the step pin. The directional pin set the direction of rotation (`LOW` (respectively `HIGH`) for counterclockwise (respectively clockwise) rotation of the motorshaft). Before the first loop, we therefore set the motor to rotate in the counterclockwise direction (l.34 in the code) then enter the first for loop where the motor is accelerated and decelerated in order to get half a period of the desired sinusoid. The direction of rotation is then reversed (l.48 in the code) and the same for loop is repeated to complete one period of oscillation. In the for loop itself, that's where we send the pulses to the step pin to perform rotation of the motor. A step rotation of the motor is achieved by sending a step signal to the step pin. This is done by setting the step pin to `HIGH` for a few microseconds and then turning it back to `LOW` (l.40 to l.42 in the code). The short delay (10 microseconds) applied between the two `digitalWrite()` is there to ensure that there is enough time for the current to reach the `HIGH` value before returning to `LOW`. Between two steps, we need to wait a time Δt as given by (4). This waiting time is computed on the go (l. 37 in the code) and directly applied using the function `delayMicroseconds()` (l.39 in the code). Note that we substract the 10 microseconds already applied between the two `digitalWrite()` to the computed delay `deltaT` in order to satisfy the total duration between two steps. With this code the motor will theoretically achieved the desired sinusoidal motion.

But here comes the problem: we have implicitly considered that all the other commands in the code except `delayMicroseconds()` were executed instantaneously so that we don't need to consider their execution time. Therefore, the code above remains valid only if the smallest Δt value along a period of oscillation remains large compared to the typical time of execution of all the other commands in the for loop. This assumption will obviously fell off for application where the motor must be driven at very high speed (i.e a large number of steps per unit of time).

In the example above, the desired sinusoid has an amplitude of 2000 motor steps (see Fig. 2 left panel) and a period T=500s. Which means the motor has to be rotated by 8000 steps in 500s. This corresponds to an average time of $\Delta t_{avg} = 6.25 \times 10^{-2}s$. But as with a sinusoidal motion, the motor starts from a zero speed then accelerate and decelerate to zero over one half of a period, the critical Δt_{crit} is actually shorter. As an example the evolution of Δt over one period of oscillations (computed from (4)) is plotted on Fig. 2 right panel. The shortest Δt value for this set of parameters is $\Delta t_{crit} \approx 3.98 \times 10^{-2}s$. This time must be large compared to the execution time of the commands in the for loop for the code presented above to achieve the desired sinusoidal motion.

D. Measuring execution time with Arduino

To get an idea of the execution time of the different commands in the for loop, we need to use some functions to measure this time. There are different functions in Arduino that allow to measure execution time. The most common are the functions `millis()` and `micros()`. These two functions work the same way and return the current time but with a different resolution (see [4] for details). Here, we are going to use the `micros()` function with the following script :

```
#include <math.h> //import library math.h from C in order to use trigonometric function
#define stepPin 8 //we define stepPin as the pin 8
#define dirPin 9  //we define dirPin as the pin 9

float T=500.0; //period of the forcing in sec
float fForcing=1/T; //frequency of the forcing
float omega=2.*PI*fForcing; //pulsation of the forcing
unsigned int Nppr=800; //number of pulses per full rotation of the motor (this is set using the
    switches on the driver)
float nbRot=5.; //number of full rotations of the motor before reversing direction of rotation
unsigned int N=round(nbRot*Nppr/2); //amplitude of the oscillations (in term of motor steps)
int nb_oscillations=1;//number of periods
float tNext; //time at the next step
float tNow=0; //time now
long deltaT; //time to wait

void setup()
{
    Serial.begin(9600); // opens serial port, sets data rate to 9600 bps
    pinMode(stepPin, OUTPUT); // set the stepPin as OUTPUT
    pinMode(dirPin, OUTPUT);  // set the dirPin as OUTPUT

    unsigned long timeBegin = micros(); // store the current time before execution of the loop

    for(unsigned long k=1; k<1000; k++) //execute the for loop 1000 times
    {
    tNext=(1./omega)*acos(1.-float(k)/N)*pow(10,6);//compute the time at which the motor should
    rotate to the next position in microseconds
    deltaT=int(round(tNext-tNow)); //compute the waiting time
    digitalWrite(stepPin, HIGH); //set the Step pin to high
    digitalWrite(stepPin, LOW); //set the step pin back to low
    tNow=tNext;  //update the current time
    }

    unsigned long timeEnd = micros();  // store the current time after execution of the loop
    unsigned long duration = timeEnd - timeBegin;
    double loopDuration = (double)duration / 1000; //compute the duration per loop

    // write to the serial monitor
    Serial.print("Execution time per loop: ");
    Serial.print(loopDuration);
    Serial.println("   s ");

}

void loop() {}
```

The preamble of the script is identical to the previous code, only the content of the `setup()` and `loop()` functions has changed. The `loop()` function is now empty and we have moved the core for loop of the previous code to the `setup()` function in order to measure its typical execution time. In the for loop, we got rid of the `delayMicroseconds()` commands as the purpose is to measure the execution time of the other commands. The time of execution is measured by measuring the current time using the function `micros()` before entering the loop, then looping 1000 times before measuring the current time again after the execution. The time per loop is then computed from the difference between these two times and send to the Serial monitor (To see the result, the user must go to tools/serial monitor in the Arduino IDE and execute the code). With a Arduino DUE board, we obtained a time of $121\mu s$ per loop (the time would be larger with a Arduino UNO board). This time is still quite small compared to the shortest Δt computed from (4) with this set of parameters and the motor will therefore execute the expected motion if triggered with this code. **But what if we now want the motor to execute oscillations of the same amplitude (2000 steps) but over a period of T = 1.0s instead of T = 500s ?** In this case the critical waiting time between two steps at the point where the speed of the motor is maximal would be $\Delta t_{crit} \approx 80\mu s$ which is shorter than the typical time we need to execute the other commands in the for loop. Which means that even if we remove the waiting time introduced with the function `delayMicroseconds()`, the execution time will be too slow for the motor to achieve the desired motion. In the next section, we show how this basic script can be optimized to achieve the desired motion.

V. ARDUINO SCRIPT OPTIMISATION : INCREASING SPEED

There are two main things that can be optimized in the core for loop of the previous program : (i) the waiting time Δt is computed on the go at each step. This is a slow operation and some precious execution time could be saved by computing the full series of waiting time Δt over one period beforehand and storing them in an array, as we shall see. (ii) the function digitalWrite() is a high level and convenient method to manipulate the pin values on a Arduino board but it is slow. Faster pin manipulation can be achieved through lower level manipulation of the pins.

A. Computing the waiting time values beforehand : a memory problem

Computing the value of Δt is by far the most expensive operation in the for loop. Using the script we used to get the typical execution time of the for loop but now removing the two `digitalWrite()` lines leads to a time of execution of $116.81\mu s$. The execution time of the loop is therefore largely dominated by the execution time of this operation. Indeed, the `acos()` function is an expensive computation that requires a large number of operations. An obvious way to optimize the code is to get rid of these operations by computing the Δt values beforehand and storing them in an array. However, we now have to face a memory problem. Standard Arduino boards come with a very low amount of RAM memory compared to even the most basic recent computer. For example the Arduino UNO Rev 3 has 2kB of RAM memory whereas the most basic Raspberry Pi 4 has 2GB instead. This means that memory optimization must be considered much more carefully when you're coding with Arduino than with normal coding on a standard computer. In our example above, a full period of oscillation is composed of 8000 steps that can be decomposed in 4 identical sections of 2000 different values of Δt. This means we have at least an array of 2000 integers containing the Δt values in microseconds which represents $2000 \times 2 = 4$kB of memory (see appendix for a list of most standard variables available with Arduino). This is already two times larger than the available RAM on a Arduino UNO (and this is without counting the other variables in the program). This problem can be solved by reducing the number of steps in order to have less Δt values to store. This can be achieved by changing the microstepping setting on the driver in order to achieve the same physical amplitude (same rotation angle) with less steps, but this obviously come with a loss in precision and can increase the vibrations (see first section of this tutorial). Another option is to upgrade the board to another one with more memory. For example the Arduino DUE has 96kB of RAM instead of 2kB for the UNO. The latter is the solution adopted for this tutorial. The wiring of this board to the other components is exactly the same as the one presented on the wiring diagram on Fig. 1 with the Arduino UNO board except that we use pins 25 and 26 of the Arduino DUE instead of 8 and 9 on the Arduino UNO (but as mentioned earlier any of the digital pins available can be used instead).

The optimized version of the code (but more expensive memory wise speaking) is now

```
#include <math.h>
#define stepPin 25 //declaration of the ste pin
#define dirPin 26 //declaration of the directional pin

float T=10.; //period of the forcing in sec
float fForcing=1/T;
float omega=2.*PI*fForcing;
unsigned int Nppr=800; //number of pulse per full rotation of the motor (this is set using the
    switches on the driver)
float nbRot=5.; //number of full rotations of the motor before reversing direction of rotation
unsigned int N=round(nbRot*Nppr/2); //amplitude of the oscillations (in term of motor steps)
int nb_oscillations=1;//number of period during the all run
float tNext; //time at the next step
float tNow=0; //time now
int *deltaT=(int*)malloc(2*N*sizeof(int)); //allocate memory for an array in order to store the
    waiting times


void setup()
{
    Serial.begin(9600);
    pinMode (stepPin, OUTPUT); //set the pulse pin as output
    pinMode (dirPin, OUTPUT); //set the direction pin as output

//initialisation of the new array
    for(unsigned long k=1; k<2*N+1; k++)
    {
        tNext=(1./omega)*acos(1.-float(k)/N)*pow(10,6);
        deltaT[k-1]=int(round(tNext-tNow));
        tNow=tNext;
    }

    delay(1500);
}


void loop()
{
    for(int i=0; i<nb_oscillations; i++)
    {
        digitalWrite(dirPin, LOW); //set the direction to counterclockwise
        for(unsigned long k=1; k<2*N+1; k++)
        {
            delayMicroseconds(deltaT[k-1]-10); // we apply the delay by simply reading the
    corresponding value in the array
            digitalWrite(stepPin, HIGH); //send a pulse to the step pin
            delayMicroseconds(10); //wait 10 microseconds to ensure the current reach its high
    value
            digitalWrite(stepPin, LOW); //the pin returns to its ground value
       }

        digitalWrite(dirPin, HIGH); //revert the direction of rotation
        for(unsigned long k=1; k<2*N+1; k++)
        {
            delayMicroseconds(deltaT[k-1]-10); // we apply the delay by simply reading the
    corresponding value in the array
            digitalWrite(stepPin, HIGH); //send a pulse to the step pin
            delayMicroseconds(10); //wait 10 microseconds to ensure the current reach its high
    value
            digitalWrite(stepPin, LOW); //the pin returns to its ground value
        }
    }
    free(deltaT); //free up the allocated memory
    exit(0); //breaks the execution
}
```

As can be seen, the only thing that has changed is that now we compute the Δt beforehand and store them in an array so that in the core loop of the script we just have to access the corresponding value in the array at each step. This is way faster than computing these waiting times on the go as done in the previous script. It is difficult to measure precisely the time taken to read the Δt value in the array but we can get an order of magnitude with the following script :

```
    #include <math.h> //import library math.h from C in order to use trigonometric function
    #define stepPin 25 //we define stepPin as the pin 25
    #define dirPin 26  //we define dirPin as the pin 26

    float T=1.; //period of the forcing in sec
    float fForcing=1/T; //frequency of the forcing
    float omega=2.*PI*fForcing; //pulsation of the forcing
    unsigned int Nppr=800; //number of pulses per full rotation of the motor (this is set using the
     switches on the driver)
    float nbRot=5.; //number of full rotations of the motor before reversing direction of rotation
    unsigned int N=round(nbRot*Nppr/2); //amplitude of the oscillations (in term of motor steps)
    int nb_oscillations=1;//number of periods
    float tNext; //time at the next step
    float tNow=0; //time now
    int *deltaT=(int*)malloc(2*N*sizeof(int)); //array to store the waiting time

    void setup()
    {
    Serial.begin(9600);
    pinMode(stepPin, OUTPUT);
    pinMode(stepPin, OUTPUT);

    //initialisation of the new array
    for(unsigned long k=1; k<2*N+1; k++)
    {
        tNext=(1./omega)*acos(1.-float(k)/N)*pow(10,6);
        deltaT[k-1]=int(round(tNext-tNow));
        tNow=tNext;
    }

    unsigned long timeBegin = micros();
    for(unsigned long k=1; k<2*N+1; k++)
    {
        delayMicroseconds(deltaT[k-1]);
    }
    unsigned long timeEnd = micros();
    unsigned long duration = timeEnd - timeBegin;
    double loopDuration = (double)duration / (2*N);
    free(deltaT);

    Serial.print("Execution time per loop: ");
    Serial.print(loopDuration);
    Serial.println("   s ");
}
void loop() {}
```

Here, we generate the array with the set of Δt corresponding to half a period of oscillations and we then loop over this array and apply the appropriate delay by reading the waiting time in the array and passing it as argument to the function `delayMicroseconds()`. In order to get an idea of the time taken by the memory reading process we measure the average time per loop over half a period and compare it to the average $\Delta t_{avg} = T/8000$ (where 8000 is the number of steps per period). We repeat the process for different values of the period T and compare the Δt_{avg} expected to the actual Δt_m measured with the above script, the results are reported in the following array :

$T(s)$	$\Delta t_{avg} = T/8000(\mu s)$	Δt_m
1.0	125	125.41
0.1	12.5	12.57
0.01	1.25	1.57

As can be seen, with a period $T = 1.0s$, the average waiting time measured is almost identical to the expected one which means the time needed to access the Δt value in the memory is infinitesimal compared to the typical Δt value. The same remark can be made for the second case with a period $T = 0.1s$ where a small delay is observed but nothing that could severely affect the expected motion of the motor ultimately. However, for the last case, when the expected waiting time Δt_{avg} becomes of the order of a microsecond, we note a significant delay in the measured waiting time. However, this delay is not due to the time needed to access the Δt value in the memory but to the function `delayMicroseconds()` itself. Indeed, this function is given to be accurate for typical delay of $3\mu s$ and above. Therefore, the time to obtain the Δt values is no longer a problem. By computing it beforehand and storing it in the memory instead of computing it on the go, we have improved the execution time of this operation from $\approx 100\mu s$ to something smaller than a microsecond (actually memory readings are theoretically of the order of a few typical clocktime which is about two order of magnitude smaller than a microsecond for a Arduino Due).

Now that we have optimized the most expensive operation in the core loop that drives the motor rotations, we can turn to the optimization of the other operations.

B. Manipulating Arduino pins at a lower level : fast `digitalWrite()`

The other main operation that we had in the for loop, after computing and applying the appropriate delay, was sending a pulse of current to the step pin. This was done by using a `digitalWrite(stepPin, HIGH)` followed by a short delay and a `digitalWrite(stepPin, LOW)`. From our earlier execution time measurements, we know that the typical time taken by a `digitalWrite()` is about $2.2\mu s$ on our Arduino Due. This operation is even slower on a Arduino Uno : about $3.4\mu s$ for pins 8-9 and a bit slower on pins 2-3 according to our measurements. As we have seen in the previous section that we can achieve accurate delay as low as $3\mu s$ with the `delayMicroseconds()` function it would be nice to reduce the time needed to manipulate the pins in order to be able to reach effective Δt as low as a few microseconds long. This can be done by manipulating pins at a lower level.

The `digitalWrite()` is a high-level and convenient function but it is pretty slow. Indeed, under the hood, it performs several operations before actually changing the state of the pin [5]. Instead, we can directly manipulate pin's state using what is called direct port manipulation. This significantly improves the execution speed as we shall see.

On a Arduino board, the pins are distributed on several different ports. For example, the Arduino UNO board has 3 ports B, C and D with 8 pins per port. The pins can be directly manipulated by using the letter associated with the corresponding port and the number of the pin. The mapping of the different pins on a given board can be found easily online. For example, on a Arduino Uno the pins 0, 1, 2, 3, 4, 5, 6, 7 correspond to the pins 0, 1, 2, 3, 4, 5, 6, 7 of the port D, respectively. The pins 8, 9, 10, 11, 12, 13 correspond to the pins 0, 1, 2, 3, 4, 6 of the port B, respectively.. To manipulate the state of the pin using direct port manipulation, you just need to know the letter associated with the port and the number of the pin.

On a Arduino Uno, the equivalent of

```
#define stepPin 8
void setup()
{
    Serial.begin(9600);
    pinMode(stepPin, OUTPUT);

    for (int i = 0; i < 1000; ++i)
    {
        digitalWrite(stepPin, HIGH);
        digitalWrite(stepPin, LOW);
    }

}
void loop() {}
```

using direct port manipulation would be :

```
#define stepPin 8
void setup()
{
    Serial.begin(9600);
    pinMode(stepPin, OUTPUT);

    for (int i = 0; i < 1000; ++i)
    {
        PORTB = B00000001;
        PORTB = B00000000;
    }

}
void loop() {}
```

The syntax works as follow : we simultaneously attribute a value to all the pins of a given port. The port is designated by its letter. Here we write `PORTB=` to manipulate the pins on port B and it would have been `PORTC=` or `PORTD=` to manipulate the pins on port C or D respectively. On the right hand side we write `B` followed by eight value corresponding to the state of the eight pins of the port. Be careful here, `B` stands for Byte it is not the letter of the port. Then comes the value we want to attribute to the pins. The pins are organized in the reverse order : the last bit corresponds to pin 0 and the first one to pin 7. Here we are manipulating pin 8 which is the pin 0 on port B on a Arduino Uno board. If the corresponding bit is set to 1, this set the current on this pin to HIGH and conversely 0 corresponds to LOW. To set pin 8 to HIGH and then LOW we therefore first attribute the value `B00000001` and then `B00000000` to `PORTB=`. By doing so we also set all the other pins on this port to `0` (LOW). This can be a problem in case we didn't want to change the values of the other pins while manipulating pin 8. For more information on precise direct port manipulation and on how to circumvent this problem, see this excellent tutorial [5].

Using direct port manipulation, the time necessary to change the state of a pin on a Arduino Uno is typically reduced by one order of magnitude. On the example above the optimized version takes an average of $0.19\mu s$ per pin manipulation instead of $3.4\mu s$ when using the `digitalWrite()` function [5]. This is a drastic improvement.

On the Arduino DUE, the syntax to manipulate the pin directly is more complicated, but the execution time improvement compared to a standard `digitalWrite()` is even more important. On a Arduino DUE, there are 5 ports A, B, C, D, E and each of them is 32bits (instead of 8bits on the Arduino UNO). There are several registers on each port that allows to read and write it. Here, the two registers we are interested in are `PIO_SODR` and `PIO_CODR`. Writing 1's to `PIO_SODR` sets the corresponding pins to HIGH where as writing 1's to `PIO_CODR` sets the corresponding pins to LOW. To alternatively change a pin from HIGH to LOW, the syntax is as follow :

```
#define stepPin 25
void setup()
{
    Serial.begin(9600);
    pinMode(stepPin, OUTPUT);

    for (int i = 0; i < 1000; ++i)
    {
        PIOD -> PIO_SODR = 0b00000000000000000000000000000001;
        PIOD -> PIO_CODR = 0b00000000000000000000000000000001;
    }
}
void loop() {}
```

Here, we are manipulating pin 25 which is the pin 0 on port D (the complete mapping of the different pins can be found online). We choose the port we want to manipulate by starting the line with `PIOX ->` where `X` is the letter of the port (`D` in our case). We then write 1's to the corresponding registers `PIO_SODR` to turn pins HIGH and `PIO_CODR` to turn pin LOW. The syntax on the right hand side starts by `0b` followed by 32 bits corresponding to the 32 pins of the port organized in the reversed order (here we are manipulating pin 25 which is pin 0 on port D so we are writing ones on the very last bit of the port).

With the code above one pin manipulation takes $0.05\mu s$ compared to $2.24\mu s$ with a `digitalWrite()`. This is about fifty times faster !! With this kind of pin manipulation all the commands in the core for loop are instantaneous compared to the minimal waiting time Δt that can be applied with `delayMicroseconds()` ($3\mu s$) and we can therefore drive stepper motor for applications where Δt between steps as low as a few microseconds are needed (it is actually possible to achieve faster motion by getting rid of the `delayMicrosecond()` function and applying the delay Δt through lower level commands but we'll stick with that solution for this tutorial).

The final optimized code to drive our stepper motor is as follows :

```
#include <math.h>
#define stepPin 25
#define dirPin 26

float T=1.5; //period of the forcing in sec
float fForcing=1/T;
float omega=2.*PI*fForcing;
unsigned int Nppr=800; //number of pulses per full rotation of the motor (this is set using the
    switches on the driver)
float nbRot=5.; //number of full rotations of the motor before reversing direction of rotation
unsigned int N=round(nbRot*Nppr/2); //amplitude of the oscillations (in term of motor steps)
int nb_oscillations=10;//number of period during the all run
float tNext; //time at the next step
float tNow=0; //time now
int *deltaT=(int*)malloc(2*N*sizeof(int)); //array to store the waiting time


void setup()
{
  Serial.begin(9600);
  pinMode (stepPin, OUTPUT); //set the pulse pin as output
  pinMode (dirPin, OUTPUT); //set the direction pin as output

  //initialisation of the new array
  for(unsigned long k=1; k<2*N+1; k++)
  {
    tNext=(1./omega)*acos(1.-float(k)/N)*pow(10,6);
    deltaT[k-1]=int(round(tNext-tNow));
    tNow=tNext;
  }

  delay(150);
}


void loop()
{
  for(int i=0; i<nb_oscillations; i++)
  {

    PIOD -> PIO_CODR = 0b00000000000000000000000000000010;//set the Dir pin to low
    for(unsigned long k=1; k<2*N+1; k++)
    {
      delayMicroseconds(deltaT[k-1]-3); // we apply the delay by simply reading the corresponding
    value in the array
      PIOD -> PIO_SODR = 0b00000000000000000000000000000001; //set the step pin to high
      delayMicroseconds(3); //apply a short delay
      PIOD -> PIO_CODR = 0b00000000000000000000000000000001;//set the step pin to high
    }

    PIOD -> PIO_SODR = 0b00000000000000000000000000000010;//reverse the direction of rotation and
    repeat
    for(unsigned long k=1; k<2*N+1; k++)
    {
      delayMicroseconds(deltaT[k-1]-3);
      PIOD -> PIO_SODR = 0b00000000000000000000000000000001;
      delayMicroseconds(3);
      PIOD -> PIO_CODR = 0b00000000000000000000000000000001;
    }
  }
  free(deltaT); //free up the allocated memory
  exit(0); //breaks the execution
}
```

VI. CODE VALIDATION

Now that we have optimized our code, it's time to give it a try in real life in order to check the actual motion of the stepper motor. For this purpose, we are going to track the angular position of the shaft of the motor with a camera. A small piece of tape is attached around the shaft of the motor in order to make the tracking of the angular position easier. The motor is then attached to a bench in front of a high speed camera, and it is driven through oscillatory motion using the code and the setup presented above. The evolution of the angular position of the shaft in the course of time is recorded with the camera (50fps). The obtained movie is then analyzed with an in house Python code (based on the scikit image library [6]) in order to extract the angular position $\theta(t)$ of the shaft on each frame. On Fig. 3, the obtained curve for $\theta(t)$ (blue symbols) is compared to the sinusoid (black line) sent as input with the code that drives the motor. As can be seen the motion of the actual angular position of the shaft follows exactly the theoretical sinusoid. Here, we show the result obtained for a sinusoid with an amplitude of $N = 2000$ steps (i.e 8000 steps per period) and a period $T = 3.0s$ but we have tried different periods including some as low as $T = 0.8s$ for the same amplitude and obtained perfect agreement between the input function and the final output motion. In fact with our setup we reach the physical limits of the motor (i.e when the motor can no longer provide the required torque at the desired speed) before reaching the electronical limits but as explained earlier, the code presented above is technically able to drive a stepper with Δt values as low as a few μs (let's say $\approx 10\mu s$ to see large).

Appendix A: Common variables in Arduino

Here is a description of the most common variables available in Arduino (more information can be found in the references on the Arduino website)

- byte : unsigned integer from 0 to 255 (8 bits)
- int : integer from -32768 to 32767 (16 bits) [on fancy Arduino boards : can be coded on 32bits instead of 16 which extends the range of value]
- unsigned int : integer from 0 to 65535 (16 bits)
- long : integer from -2 147 483 648 to 2 147 483 647 (32 bits) [when doing maths with integers, at least one of the values must be forced as a long using the letter L e.g : 3232L]
- unsigned long : integer from 0 to 4 294 967 295 [when doing maths with integers, at least one of the values must be forced as a long using the letters UL e.g : 3232UL]

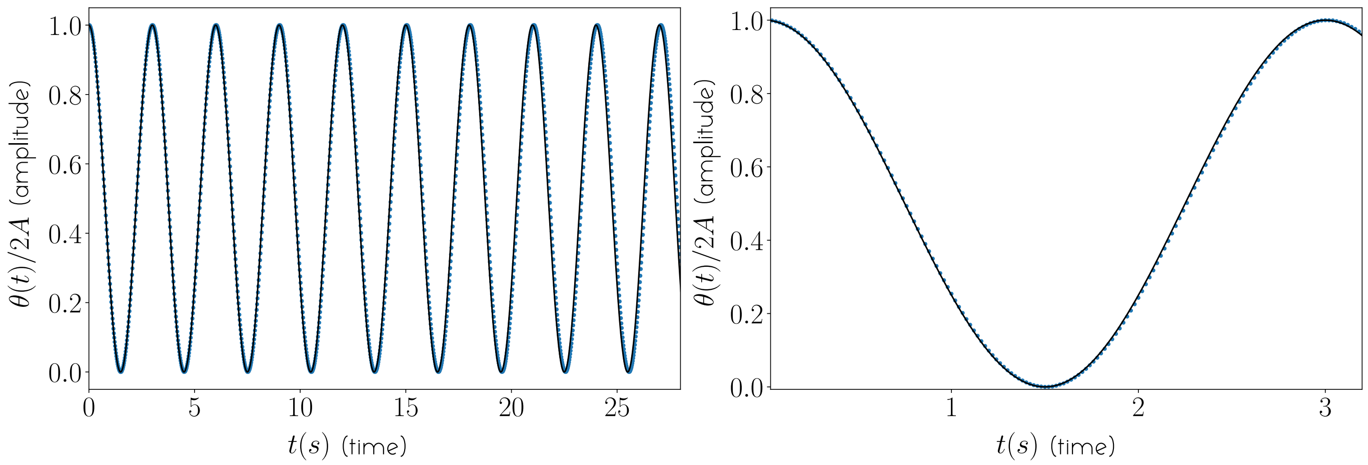

Figure 3. **Comparison between the input function sent to the motor and the actual motion of the shaft** Left : evolution of the non-dimensional angular position of the shaft over 9 periods. Right : same over 1 period only. The recorded position of the shaft (blue symbols) is almost identical to the input function (black line) sent to the motor.

- float and double : floating point number between -3.4028235E+38 and 3.4028235E+38 [float and double are exactly equivalent on standard Arduino boards like the Arduino Uno]

[1] "DroneBot Workshop," https://dronebotworkshop.com/ (2021), [Online; accessed 21-July-2021].
[2] "Arduino," https://www.arduino.cc/ (2021), [Online; accessed 21-July-2021].
[3] "Arduino digitalWrite()," https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/ (2021), [Online; accessed 21-July-2021].
[4] "Arduino millis() and micros()," https://roboticsbackend.com/arduino-millis-vs-micros/ (2021), [Online; accessed 21-July-2021].
[5] "Optimize digitalWrite()," https://roboticsbackend.com/arduino-fast-digitalwrite/ (2021), [Online; accessed 21-July-2021].
[6] "Scikit image library," https://scikit-image.org/ (2021), [Online; accessed 21-July-2021].