FlexyStepper:

This Arduino library is used to control one or more stepper motors. The motors are accelerated and decelerated as they travel to their destination. The library has been optimized for flexible control where speeds and positions can be changed while in motion.

Stepper Motor Driver Boards:

FlexyStepper requires that your stepper motor be connected to the Arduino using drive electronics having a "Step and Direction" interface. For a hookup guide, go to: https://github.com/Stan-Reifel/FlexyStepper

Examples of Stepper Motor Driver Boards are:

Pololu's DRV8825 Stepper Motor Driver Carrier: https://www.pololu.com/product/2133

Pololu's A4988 Stepper Motor Driver Carrier: https://www.pololu.com/product/2980

Sparkfun's Big Easy Driver: https://www.sparkfun.com/products/12859

GeckoDrive G203V industrial controller: https://www.geckodrive.com/g203v.html

For all driver boards, it is VERY important that you set the motor current. This is typically done by adjusting a potentiometer on the board. Read the driver board's documentation to learn how.

Selecting a power supply:

Stepper motors will spin with greatest torque at speed, when using a power supply voltage that is many times higher than the motor's voltage rating. Don't worry about this, the current setting on the driver board keeps the motor protected. I often use NEMA 17 motors with a voltage rating around 3V, then power my driver board with a 24V supply. Just make sure that your power supply voltage does not exceed the voltage rating of the driver board. 12V motors with a 12V supply will have much lower torque than my 3V/24V combo. (Note: NEMA 17 describes the size of the motor, 1.7 inches wide)

Smaller motors can typically spin faster than larger ones. The best way to evaluate a motor is by looking at its "torque curve". Most of the stepper motors sold by www.pololu.com have a data sheet on their website showing a torque curve (motors sold on Amazon usually do not).

Microsteps:

Most stepper motors have a 1.8 degree step angle, which results in 200 steps/revolution. A typical *Stepper Motor Driver Board* allows you to configure it for Microstepping. Microstepping gives you greater angular resolution and smoother rotates. If your board is setup for 2x microstepping, then you will need to move 400 steps for one rotation. Using 8x microstepping requires 1600 steps / revolution.

FlexyStepper vs SpeedyStepper:

The *FlexyStepper* library allows you to make changes to the target position, speed and rate of acceleration while the motor is turning. The only down side to these capabilities is that they reduce the number of steps/second that can be generated. This results in slower motor speeds. The *SpeedyStepper* library doesn't allows in motion changes, but has a faster step rate. See:
https://github.com/Stan-Reifel/SpeedyStepper

FlexyStepper can generate a maximum of about 7,000 steps per second using an Arduino Uno. Assuming a system driving only one motor at a time, in full step mode, with a 200 steps per rotation motor, the maximum speed is about 35 RPS or 2100 RPM (most stepper motor can not go this fast). Driving one motor in half step mode, a maximum speed of 17 RPS or 1050 RPM can be reached. In quarter step mode about 9 RPS or 525 RPM. Running multiple motors at the same time will reduce the maximum speed. For example running two motors will reduce the step rate by half or more.

Absolute vs relative moves:

The FlexyStepper functions that make a motor move come in two flavors: *Absolute* and *Relative*. Relative moves will use a coordinate system that is relative to the motor's current position. For example moving relative 200 steps, then another 200, then another 200, will turn 600 steps in total (and end up at an absolute position of 600).

Absolute moves use a coordinate system that is referenced to the original position of the motor when it is first turned on. First issuing an absolute move to position 200 will rotate forward one rotation. Then running the next absolute move to position 400 will turn just one more revolution in the same direction. Finally, an absolute move to position 0 will rotate backward two revolutions, back to the starting point.

Units of travel:

Frequently issuing move commands in units of "Steps" is not as intuitive as other units, such as *Millimeters* or *Rotations*. Many stepper motors are used in projects where the motions are linear instead of rotational. Examples are XY mechanisms such as plotters and 3D printers. In these cases issuing move commands in Millimeters is much more intuitive than Steps.

If you choose to work in units of Rotations, you will first need tell the library "how many steps makes one rotation". Do so using the function: *setStepsPerRevolution()*. Then move the motor with calls to *moveToPositionInRevolutions()* and *moveRelativeInRevolutions()*.

If you choose to work in units of Millimeters, you will need tell the library "how many steps makes one millimeter". Do so using the function: *setStepsPerMillimeter()*. Then move the motor with calls to *moveToPositionInMillimeters()* and *moveRelativeInMillimeters()*.

Blocking vs Non-blocking function calls:

The easiest way to tell a stepper motor to move is using a function like this: moveToPositionInSteps(125)

This function will rotate the motor until it gets to coordinate 125, but the function call doesn't return until the motion is complete. This is a *Blocking* function.

Blocking functions are a limitation if you want to do other things while the motor is running. Examples of "other things" might be to: 1) Run two motors at once. 2) Check if a button is press, then decelerate to a stop. 3) Turn on a solenoid when the motor moves past position 850. 4) Flash a strobe every 200 steps. 5) Run continuously as long as a temperature is below 125 degrees. To do these types of things *Non-blocking* functions are used. Here is an example:

```
//
// setup the motor so that it will rotate 2000 steps, note: this
// command does not start moving yet
//
stepper.setTargetPositionInSteps(2000);

//
// now execute the move, looping until the motor has finished
//
while(!stepper.motionComplete())
{
  stepper.processMovement();       // this call moves themotor

  //
  // check if motor has moved past position 400, if so turn On the LED
  //
  if (stepper.getCurrentPositionInSteps() == 400)
    digitalWrite(LED_PIN, HIGH);
}
```

A limitations to consider is the code you run in the *while* loop needs to execute VERY fast. Perhaps no longer than 0.05 milliseconds.

Homing:

Many stepper motor projects need to move to exact positions. This usually requires that the stepper motor library knows where the motor is before it makes its first move. To achieve this automatically, you will use a *Limit Switch* that is pressed by the mechanism when the motor moves all-the-way to one end of its travel. This procedure called *Homing*, is executed when power is first applied. A homing program moves the motor toward the limit switch until the switch is pressed, then resets the library so that the motor is now at position 0.

Home automatically is easily done using one of these functions:

- moveToHomeInSteps()
- moveToHomeInRevolutions()
- moveToHomeInMillimeters()

Usage examples:

```
Near the top of the program, add:
    include "FlexyStepper.h"

For each stepper, declare a global object outside of all functions as follows:
    FlexyStepper stepper1;
    FlexyStepper stepper2;

In Setup(), assign IO pins used for Step and Direction:
    stepper1.connectToPins(10, 11);
    stepper2.connectToPins(12, 14);
```

```
Move one motor in units of steps:
    //
    // set the speed in steps/second and acceleration in steps/second/second
    //
    stepper1.setSpeedInStepsPerSecond(100);
    stepper1.setAccelerationInStepsPerSecondPerSecond(100);

    //
    // move 200 steps in the backward direction
    //
    stepper1.moveRelativeInSteps(-200);

    //
    // move to an absolute position of 200 steps
    //
    stepper1.moveToPositionInSteps(200);
```

```
Move one motor in units of revolutions:
    //
    // set the number of steps per revolutions, 200 with no microstepping,
    // 800 with 4x microstepping
    //
    stepper1.setStepsPerRevolution(200);

    //
    // set the speed in rotations/second and acceleration in
    // rotations/second/second
    //
    stepper1.setSpeedInRevolutionsPerSecond(1);
    stepper1.setAccelerationInRevolutionsPerSecondPerSecond(1);

    //
    // move backward 1.5 revolutions
    //
    stepper1.moveRelativeInRevolutions(-1.5);

    //
    // move to an absolute position of 3.75 revolutions
    //
```

```
    stepper1.moveToPositionInRevolutions(3.75);
```

```
Move one motor in units of millimeters:
    //
    // set the number of steps per millimeter
    //
    stepper1.setStepsPerMillimeter(25);

    //
    // set the speed in millimeters/second and acceleration in
    // millimeters/second/second
    //
    stepper1.setSpeedInMillimetersPerSecond(20);
    stepper1.setAccelerationInMillimetersPerSecondPerSecond(20);

    //
    // move backward 15.5 millimeters
    //
    stepper1.moveRelativeInMillimeters(-15.5);

    //
    // move to an absolute position of 125 millimeters
    //
    stepper1.moveToPositionInMillimeters(125);
```

```
Move two motors in units of revolutions:
    //
    // set the number of steps per revolutions, 200 with no microstepping,
    // 800 with 4x microstepping
    //
    stepper1.setStepsPerRevolution(200);
    stepper2.setStepsPerRevolution(200);

    //
    // set the speed in rotations/second and acceleration in
    // rotations/second/second
    //
    stepper1.setSpeedInRevolutionsPerSecond(1);
    stepper1.setAccelerationInRevolutionsPerSecondPerSecond(1);
    stepper2.setSpeedInRevolutionsPerSecond(1);
    stepper2.setAccelerationInRevolutionsPerSecondPerSecond(1);

    //
    // setup motor 1 to move backward 1.5 revolutions, this step does not
    // actually move the motor
    //
    stepper1.setTargetPositionRelativeInSteps(-1.5);

    //
    // setup motor 2 to move forward 3.0 revolutions, this step does not
    // actually move the motor
```

```
    //
    stepper2.setTargetPositionRelativeInSteps(3.0);

    //
    // execute the moves
    //
    while((!stepper1.motionComplete()) || (!stepper2.motionComplete()))
    {
      stepper1.processMovement();
      stepper2.processMovement();
    }
```

The library of functions:

Setup functions:

```
//
// connect the stepper object to the IO pins
//  Enter:  stepPinNumber = IO pin number for the Step
//          directionPinNumber = IO pin number for the direction bit
//          enablePinNumber = IO pin number for the enable bit (LOW is enabled)
//            set to 0 if enable is not supported
//
void connectToPins(byte stepPinNumber, byte directionPinNumber)
```

Functions with units in millimeters:

```
//
// set the number of steps the motor has per millimeter
//
void setStepsPerMillimeter(float motorStepPerMillimeter)


//
// get the current position of the motor in millimeter, this functions is updated
// while the motor moves
//  Exit:  a signed motor position in millimeter returned
//
float getCurrentPositionInMillimeters()


//
// set current position of the motor in millimeter, this does not move the motor
//
void setCurrentPositionInMillimeters(float currentPositionInMillimeter)
```

```
//
// set the maximum speed, units in millimeters/second, this is the maximum speed
// reached while accelerating
// Note: this can only be called when the motor is stopped
//  Enter:  speedInMillimetersPerSecond = speed to accelerate up to, units in
//          millimeters/second
//
void setSpeedInMillimetersPerSecond(float speedInMillimetersPerSecond)


//
// set the rate of acceleration, units in millimeters/second/second
// Note: this can only be called when the motor is stopped
//  Enter:  accelerationInMillimetersPerSecondPerSecond = rate of acceleration,
//          units in millimeters/second/second
//
void setAccelerationInMillimetersPerSecondPerSecond(
                     float accelerationInMillimetersPerSecondPerSecond)


//
// home the motor by moving until the homing sensor is activated, then set the
// position to zero, with units in millimeters
//  Enter:  directionTowardHome = 1 to move in a positive direction, -1 to move in
//             a negative directions
//          speedInMillimetersPerSecond = speed to accelerate up to while moving
//             toward home, units in millimeters/second
//          maxDistanceToMoveInMillimeters = unsigned maximum distance to move
//             toward home before giving up
//          homeSwitchPin = pin number of the home switch, switch should be
//             configured to go low when at home
//  Exit:   true returned if successful, else false
//
bool moveToHomeInMillimeters(long directionTowardHome,
  float speedInMillimetersPerSecond, long maxDistanceToMoveInMillimeters,
  int homeLimitSwitchPin)


//
// move relative to the current position, units are in millimeters, this function
// does not return until the move is complete
//  Enter:  distanceToMoveInMillimeters = signed distance to move relative to the
//          current position in millimeters
//
void moveRelativeInMillimeters(float distanceToMoveInMillimeters)


//
// setup a move relative to the current position, units are in millimeters, no
// motion occurs until processMove() is called
//  Enter:  distanceToMoveInMillimeters = signed distance to move relative to the
//          current position in millimeters
```

```
//
void setTargetPositionRelativeInMillimeters(float distanceToMoveInMillimeters)



//
// move to the given absolute position, units are in millimeters, this function
// does not return until the move is complete
//  Enter:  absolutePositionToMoveToInMillimeters = signed absolute position to
//            move toin units of millimeters
//
void moveToPositionInMillimeters(float absolutePositionToMoveToInMillimeters)


//
// setup a move, units are in millimeters, no motion occurs until processMove()
// is called
//  Enter:  absolutePositionToMoveToInMillimeters = signed absolute position to
//          move to in units of millimeters
//
void setTargetPositionInMillimeters(float absolutePositionToMoveToInMillimeters)


//
// Get the current velocity of the motor in millimeters/second.  This functions is
// updated while it accelerates up and down in speed.  This is not the desired
// speed, but the speed the motor should be moving at the time the function is
// called.  This is a signed value and is negative when motor is moving backwards.
// Note: This speed will be incorrect if the desired velocity is set faster than
// this library can generate steps, or if the load on the motor is too great for
// the amount of torque that it can generate.
//  Exit:  velocity speed in millimeters per second returned, signed
//
float getCurrentVelocityInMillimetersPerSecond()
```

Functions with units in revolutions:

```
//
// set the number of steps the motor has per revolution
//
void setStepsPerRevolution(float motorStepPerRevolution)


//
// get the current position of the motor in revolutions, this functions is updated
// while the motor moves
//  Exit:  a signed motor position in revolutions returned
//
float getCurrentPositionInRevolutions()
```

```
//
// set current position of the motor in revolutions, this does not move the motor
//
void setCurrentPositionInRevolutions(float currentPositionInRevolutions)


//
// set the maximum speed, units in revolutions/second, this is the maximum speed
// reached while accelerating.  Note: this can only be called when the motor is
// stopped
//  Enter:  speedInRevolutionsPerSecond = speed to accelerate up to, units in
//            revolutions/second
//
void setSpeedInRevolutionsPerSecond(float speedInRevolutionsPerSecond)


//
// set the rate of acceleration, units in revolutions/second/second
// Note: this can only be called when the motor is stopped
//  Enter:  accelerationInRevolutionsPerSecondPerSecond = rate of acceleration,
//            units inrevolutions/second/second
//
void setAccelerationInRevolutionsPerSecondPerSecond(
                     float accelerationInRevolutionsPerSecondPerSecond)


//
// home the motor by moving until the homing sensor is activated, then set the
// position to zero, with units in revolutions
//  Enter:  directionTowardHome = 1 to move in a positive direction, -1 to move in
//             a negative directions
//          speedInRevolutionsPerSecond = speed to accelerate up to while moving
//             toward home, units in revolutions/second
//          maxDistanceToMoveInRevolutions = unsigned maximum distance to move
//             toward home before giving up
//          homeSwitchPin = pin number of the home switch, switch should be
//             configured to go low when at home
//  Exit:   true returned if successful, else false
//
bool moveToHomeInRevolutions(long directionTowardHome,
  float speedInRevolutionsPerSecond, long maxDistanceToMoveInRevolutions,
  int homeLimitSwitchPin)


//
// move relative to the current position, units are in revolutions, this function
// does not return until the move is complete
//  Enter:  distanceToMoveInRevolutions = signed distance to move relative to the
//          current position in revolutions
//
void moveRelativeInRevolutions(float distanceToMoveInRevolutions)
```

```
//
// setup a move relative to the current position, units are in revolutions, no
// motion occurs until processMove() is called
//  Enter:  distanceToMoveInRevolutions = signed distance to move relative to the
//            currentposition in revolutions
//
void setTargetPositionRelativeInRevolutions(float distanceToMoveInRevolutions)


//
// move to the given absolute position, units are in revolutions, this function
// does not return until the move is complete
//  Enter:  absolutePositionToMoveToInRevolutions = signed absolute position to
//          move to in units of revolutions
//
void moveToPositionInRevolutions(float absolutePositionToMoveToInRevolutions)


//
// setup a move, units are in revolutions, no motion occurs until processMove()
// is called
//  Enter:  absolutePositionToMoveToInRevolutions = signed absolute position to
//          move to in units of revolutions
//
void setTargetPositionInRevolutions(float absolutePositionToMoveToInRevolutions)


//
// Get the current velocity of the motor in revolutions/second.  This functions is
// updated while it accelerates up and down in speed.  This is not the desired
// speed, but the speed the motor should be moving at the time the function is
// called.  This is a signed value and is negative when motor is moving backwards.
// Note: This speed will be incorrect if the desired velocity is set faster than
// this library can generate steps, or if the load on the motor is too great for
// the amount of torque that it can generate.
//  Exit:  velocity speed in revolutions per second returned, signed
//
float getCurrentVelocityInRevolutionsPerSecond()
```

Functions with units in steps:

```
//
// set the current position of the motor in steps, this does not move the motor
// Note: This function should only be called when the motor is stopped
//    Enter:  currentPositionInSteps = the new position of the motor in steps
//
void setCurrentPositionInSteps(long currentPositionInSteps)
```

```
//
// get the current position of the motor in steps, this functions is updated
// while the motor moves
//  Exit:  a signed motor position in steps returned
//
long getCurrentPositionInSteps()


//
// set the maximum speed, units in steps/second, this is the maximum speed reached
// while accelerating
// Note: this can only be called when the motor is stopped
//  Enter:  speedInStepsPerSecond = speed to accelerate up to, units in steps/second
//
void setSpeedInStepsPerSecond(float speedInStepsPerSecond)


//
// set the rate of acceleration, units in steps/second/second
// Note: this can only be called when the motor is stopped
//  Enter:  accelerationInStepsPerSecondPerSecond = rate of acceleration, units in
//          steps/second/second
//
void setAccelerationInStepsPerSecondPerSecond(
                     float accelerationInStepsPerSecondPerSecond)


//
// home the motor by moving until the homing sensor is activated, then set the
// position to zero with units in steps
//  Enter:  directionTowardHome = 1 to move in a positive direction, -1 to move in
//             a negative directions
//          speedInStepsPerSecond = speed to accelerate up to while moving toward
//             home, units in steps/second
//          maxDistanceToMoveInSteps = unsigned maximum distance to move toward
//             home before giving up
//          homeSwitchPin = pin number of the home switch, switch should be
//             configured to go low when at home
//  Exit:   true returned if successful, else false
//
bool moveToHomeInSteps(long directionTowardHome,
  float speedInStepsPerSecond, long maxDistanceToMoveInSteps, int homeLimitSwitchPin)


//
// move relative to the current position, units are in steps, this function does
// not return until the move is complete
//  Enter:  distanceToMoveInSteps = signed distance to move relative to the current
//            position in steps
//
void moveRelativeInSteps(long distanceToMoveInSteps)
```

```
//
// setup a move relative to the current position, units are in steps, no motion
// occurs until processMove() is called
//  Enter:  distanceToMoveInSteps = signed distance to move relative to the current
//            positionin steps
//
void setTargetPositionRelativeInSteps(long distanceToMoveInSteps)


//
// move to the given absolute position, units are in steps, this function does not
// return until the move is complete
//  Enter:  absolutePositionToMoveToInSteps = signed absolute position to move to
//            in units of steps
//
void moveToPositionInSteps(long absolutePositionToMoveToInSteps)


//
// setup a move, units are in steps, no motion occurs until processMove() is called
//  Enter:  absolutePositionToMoveToInSteps = signed absolute position to move to
//            in units of steps
//
void setTargetPositionInSteps(long absolutePositionToMoveToInSteps)


//
// setup a "Stop" to begin the process of decelerating from the current velocity
// to zero, decelerating requires calls to processMove() until the move is complete
// Note: This function can be used to stop a motion initiated in units of steps
// or revolutions
//
void setTargetPositionToStop()


//
// if it is time, move one step
//  Exit:  true returned if movement complete, false returned not a final target
//           position yet
//
bool processMovement(void)


//
// Get the current velocity of the motor in steps/second.  This functions is updated
// while it accelerates up and down in speed.  This is not the desired speed, but
// the speed the motor should be moving at the time the function is called.  This
// is a signed value and is negative when the motor is moving backwards.
// Note: This speed will be incorrect if the desired velocity is set faster than
// this library can generate steps, or if the load on the motor is too great for
// the amount of torque that it can generate.
```

```
//  Exit:  velocity speed in steps per second returned, signed
//
float getCurrentVelocityInStepsPerSecond()


//
// check if the motor has competed its move to the target position
//  Exit:  true returned if the stepper is at the target position
//
bool motionComplete()
```