Force ramping curves

Background

In our application we often need to control the tension that we apply to the cable and ensure that it is changed safely in case a user is under the load we are generating. If the load/weight is changed too quickly the cable could be jerked in an uncomfortable manner, if it is changed too slowly then the user is waiting around to get to the desired weight for lifting. Tonal's internal user testing has shown that a trapezoidal ramp profile has the best user experience - It allows us to increase weight the most quickly while still allowing users to maintain control.

Problem Statement

Your task is to write code in C that, when invoked, transitions a user's weight from the current weight (a.k.a. "Starting weight") to the desired weight (a.k.a "end weight") by making many incremental weight changes. These weight changes can be assumed to take place instantaneously.

The end result will be a trapezoidal weight ramping profile. A trapezoidal weight ramping curve has a lot of similarities to the trapezoidal speed ramping curve described here - see the section labeled "Trapezoidal move profile":

https://www.linearmotiontips.com/how-to-calculate-velocity/

Your trapezoidal weight ramp should aim to complete the transition to the end weight as quickly as possible while not violating the bounds as listed in the requirements below.

Requirements

1. All work should be original - we love to reuse open source work but we would like to review original works for this problem
2. When changing weight upwards (i.e. increasing weight), the maximum ramp rate is 80 lbs/sec
3. When changing weight downwards (i.e. decreasing weight), the maximum ramp rate is -60 lbs/sec
4. ~⅙ of the total weight should be ramped in the first phase of the trapezoid
5. ~⅔ of the total weight should be ramped in the second phase of the trapezoid
6. ~⅙ of the total weight should be ramped in the third phase of the trapezoid
7. The ramp profile file(s) should be structured in a way that it would be easily ported to a bare-metal embedded processor

8. Architect your code in a way that the ramping profile can be run multiple times concurrently - (assume there are multiple systems that need to utilize this code that may run in parallel)
9. Assume that the calculation of your ramping curve is called from an ISR every 20 ms
10. The input weight is a variable that can be easily changed anywhere from 5 to 100 lbs in different ramp contexts

Deliverables

When you have completed your solution, please provide:

1. Your implementation and all related sources
2. A list of any assumptions you made
3. A list of the parameters used to test your final solution
4. Posix based project including the ramping profile, build instructions and test harness.
5. Sample .csv for a 5-100 lb ramp with each row including the timestamp, total weight, and incremental weight applied during that time-step.
6. A image of a graph showing the sample .csv from the 5-100lb ramp - see the provided plot_weight.py for help (more documentation below)
7. Be prepared to discuss architectural decisions that were made as well as design tradeoffs around latency, resource usage, maintainability, etc.

Support

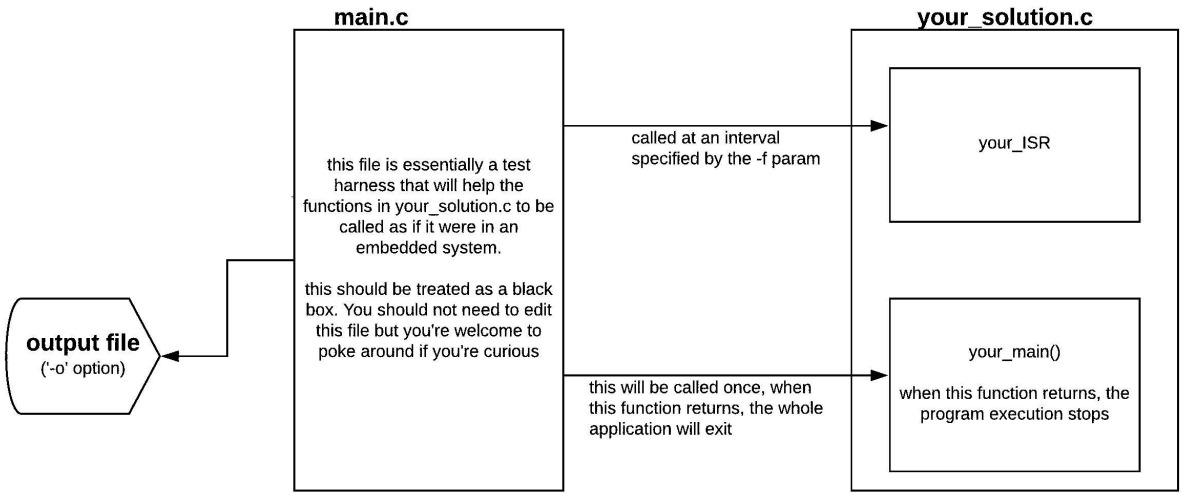
It is not our intention that you spin your wheels trying to figure out our test harness. If something is unclear please reach out so we can get you started. If you have any other questions about the problem statement, please let us know so we can clarify or just make an assumption and write it down in your code comments so we can discuss. We hope you will enjoy this exercise.

Hardware Emulation

The rest of this document is aimed at getting you comfortable with the emulation software (a.k.a. “Test harness”) that has been provided with this problem statement. Refer to it as needed.

Hardware Emulation Test Harness

In order to aid you in this exercise, we have provided a testing harness that emulates the hardware available in an embedded system. There are two relevant files, main.c and your_solution.c. The main.c file that is provided manages the hardware emulation and should not need to be edited or even read … unless you are curious. A makefile has also been provided that will build and link all .c files in the working directory. It was intended that your work be done in your_solution.c



The following command line options are available in the test harness:

Option	Meaning	required	Notes
-o	Output file name	yes	This file will be a .csv containing “time (ms), weight (lb)” pairs needed to plot with the included plot_weight.py. The contents of this file are generated when calls to applyWeight() are made. When the program launches, the starting weight is automatically logged
-s	Starting weight	yes	This is the starting weight or current weight. It can be anywhere from 5 to 100 lbs.
-e	Ending weight	yes	This is the ending weight or target weight. It can be anywhere from 5 to 100 lbs.

-f	Delay between calls to YOUR_ISR()	yes	Takes as a parameter a millisecond delay. You should pick this value to reasonably match the accel sample rate. Don't worry about jitter, etc.
-d	Debug option	no	If this flag is provided, the print_debug_log() function will cause the incoming strings to be printed like 'printf()'. If this flag is not provided the print_debug_log() will have no effect The get_is_debug() function will return this value as a boolean, true if set, false otherwise

Your_Solution.c

You are welcome to create additional files as and if needed, but it is intended that your solution would live in the file your_solution.c that is provided. You will find 2 functions in your_solution.c that are important to this question:

```
/*
 * This function is the equivalent of a typical main() function.
 * This thread will run at a lower priority than YOUR_ISR().
 * When this function returns, the harness will cause program execution to end immediately.
 */
void your_main(void);
```

```
/*
 * This function simulates an on-target ISR. it is called at a regular interval
 * at a higher priority than the your_main() function.
 *
 * The rate at which this function is called by the test harness can be set
 * with the '-f' option. the '-f' option takes the time between calls in
 * milliseconds as the parameter
 */
void YOUR_ISR(void);
```

Helper & Emulator functions:

We have provided the following function declarations for you to use during development in util.h, they exist to help you and you can choose to use them or not.

```
// this function prints a log out when the '-d' option is provided on the command line
int print_debug_log(const char *format, ...);

// this function will return the state of the '-d' option. True if '-d' is set. False otherwise.
bool get_is_debug(void);

// get the current system time in microseconds
uint64_t current_timestamp_us(void);

/* set the weight the motor is currently generating. this weight will be applied
```

```
 * instantaneously. it is up to you to make subsequent calls to this function
 * to get a weight ramp that follows the bounds of the question - instead of applying the weight,
 * this emulation hardware just logs this to the output file provided in order to generate a plot */
void apply_weight(float weight_lbs);

/* get the starting weight or current weight as set by the command line option */
uint8_t get_start_weight(void);

/* get the ending or desired weight as set by the command line option */
uint8_t get_end_weight(void);
```

A note about control systems in this problem

For those familiar with control systems, you can consider calls to apply_weight() to be what sets our reference trajectory in our underlying controller. If this is not helpful or does not hold meaning for you please feel free to ignore.

Using The Emulator

This section is here purely to help you get acclimated to this question more quickly.

Software Needed

The emulator was developed on a Mac v10.15.4 with the following software versions. It has also been tested on Ubuntu 16.04 LTS

```
Make --version
GNU Make 3.81
```

```
 gcc --version
Configured with: --prefix=/Library/Developer/CommandLineTools/usr
--with-gxx-include-dir=/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/c++/4.2.1
Apple clang version 11.0.0 (clang-1100.0.33.17)
Target: x86_64-apple-darwin19.4.0
```

Running the Emulator

In order to build all .c sources into your solution, you can run make against the provided makefile:

```
>>make
gcc -o solution main.c your_solution.c -Wall
```

After building, you can run the sample code we have provided at a 20 Hz ISR frequency in debug mode by running the following command. The starting weight is 5 lbs. The ending weight is 100 lbs and we're generating test.csv.

```
>> ./solution -f 20 -o test.csv -s 5 -e 100 -d
starting application
time: 1601659969923501. delta = 1601659969923501
time: 1601659969944163. delta = 20662
time: 1601659969965192. delta = 21029
time: 1601659969986233. delta = 21041
... <more timestamps here> ...
time: 1601659971844662. delta = 21047
time: 1601659971865068. delta = 20406
time: 1601659971886112. delta = 21044
ending application
```

You can also run the sample code silently by removing the -d option.

```
>> ./solution -f 20 -o test.csv -s 5 -e 100
```

You will see the test.csv file now contains time (ms) and weight. Note that the starting weight is automatically added as the first row.

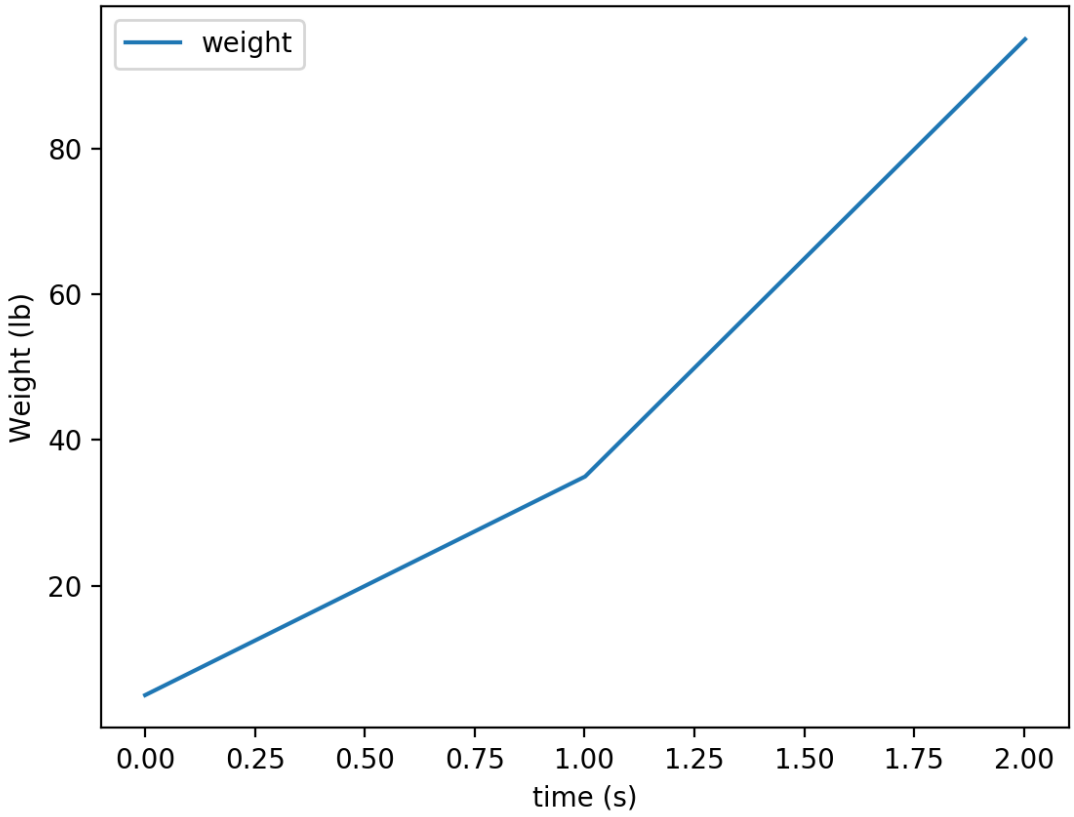
```
>>cat test.csv
0.000000, 5.00
1001.312000, 35.00
2002.486000, 95.00
```

Plotting the Result

Now that you have run your weight ramp simulation, you can now run the python plotting script on the output file. Remember, each row in the output file is generated from a single call to the apply_weight() function :

```
>> python3 plot_weights.py -f test.csv
```

The starting code will generate the following plot if untouched:


weight
80
60
40
20
Weight (lb)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
time (s)