




















TABLE OF CONTENTS

| | | |
|-------|---------------------------------|---|
| 1 | OOP Design..... | 2 |
| 2 | PID Controller | 2 |
| 2.1 | Basics..... | 2 |
| 2.2 | Implementation..... | 3 |
| 2.2.1 | Header file | 3 |
| 2.2.2 | PID..... | 4 |
| 2.2.3 | Scaling | 4 |
| 3 | Serial Communication | 6 |
| 3.1 | Interface | 6 |
| 3.1.1 | Machine Code..... | 6 |
| 3.1.2 | PC Message..... | 7 |
| 3.1.3 | Feedback (Acknowledgement)..... | 7 |
| 3.1.4 | Sensor Readings | 8 |
| 3.2 | Implementation..... | 9 |
| 3.2.1 | Header File | 9 |

Please disregard the table of content when converting this document to WordPress.

1 OOP DESIGN

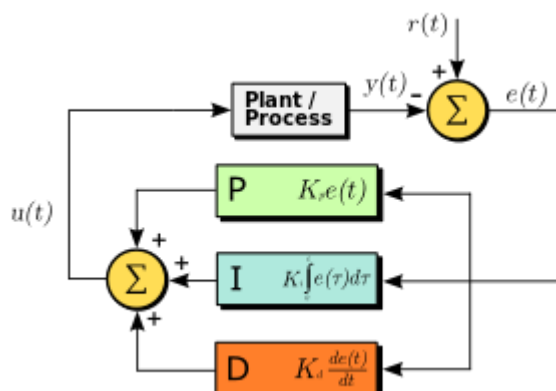
The object-oriented design is implemented for Arduino to enhance the readability and maintainability.

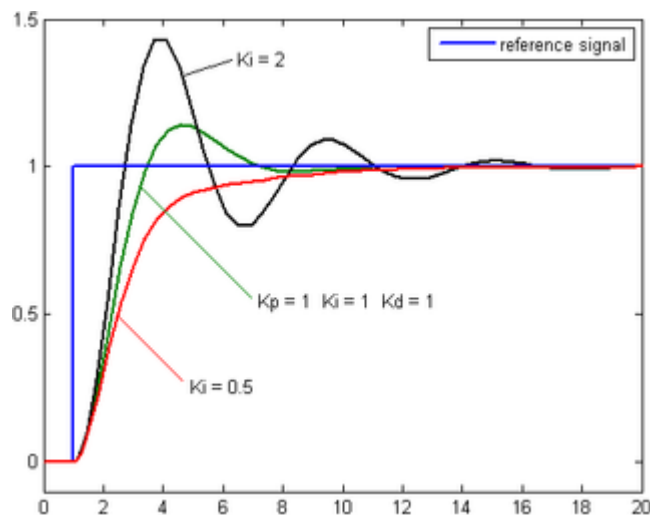
| | | | |
|--|-----------------|--------------|-------|
|  Calibrator.ino | 3/30/2014 13:24 | Arduino file | 4 KB |
|  Config.ino | 3/30/2014 14:07 | Arduino file | 2 KB |
|  ErrorCumulator.ino | 3/30/2014 9:54 | Arduino file | 6 KB |
|  Eyes.ino | 3/30/2014 14:01 | Arduino file | 4 KB |
|  mains.ino | 3/30/2014 13:11 | Arduino file | 2 KB |
|  movements.ino | 3/30/2014 9:54 | Arduino file | 9 KB |
|  PidMgr.ino | 3/31/2014 11:21 | Arduino file | 2 KB |
|  Serial.ino | 3/30/2014 14:06 | Arduino file | 5 KB |
|  test.ino | 3/30/2014 12:42 | Arduino file | 1 KB |
|  utils.ino | 3/30/2014 14:03 | Arduino file | 3 KB |
|  Calibrator.h | 3/30/2014 12:37 | H File | 1 KB |
|  Config.h | 3/30/2014 12:21 | H File | 1 KB |
|  ErrorCumulator.h | 3/30/2014 9:54 | H File | 2 KB |
|  Eyes.h | 3/30/2014 12:40 | H File | 2 KB |
|  globals.h | 3/30/2014 12:48 | H File | 2 KB |
|  PidMgr.h | 3/30/2014 9:54 | H File | 1 KB |
|  Pin.h | 3/30/2014 9:52 | H File | 3 KB |
|  PinChangeInt.h | 3/30/2014 9:52 | H File | 21 KB |
|  Serial.h | 3/30/2014 9:54 | H File | 1 KB |

2 PID CONTROLLER

2.1 BASICS

A proportional-integral-derivative controller (PID controller) is a control loop feedback mechanism (controller) used in robotics. A PID controller calculates an "error" value as the difference between a measured process variable and a desired setpoint. The controller attempts to minimize the error in outputs by adjusting the process control inputs.





2.2 IMPLEMENTATION

The Arduino PID library is used. You can find it on [GitHub](https://github.com/br3ntn/Arduino-PID-Library).

2.2.1 HEADER FILE

```
#ifndef PIDMGR_H
#define PIDMGR_H
#include <PID_v1.h>
class PidMgr {
public:
    PidMgr();
    void setScale(double scale);
    void init();
    void restore();
    double getCurrentScale();

    double SetpointLeft, InputLeft, OutputLeft;
    double SetpointRight, InputRight, OutputRight;
    double SetpointMid, InputMid, OutputMid;

    PID* leftPID;
    PID* rightPID;
    PID* midPID;

private:
    double current_scale;

    static const double kp=0.5, ki=0.25, kd=0;
    static const double kp_mid=1, ki_mid=0.05, kd_mid=0.25;
};
#endif
```

2.2.2 PID

PidMgr has three PID calculators for left motor, right motor, and center of robot respectively.

```
PID* leftPID;  
PID* rightPID;  
PID* midPID;
```

Coefficients are tested and chosen empirically

```
static const double kp=0.5, ki=0.25, kd=0;  
static const double kp_mid=1, ki_mid=0.05, kd_mid=0.25;
```

2.2.3 SCALING

Scaling the set point, lower limit, and upper limit in order to make the robot can change the speed during the run time while maintain the accuracy of PID calculation.

```
void PidMgr::setScale(double scale) {  
    this->SetpointLeft *= scale;  
    this->SetpointRight *= scale;  
    this->SetpointMid *= scale; // always 0  
  
    Config::PID_SETPOINT *= scale;  
    Config::PID_UPPER_LIMIT *= scale;  
    Config::PID_LOWER_LIMIT *= scale;  
  
    Config::MAX_SPEED *= scale;  
    Config::TARGET_SPEED *= scale;  
    Config::MIN_SPEED *= scale;  
  
    this->leftPID->SetOutputLimits(Config::PID_LOWER_LIMIT,  
    Config::PID_UPPER_LIMIT);  
    this->rightPID->SetOutputLimits(Config::PID_LOWER_LIMIT,  
    Config::PID_UPPER_LIMIT);  
    this->midPID->SetOutputLimits(-Config::PID_SETPOINT/2,  
    Config::PID_SETPOINT/2);  
  
    this->current_scale *= scale;  
}
```

2.2.4 INPUTS CALCULATION

We have predefined a set point, but we need to look for the Input of PID thus PID will then calculate the Output. In the movments.ino:

2.2.4.1 TRAVEL DISTANCE

In every while loop, use the encoder to get number of ticks and then calculate the robot's travel distance

```
leftPololuCount = leftCnt;
rightPololuCount = rightCnt;
long timez = millis() - timing; // time passed by
// incremental
double leftTicks = abs(leftPololuCount - previousLeftTick); //
positive for forward, negative for backward
double rightTicks = abs(rightPololuCount - previousRightTick);

// distanceToTravel is incremental
double leftcm = Config::DISTANCE_PER_TICK_CM * leftTicks;
double rightcm = Config::DISTANCE_PER_TICK_CM * rightTicks;
double distanceToTravel = (leftcm + rightcm)/2.0;
```

2.2.4.2 PID INPUT

The speed of robot is calculated by numberOfTicks / deltaTime

```
leftTicks /= (timez/1000.0);
rightTicks /= (timez/1000.0); // ms
if(leftPololuCount*previousLeftTick>0 &&
rightPololuCount*previousRightTick>0) { // avoid overflow
    pidMgr->InputLeft = leftTicks;
    pidMgr->InputRight = rightTicks;
    pidMgr->InputMid = errorCumulator->deltaY /
Config::DISTANCE_PER_TICK_CM;
}
```

2.2.4.3 PID OUTPUT

The output of PID is calculated from PID->Compute(). The PID output is then mapped to the motor speed from 0 – 400 to let the PololuMotorShield to drive the motor.

```
pidMgr->midPID->Compute();
pidMgr->SetpointRight = Config::PID_SETPOINT + map(
    pidMgr->OutputMid, -Config::PID_SETPOINT/2,
    Config::PID_SETPOINT/2,
    -Config::PID_SETPOINT,
    +Config::PID_SETPOINT
);
pidMgr->rightPID->Compute();
pidMgr->leftPID->Compute();
```

```
previousLeftTick = leftPololuCount;
previousRightTick = rightPololuCount;
timing = millis();

int m1Speed = isM1Forward * map(pidMgr->OutputLeft,
Config::PID_LOWER_LIMIT, Config::PID_UPPER_LIMIT, Config::MIN_SPEED,
Config::MAX_SPEED);
int m2Speed = isM2Forward * map(pidMgr->OutputRight,
Config::PID_LOWER_LIMIT, Config::PID_UPPER_LIMIT, Config::MIN_SPEED,
Config::MAX_SPEED);
motorShield.setSpeeds(m1Speed, m2Speed);
```

3 DEAD RECKONING

TODO

4 SERIAL COMMUNICATION

4.1 INTERFACE

4.1.1 MACHINE CODE

Command to be executed is using 7-digit machine code since it saves memory on Arduino;

```
# in Python

machine_code = "xxxxxxx"

machine_code[0:2] # constructs 2-digit function codes
machine_code[2:7] # constructs parameter with 2 decimal places
```

example:

```
"0012312"
```

This code is translated into function code 00 with parameter 123.12

The example commands Arduino to moveForward by 123.12 cm.

List of function codes currently available:

```
00: void moveForward(double dist);
01: void turnLeft(double angle);
02: void turnRight(double angle);

10: void getSensorReadings(); // ad-hoc request for sensor readings (ad-hoc).

20: moveForward with Sensor Reading (returns two packages of data)
21: turnLeft with Sensor Reading
22: turnRight with Sensor Reading

98: void calibrate(int situation); // see calibration section below
```

NOTICE: In python each json serial command at very end must end with

```
'\n'
```

NOTICE: incoming serial buffer size of Arduino is limited, thus only 7 concurrent machine code command is allowed

4.1.2 PC MESSAGE

```
{
  "function": function_code,
  "parameter": parameter
}
```

4.1.3 FEEDBACK (ACKNOWLEDGEMENT)

Json format ([Python Reference](#))

```
{
  "function": function_code,
  "status": status_code
}
```

example:

```
{"function": 0, "status": 200}
```

status_code is the HTTP status code as in [this](#).

The most common status code is 200 (i.e. OK).

@Deprecated

```
{"function": 99, "status": 200}
```

@Deprecated 99 denotes the Arduino is ready to do serial communication

4.1.4 SENSOR READINGS

In the case you don't know json array, refer to [this](#) and [this](#).

```
{
  "sensors": [
    {"sensor": sensor_code, "value": return_value},
    {"sensor": sensor_code, "value": return_value},
    {"sensor": sensor_code, "value": return_value},
    {"sensor": sensor_code, "value": return_value},
    {"sensor": sensor_code, "value": return_value},
    {"sensor": sensor_code, "value": return_value}
  ]
}
```

This json array indicates obstacle distances (in cm) from front, left, right. The distances are from periphery of the robot.

Example

```
{"sensors":[{"sensor":0,"value":100}, {"sensor":1,"value":30}, {"sensor":2,"value":30}, {"sensor":10,"value":30}, {"sensor":11,"value":30}, {"sensor":12,"value":30}]}
```

List of sensor codes currently available:

```
0: front ultra sensor; // sensor range 10 - 90 cm
1: front left sensor; // 10 - 40 cm
2: front right sensor; // 10 - 40 cm
10: sided left sensor; // 0 - 30 cm
```



```
11: sided right sensor; // 0 - 30 cm
12: sided ultra sensor; // 10 - 90 cm
```

Notice: if the distance is beyond the sensor's range, it will return -1;

4.2 IMPLEMENTATION

4.2.1 HEADER FILE

```
#ifndef SerialSenderReceiver_H
#define SerialSenderReceiver_H
// #include <aJSON.h> // https://github.com/interactive-matter/aJson
class SerialCommnder
{
public:
    SerialCommnder();
    bool receive_exec_command();
    void send_command_complete(int function_code, int status_code);
    void send_sensor_readings(int front_value, int front_left_value,
int front_right_value, int left_value, int right_value, int
side_ultra_value);
    void send_ready_signal();
    int get_command();

private:
    int command;
    bool is_started;
    bool exec_command(int function_code, double parameter);
    void routine_clean(int function_code);
};
#endif
```