UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

RAFAEL CAZAROTTO

CONSTRUÇÃO DE UM MÓDULO DIDÁTICO DE BAIXO CUSTO PARA CONTROLE DE NÍVEL USANDO O ARDUINO

FRANCISCO BELTRÃO

2021

RAFAEL CAZAROTTO

CONSTRUÇÃO DE UM MÓDULO DIDÁTICO DE BAIXO CUSTO PARA CONTROLE DE NÍVEL USANDO O ARDUINO

Construction of a low-cost teaching module for level control using arduino

Trabalho de conclusão de curso de graduação apresentada como requisito para obtenção do título de Bacharel em Engenharia química da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador(a): Prof. Dr. Douglas Junior Nicolin.

Coorientador(a): Prof. Dr. Jonas Joacir Radtke.

FRANCISCO BELTRÃO

2021

RAFAEL CAZAROTTO

CONSTRUÇÃO DE UM MÓDULO DIDÁTICO DE BAIXO CUSTO PARA CONTROLE DE NÍVEL USANDO O ARDUINO

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia Química da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 10/dezembro/2021

_____

Douglas Junior Nicolin
Doutorado
Universidade Tecnológica Federal do Paraná

_____

Jonas Joacir Radtke
Doutorado
Universidade Tecnológica Federal do Paraná

_____

André Zuber
Doutorado
Universidade Tecnológica Federal do Paraná

"A folha de aprovação assinada encontra-se na Coordenação do Curso"

FRANCISCO BELTRÃO

2021

AGRADECIMENTOS

Gostaria de agradecer a todos os meus amigos e familiares que mesmo sem saber, me ajudaram de inúmeras maneiras.

Gostaria de agradecer a minha mãe Maristela Padilha e ao meu pai Clademir Luiz Cazarotto, só sou o que sou pela excepcional criação e cuidado que sempre recebi. Além de sempre me apoiarem, ambos sorriram e choraram comigo em inúmeros momentos, sofreram e comemoraram, independente da distância. Afirmo com toda certeza de que foram os maiores responsáveis por todas as conquistas que obtive.

Impossível esquecer dos meus irmãos nesse momento, Luiz Fernando Lorenci e Mateus Cazarotto, por me proporcionarem um apoio imensurável em situações difíceis, além de todos os momentos de distração que ajudaram a aliviar o estresse e a pressão.

Gostaria de agradecer a todos os meus amigos de Francisco Beltrão – PR, em especial, meu colega de apartamento Chistopher de Lima Ferreira por comemorar e sofrer comigo nesta fase tão importante. Gostaria de agradecer os almoços feitos por ele em todos os momentos que estava muito atarefado, como ele costuma dizer “te alimente guri”.

Minha vontade era de citar todas as pessoas que foram importantes nessa etapa e comentar um pouco sobre cada, porém meu espaço é limitado, e não quero deixar ninguém de fora. Saibam que todos possuem um lugar muito especial em minha vida, e não preciso citar todos os nomes, pois, se você é uma delas, já te deixei isso bem claro.

Por último, mas não menos importante, gostaria de agradecer ao meu orientador Douglas Nicolin por toda a ajuda prestada, e por toda a atenção que me foi dada. Gostaria de agradecer também ao meu coorientador Jonas Radtke, além da ajuda prestada no trabalho, ele foi o responsável por despertar esse interesse nos microcontroladores.

Muito obrigado!

“A imaginação muitas vezes nos leva a mundos que nunca sequer existiram. Mas sem ela não vamos a lugar algum.”

(Carl Sagan).

RESUMO

Controle de processos é uma necessidade industrial. Diversos processos industriais são passiveis de implementação de métodos de controle, sendo o controlador PID (Proporcional Integral Derivativo) o mais utilizado. Por esse motivo é importante que o perfil do egresso do curso de graduação em Engenharia possua conhecimento nessa área. Devido ao grau de complexidade das matérias do curso de engenharia, módulos didáticos são utilizados como ferramenta educacional objetivando um melhor aprendizado, a matéria de controle de processos se beneficia muito bem dessas ferramentas, porém, módulos didáticos de controle possuem alto custo, dificultando sua aquisição. Por esse motivo, o objetivo desse trabalho de conclusão de curso é a construção de um módulo didático de controle de nível, possibilitando a aplicação do controlador PID e seus derivados (Proporcional e/ou Integral e/ou Derivativo) para utilização em âmbito educacional, proporcionando uma melhor experiência para os estudantes de engenharia. A fim de minimizar os custos, o controlador utilizado foi uma placa Arduino Nano, a qual possui custo relativamente baixo e uma fácil implementação, o material base utilizado para a confecção do módulo foi reaproveitado de projetos antigos do FabLab do *campus* Francisco Beltrão, da Universidade Tecnológica Federal do Paraná.

Palavras-chave: Controle de processos; Módulo didático; PID.

ABSTRACT

Process control is an industrial necessity, several industrial processes are passive in the implementation of methods of control, being the PID controller (Proportional Integral Derivative) the most used. For this reason, it is important that the profile of graduates from the undergraduate course of Engineering has knowledge in this area. Due to the degree of complexity of the subjects of the Engineering course, didactic modules are used as an educational tool aiming at better learning, the subject of process control benefits very well from these tools, however, didactic control modules have a high cost, making their acquisition difficult. For this reason, the objective of this final paper is to build a didactic module of level control, enabling the application of the PID controller and its derivatives (Proportional and / or Integral and / or Derivative) to use it in the educational environment, providing a better experience for engineering students. In order to minimize costs, the controller used was an Arduino Nano board, which has relatively low cost and easy implementation. The base material used to make the module was reused from old FabLab projects at the Francisco Beltrão campus, at Universidade Tecnológica Federal of Paraná.

Keywords: process control; didactic modules; pid.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SÍMBOLOS

P	Controlador proporcional.
I	Controlador integral.
D	Controlador derivativo.
PI	Controlador proporcional integral.
PD	Controlador proporcional derivativo.
PID	Controlador proporcional integral derivativo.
PWM	*Pulse Width Modulation*.
SP	*Setpoint*.
E	Variável de erro.
K_p	Constante proporcional.
K_i	Constante integral.
K_d	Constante derivativa.

SUMÁRIO

1 INTRODUÇÃO

O controle de processos consiste na técnica de manter variáveis de um processo em valores desejados para o sistema (ROGGIA; FUENTES, 2016). Certos processos industriais requerem um nível mais elevado de atenção. Desta maneira sistemas de controle automatizados foram desenvolvidos para suprir essas necessidades, tornando os processos mais eficientes, seguros e menos suscetíveis a erros humanos de operação. Dentre os sistemas passíveis de automatização estão os sistemas de nível de líquidos, que são importantes em diversas aplicações, como na indústria química, petroquímica, nuclear e de celulose (GOSMANN, 2002).

O controle de nível é um problema em várias aplicações industriais, como no tratamento de água, nos tanques de armazenamento de líquidos e em reatores nas plantas industriais (ZHOU et al., 2017). A fim de promover a segurança operacional, o aumento da eficiência do processo e a qualidade dos equipamentos, um controle de nível é necessário. Esse controle mantém o processo o mais próximo possível do planejado, mantendo as variáveis em seus valores desejados, além de garantir a manutenção do balanço de massa da planta em quantidades adequadas.

Medir a variável de nível em processos industriais é quantificar referências por meio de monitoramento contínuo ou discreto, com o objetivo de avaliar e controlar volumes em tanques ou recipientes de armazenamento. O monitoramento contínuo é definido como a medição do nível real no tanque ou recipiente e o monitoramento discreto é baseado na análise de haver ou não líquidos em determinado ponto de interesse dos tanques. A detecção de nível depende de vários fatores, como o tipo de líquido e seu comportamento, o formato do tanque, condições de sujidade do ambiente, entre outros fatores (STEVAN, 2015). Realizar a medição corretamente possibilita implementar um controle de nível em diversas áreas industriais.

Vários controladores estão disponíveis no mercado para implementação. Dentre eles estão os controladores Proporcional (P), Proporcional e Derivativo (PD), Proporcional e Integral (PI) e Proporcional, Integral e Derivativo (PID) são os que mais se destacam e são mais utilizados em diversos ramos da engenharia, sobretudo no ramo industrial (SIMONELLI et al., 2017). Os controladores PID são projetados para inúmeros sistemas, possuem baixo custo e relativa facilidade de projeto (MAITELLI; CARVALHO, 2003). É de suma importância um profissional qualificado para realizar

a sintonia dos parâmetros do controlador, pois uma sintonia conduzida de maneira errada resulta em problemas no processo.

A graduação de engenharia possui temas cujo grau de complexidade é muito alto, o que pode ocasionar uma falta de entusiasmo aos estudantes dificultando assim o desenvolvimento dessas competências (WAWGINIAKS, 2017). Reproduzir situações industriais em escala laboratorial permite ao estudante avaliar diferentes parâmetros para o processo e observar, na prática, a influência direta de sua decisão, além do fato de que o ensino na área das engenharias é constituído comumente pela transmissão de informações, onde o aluno se habitua a ouvir e repetir encadeamentos lógicos. Dessa maneira o aluno entra em uma zona de conforto que acaba limitando sua capacidade de tomar decisões.

Alunos que não ultrapassam o limite de seguir o pensamento de outrem estarão sempre em desvantagem no mundo competitivo (BIANCHINI; GOMES, 2007). Nesse cenário, os módulos didáticos tornam-se atrativos para um ensino mais dinâmico.

Os módulos experimentais são importantes na graduação de engenharia porque permitem uma didática mais ativa, possibilitando um contato direto do estudante com a simulação de um problema real. Esse contato permite uma visualização em tempo real da tomada de decisão e estimula a capacidade de análise do estudante, resultando em uma compreensão mais aprofundada do problema, possibilitando uma formação mais completa.

Segundo o Conselho Nacional de Educação, Resolução n° 2, de 24 de abril de 2019 Capítulo II, Art. 3º inciso I e II, que estabelece as novas Diretrizes Curriculares Nacionais para os cursos de Engenharia, é esperado um perfil do egresso do curso de graduação em Engenharia que contenha as seguintes características: ser crítico, reflexivo, criativo, com forte formação técnica e estar apto a utilizar novas tecnologias com atuação inovadora. Já o Capítulo III, referente à organização do curso de graduação em engenharia, Art. 6º, § 2º do inciso VIII, menciona que se deve estimular atividades que articulem simultaneamente a teoria, a prática e o contexto de aplicação, necessárias para o desenvolvimento das competências do perfil de egresso (BRASIL, 2019). Dessa maneira módulos didáticos são essenciais para a formação consistente em engenharia, por cumprirem em escala laboratorial a articulação entre teoria e prática em sua plenitude. Porém, o alto custo desses módulos acaba dificultando sua propagação no âmbito educacional.

Um módulo de bancada da empresa Datapool®, modelo TNPV 2101B, com controle de temperatura, nível, pressão e vazão, por exemplo, custa R$ 49.671,10. Uma versão mais simples desse módulo pode ser construída com menos de R$ 5.000,00. Porém, com um menor número de variáveis de interesse.

Por mais que o modelo citado anteriormente possua quatro variáveis de interesse (vazão, nível, temperatura e pressão), analisar um sistema com apenas uma variável já torna o módulo proveitoso, devido à possiblidade de observar os efeitos de diferentes controladores em um mesmo sistema. Várias opções são passiveis de análise como: resposta do sistema a perturbações (com diferentes controladores), problema regulador (perturbações nas variáveis de entrada), problema servo (perturbações no ponto fixo), estabilidade etc.

O presente trabalho aborda a construção de um módulo de tanque com um sistema de controle de nível para a disciplina de Controle de Processos. A fim de reduzir os custos com a produção, o módulo foi construído no FabLab do campus Francisco Beltrão, da Universidade Tecnológica Federal do Paraná, utilizando materiais já adquiridos e destinados para esses projetos. O microcontrolador Arduino foi utilizado para controlar o sistema, o qual é uma plataforma open-source[1] de prototipagem eletrônica baseada em flexibilidade, com hardware e software fáceis de serem usados e adaptados aos mais diferentes cenários e aplicações (STEVAN JUNIOR, 2015). O Arduino foi responsável por realizar a leitura dos sensores, os cálculos e o controle dos periféricos. A configuração inicial do módulo foi feita utilizando um software desenvolvido para esse fim.

O software foi desenvolvido em Python[2] e utilizado como uma ferramenta para configuração dos parâmetros do microcontrolador e visualização dos dados obtidos, possibilitando a análise detalhada do experimento.

---

[1] *Open-source* é um termo utilizado para se referir a um software com seu código fonte disponibilizado para o público, podendo modificá-lo e distribui-lo conforme sua necessidade.

[2] Python é uma linguagem de programação de alto nível que utiliza orientação a objeto focada em produtividade e legibilidade (COUTINHO, 2019).

2 OBJETIVOS

2.1 Objetivo geral

Construir um módulo didático de controle de nível de líquidos em tanques com o uso do Arduino como aparato lógico de controle.

2.2 Objetivos específicos

- Desenvolver um protótipo estrutural para testes;
- Programar o código do microcontrolador na plataforma IDE[3] Arduino;
- Desenvolver um *software* para a comunicação entre computador e Arduino, utilizando a linguagem de programação Python;
- Construir a estrutura final do módulo;
- Executar testes de funcionamento do módulo para coleta de dados;

[3] A IDE do Arduino é um programa que tem como objetivo facilitar a programação e o envio do código para a plataforma Arduino.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Módulos didáticos

A busca pelo ensino eficaz precisa estar relacionada com as tecnologias modernas. Uma das características fundamentais para um profissional é o domínio dessas tecnologias propostas pela sociedade atual. Por esse motivo, apenas a simples passagem de conhecimento não basta, é necessário desenvolver algo que faça o estudante refletir e dialogar, aprimorando seu modo de pensar, agir e decidir (HECK, 2017). Essas competências são necessárias para um profissional de engenharia.

Considerando atender às novas expectativas do mercado, a adaptação dos cursos de engenharia é uma necessidade. A formação com base em competências tem exigido a reformulação dos cursos. Segundo o Conselho Nacional de Educação, que estabelece as novas Diretrizes Curriculares Nacionais para os cursos de Engenharia, é esperado que o estudante contenha as seguintes características após o término da graduação: ser crítico, reflexivo, criativo, com forte formação técnica e estar apto a utilizar novas tecnologias com atuação inovadora (BRASIL, 2019). Tendo isso em vista, é essencial despertar o interesse do aluno para desenvolver tais competências

Apesar de usados como sinônimos, os termos motivação e interesse possuem significados diferentes; motivação é associada com a disposição interna para a ação, enquanto interesse está relacionado ao fascínio em algo. O interesse é dividido em duas subcategorias, interesse pessoal e interesse situacional. A subcategoria pessoal está associada à preferência da pessoa quando possui a oportunidade de escolha, escolhendo o que prefere para estudar. Já o interesse situacional é estimulado em determinadas situações ou contextos, onde pode ser passível de influência por um determinado tempo. Dessa maneira, o professor possui a oportunidade de influenciar o aluno em um contexto específico (COUTO, 2009). Por esse motivo, módulos experimentais são ferramentas poderosas para a didática do docente.

Vários trabalhos se dedicaram à construção de módulos didáticos de controle de processos, como por exemplo o trabalho de conclusão de curso realizado pelo aluno Bruce Paiva Duarte em 2019, UTFPR, *campus* Francisco Beltrão. O trabalho

citado teve como objetivo controlar a temperatura de um reator fotocatalítico utilizando a alteração do fluxo de ar com o controlador PID (DUARTE, 2019).

Já o trabalho de conclusão de curso realizado pelo aluno Bruno Azevedo Ferraz De Negreiros em 2019, UTFPR, *campus* Francisco Beltrão, teve como objetivo a construção de um módulo didático de absorção. Em sua conclusão, consta uma validação do módulo como ferramenta educacional, sendo atribuída uma nota “ótima” pelos docentes. Além do custo reduzido, o módulo se mostrou útil para construir uma didática mais eficiente (NEGREIROS, 2019).

3.2 Controle de nível

O controle de nível de líquidos e de fluxo entre tanques são problemas recorrentes na indústria (NAIR; MARY; LINSELY, 2017). Esse tipo de controle está presente em diversas áreas, como por exemplo no tratamento de água, na indústria química e em processos bioquímicos (BAŞÇI; DERDIYOK, 2016). Em processos que utilizam tanques sequenciais, esse controle torna-se ainda mais importante.

Diversos processos industriais utilizam tanques de líquidos em série. Por esse motivo, o controle de nível tem sua importância intensificada. Esse controle é um dos responsáveis por manter um balanço de massa estável (RAMOS; WENSE, 2008), visto que a vazão de saída do primeiro tanque influencia a altura de líquido do segundo, consequentemente alterando a vazão do mesmo (NAIR; MARY; LINSELY, 2017). Essas interações entre os tanques ocasionam perturbações no sistema.

A taxa de mudança de fluidos de um tanque para outro precisa ser feita de maneira suave (SBARBARO; ORTEGA, 2007). Perturbações tendem a se intensificar à medida que o sistema se torna mais complexo. Tais perturbações influenciam toda a linha do processo, podendo ocasionar acidentes ou alterar a qualidade dos produtos.

O controle de nível afeta diretamente a qualidade dos produtos e a segurança dos equipamentos (BAŞÇI; DERDIYOK, 2016). Além de realizar a manutenção do balanço de massa do processo, evitar a cavitação de bombas e o transbordamento de tanques, um controle eficiente reduz o tempo de planta parada, consequentemente aumentando a eficiência do processo. Esses fatores tornam necessária a implementação de um sistema de controle.

3.3 Sistemas de controle

Um sistema de controle é constituído basicamente de três componentes, sendo eles: sensor, controlador e o elemento de controle final (JOVIC, 1986). O sensor é responsável por medir a variável de interesse do processo e transmiti-la para um controlador.

O controlador serve como mediador entre o sensor e o elemento final e normalmente é formado por um elemento de processamento, com entradas analógicas e saídas digitais. O controlador recebe esse sinal e realiza a tomada de decisão, baseando-se em comparações entre o valor medido pelo sensor e o valor desejado (SMITH; CORRIPIO, 2008).

A comunicação com o elemento de controle final é realizada com um sinal digital variável, chamado de modulação por largura de pulso (PWM - *Pulse Width Modulation*). O sinal PWM é completamente digital desde sua fonte até seu destino, resultando em uma melhor imunidade a ruídos (BARR, 2001).

Ao se combinar de maneira adequada estes três principais elementos, é possível implementar a estratégia de controle automático por realimentação.

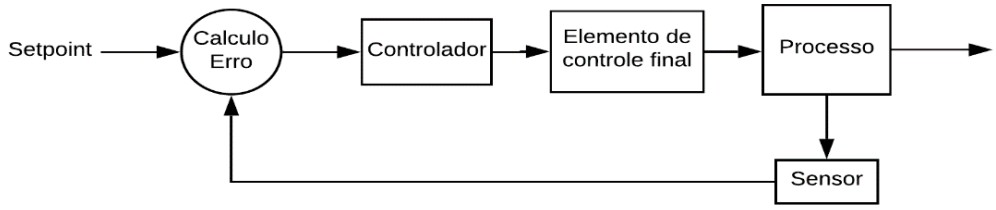
3.4 Controle por realimentação

O sistema de controle por realimentação (*feedback*) consiste em controlar uma variável do processo, a qual deverá apresentar valor igual ao *setpoint* (SP); a diferença entre essas medidas resulta em uma variável chamada de erro (E) (BISTA, 2016). O controlador só atua quando há um desvio da variável medida na saída do processo em relação ao *setpoint* (JUNIOR, 2010).

A Figura 1 representa um sistema típico de controle por realimentação. Ao medir a variável do processo, o sensor envia o sinal que será comparado com o *setpoint*, gerando um sinal proporcional ao erro calculado. O controlador recebe este sinal e emite um sinal de saída ao elemento de controle final, que atuará no processo com o objetivo de eliminar o erro observado entre variável controlada e *setpoint* (BISTA, 2016).

Portanto, o objetivo de um controle por realimentação é manipular o elemento de controle final, para que o erro seja o menor possível, e assim o sistema permaneça estável e operando dentro das especificações de projeto.

Figura 1- Diagrama de blocos



Fonte: Autoria Própria (2020)

3.4.1 Controle Regulador e Servo

O controle regulador se aplica em processos em que a variável se desvia do ponto fixo por causa de distúrbios, nesse caso o controlador tem como objetivo a manutenção da variável de entrada para evitar desvios no ponto fixo. Esse tipo de problema é mais comum nas indústrias de processos (SMITH; CORRIPIO, 2008).

O controlador Servo se aplica a sistemas onde a variação do *setpoint* é prevista como parte do processo. O controlador tem como objetivo alterar o sistema em direção ao novo *setpoint*, estabelecendo ou uma nova condição operacional desejada, ou seguindo instruções de alcance de valores de uma variável que faz parte da operação do equipamento e que deve variar para que o processo seja cumprido.

3.4.2 Ação Proporcional De Controle (P)

O controle proporcional é a ação mais simples aplicada a sistemas de controle em geral (SILVA, 2014). Sua equação de projeto é apresentada pela Eq. 1, sendo u(t) a variável de saída do controlador, E(t) o erro e K_p a constante proporcional:

$$u(t) = K_p E(t) \quad (1)$$

Quanto maior o erro, maior é a resposta do controlador proporcional. Esse controle possibilita alcançar o *setpoint* rapidamente, porém pode ocasionar um erro residual (*offset)* entre o valor atingido pela variável controlada, após a ação de controle, e o *setpoint* (RODRIGUES, 2011).

3.4.3 Ação Integral De Controle (I)

A ação integral leva em consideração todo o erro presente no sistema, desde o ponto inicial até o momento específico do processo (BISTA, 2016). A influência integral é proporcional à variação de tempo do erro (RODRIGUES, 2011).

A ação do controle integral é definida pela Eq. 2, sendo u(t) a variável de saída do controlador, E(τ) o erro e K_i a constante integral:

$$u(t) = K_i \int_o^t E(\tau) d\tau \quad (2)$$

Esse tipo de ação permite que o erro em regime permanente seja diminuído o máximo possível (SILVA, 2014).

3.4.4 Ação Derivativa De Controle (D)

A ação derivativa é proporcional à taxa de variação do erro e permite estimar o erro futuro, assim aumentando ou diminuindo a velocidade de correção (RODRIGUES, 2011).

A ação do controle derivativo é definida pela Eq. 3, sendo u(t) a variável de saída do controlador, E(t) o erro e K_d a constante derivativa:

$$u(t) = K_d \frac{dE(t)}{dt} \quad (3)$$

Um dos grandes problemas atribuídos à ação de controle derivativa refere-se a ruídos de alta frequência (SILVA, 2014). Devido ao fato de trabalhar com a antecipação do erro do sistema, o controlador pode tomar uma ação precipitada em virtude do ruído de leitura.

3.4.5 Controle PID

Este controlador envolve todas as características descritas anteriormente, possibilitando superar as desvantagens dos efeitos separados. Funcionando de forma conjunta contribui, assim, com o aumento de desempenho, visto que a ação

integrativa elimina o erro no regime permanente e a derivativa antecipa o erro (SILVA, 2014).

Esse tipo de controle é largamente utilizado na indústria por possibilitar um controle com boa estabilidade e baixo erro de *offset* (GUERRA, 2009).

A ação do controle proporcional-integral-derivativo é definida pela Eq. 4, sendo u(t) a variável de saída do controlador, E(t) o erro, K_p a constante proporcional, K_i a constante integral e K_d a constante derivativa:

$$u(t) = K_p E(t) + K_i \int_o^t E(\tau) d\tau + K_d \frac{dE(t)}{dt} \quad (4)$$

Uma das opções de implementação de controle automático que possui baixo custo é o uso da plataforma Arduino para a compilação dos cálculos de controle. Além do baixo custo com montagem e manutenção, é uma opção prática, flexível e de fácil manipulação (MIR; SWARNALATHA, 2018).

3.5 Placa Arduino

O Arduino é uma placa de *hardware* livre, ideal para a criação e prototipagem de dispositivos que permitem a interação com o usuário/ambiente. Através do *software* livre disponibilizado para a programação da placa, com uma linguagem baseada em C/C++, o Arduino dispensa o uso de programadores para o chip, facilitando ainda mais a utilização da placa (DE SOUZA et al., 2011).

A Figura 2 é a placa Arduino mais comum para prototipagem, o Arduino UNO, devido ao seu baixo custo e sua facilidade de utilização. De acordo com a imagem, é possível verificar que todas as portas da placa são conexões específicas para *jumpers*[4], assim facilitando a prototipagem. Em destaque na Figura 2 estão as portas digitais e analógicas, utilizadas para leitura dos sensores e controle dos atuadores ligados ao Arduino.

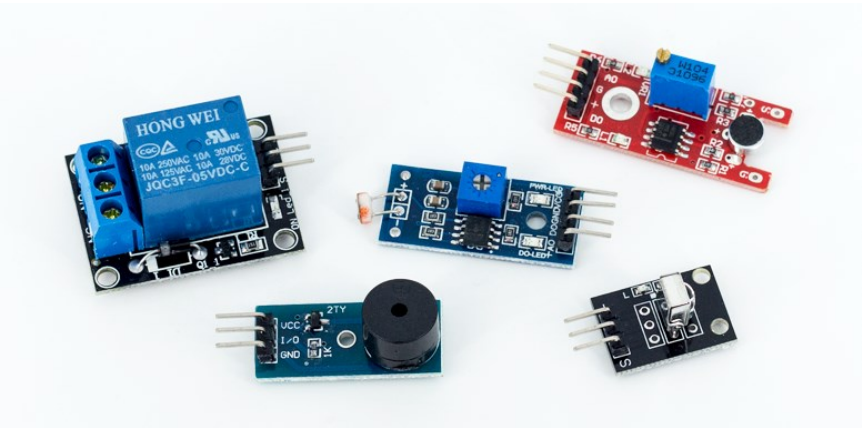
[4] Pequeno condutor utilizado para conectar dois pontos de um circuito eletrônico.

Figura 2 - Arduino UNO

Fonte: Embarcados (2020)

O Arduino conta com muitos sensores adaptados para facilitar ainda mais a criação de novos dispositivos, como demonstra a Figura 3. Comumente chamados de módulos sensores, esses dispositivos têm as conexões já pensadas para serem utilizadas com *jumpers*, facilitando a prototipagem. Dentre esses módulos estão: o módulo de distância, sensor de temperatura, sensor de umidade, tela LCD, sensor infravermelho, módulo relé, sensor de gás etc.

Figura 3 - Módulos diversos para Arduino.

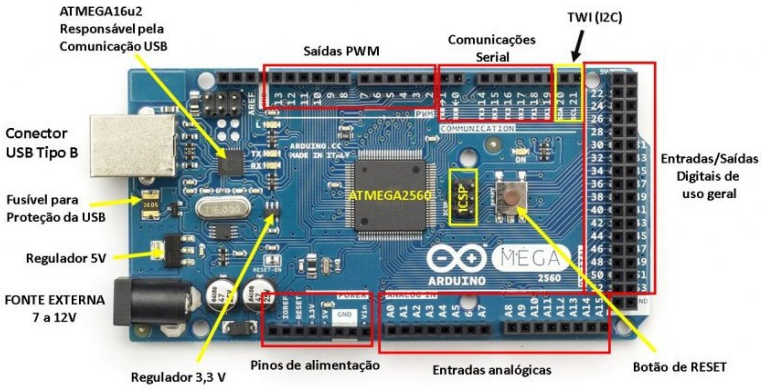


Fonte: Filipeflop (2014)

Diversos modelos da placa Arduino estão disponíveis no mercado, variando de acordo com a necessidade. Dentre as variações disponíveis observa-se: quantidade de portas digitais/analógicas, capacidade de processamento, quantidade de memória disponível e tamanho da placa. Como por exemplo, o Arduino MEGA 2560, Figura 4, possui 54 pinos de entradas e saídas digitais. Desses pinos, 14 podem

ser usados como PWM. Já o Arduino UNO possui 14 pinos de entradas e saídas digitais e, desses 14 pinos, 6 podem ser usados como PWM (SOUZA, 2020).

Figura 4 - Arduino MEGA



Fonte: Embarcados (2020)

3.6 Python

O Python é uma linguagem de programação que possui como objetivo principal a criação de algoritmos eficientes e produtivos. Em comparação a linguagens como o C/C++ e Java, o algoritmo escrito em Python é, normalmente, de duas a dez vezes mais curto, resultando em um menor tempo de escrita (MUELLER, 2020).

Guido Van Rossum, programador e pesquisador do Instituto Nacional de Matemática e Ciência da Holanda, foi o criador da linguagem Python, feito realizado no final da década de 80 (COUTINHO, 2019).

O Python possui diversas facilidades, além da sintaxe simples. Diferente das outras linguagens, toda linha de Python não precisa de um símbolo para declarar o final da linha (em outras linguagens, como por exemplo no C/C++, é comum erros de compilação pela simples falta de um “”;” no final de alguma linha). Outra facilidade de Python é o fato de não precisar declarar o tipo da variável, tornando o processo de escrita muito mais rápido e eficiente. Como o código se torna mais simples, a legibilidade aumenta muito, facilitando a implementação futura de funções.

4 METODOLOGIA

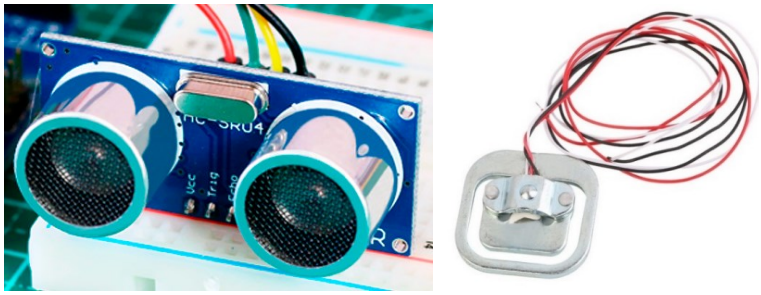
A construção do módulo foi dividida em três etapas principais: o estudo teórico da estratégia de controle a ser implementada, a montagem do módulo e o desenvolvimento dos códigos. Essas etapas foram ser realizadas de maneira simultânea, desenvolvendo parcialmente os tópicos independentes. O passo inicial foi o estudo teórico.

Após estabelecer os aspectos teóricos do controle de nível do módulo, foi feito o projeto do módulo experimental. Foram utilizados dois tanques retangulares, alocados um sobre o outro verticalmente. O tanque superior possui medidas de 20x20 cm de base e as quatro laterais com 20x30 cm, esse é o tanque onde o controle de nível será realizado. O tanque inferior foi responsável apenas por armazenar a água. O módulo possui um sistema de encanamento com 20 mm de diâmetro. A vazão de saída do tanque de cima foi controlada com um registro esfera de 20 mm, e a vazão de entrada aconteceu por meio de uma bomba 12v, 22W com vazão de 800 L/h, simulando um processo industrial. O registro atuou na perturbação do sistema, simulando uma variação na vazão de saída. Dessa maneira, o sistema foi capaz de reagir de acordo com o controlador utilizado para realizar o ajuste de nível de líquido no tanque.

O controle foi feito por meio de um microcontrolador Arduino, com dois sinais de medida de nível, um sinal realizado pelo sensor ultrassônico HC-SR04 (Figura 5A) e a outra leitura de nível foi feita com uma célula de carga (Figura 5B).

Figura 5 - Módulos utilizados na leitura de nível

(a) Sensor ultrassônico HC-SR04 **(b) Célula de carga de 50 kg**



Fonte: Filipeflop (2011), figura 6a, Filipeflop (2020), figura 6b

O sensor ultrassônico foi escolhido por ser uma opção simples e barata, porém este sensor não é muito preciso. Ambos os sensores realizaram a leitura do nível do tanque de maneira indireta; o sensor ultrassônico realizou uma leitura da distância atual até o final da coluna de água. Por esse motivo foi necessário informar a área da base do tanque e qual era a distância do fundo do tanque até o sensor, para que o cálculo da quantidade de líquido no tanque pudesse ser feito. O funcionamento da balança (conjunto das células de carga com o amplificador) também precisou de parâmetros, como a massa do tanque vazio e a densidade do líquido. No caso de líquidos com a densidade desconhecida, o sensor de distância realizou uma leitura de volume e aproximou uma densidade com medições de massa. Todos esses parâmetros foram enviados a um programa desenvolvido em Python.

O programa escrito em Python possui uma interface gráfica para que o usuário possa visualizar o gráfico de variação de nível e enviar os parâmetros para o módulo, como as constantes do controlador, área da base do tanque, modo de operação, realização da estimativa de densidade, entre outros. Através do *software* foi possível obter os dados das leituras realizadas pelo módulo. A comunicação com o módulo foi feita por meio de uma porta serial, com a qual o Python se comunicou com o programa gravado no Arduino.

O programa feito para o Arduino foi responsável por receber os parâmetros e realizar os ajustes necessários para o controle do módulo de nível. O controle da bomba foi feito por meio da equação do controlador, configurada pelo usuário. A leitura dos sensores e a conversão dos valores lidos para valores de nível aconteceram nessa etapa. O programa envia o valor do nível para o algoritmo escrito em Python construir o gráfico do comportamento do nível de líquido no tanque em função do tempo. Os parâmetros do processo ficaram guardados na memória EEPROM[5] do Arduino.

Para se obter os parâmetros K_p, K_i e K_d de maneira experimental, foi utilizado o método da malha aberta de Ziegler-Nichols. Esse método é baseado em causar uma perturbação no elemento final de controle (no caso do presente trabalho este elemento é a potência da bomba) e permitir que essa perturbação se propague pelo processo sem que haja atuação do controlador na correção dessa perturbação. Por isso o

[5] EEPROM, sigla do inglês *Electrically-Erasable Programmable Read-Only Memory,* é uma memória não volátil, ou seja, não é zerada após o desligamento.

método é chamado de malha aberta, uma vez que não há o fechamento da transmissão de sinal para completar a ideia de realimentação (feeback). A resposta do sistema é então coletada e o gráfico resultante traz consigo toda a dinâmica presente no elemento final de controle, processo e sensores de medida. Com esse gráfico (Figura 6) é possível obter os parâmetros necessários para a sintonização usando a tabela de Ziegler-Nichols (Tabela 1) do método da malha aberta.

Tabela 1 - Fórmulas de sintonização para resposta de razão de decaimento de um quarto

Controlador	Kp	T_i	T_d
P	$\frac{1}{K} * \left(\frac{t_0}{\tau}\right)^{-1}$	-	-
PI	$\frac{0,9}{K} * \left(\frac{t_0}{\tau}\right)^{-1}$	$3,33 * t_0$	-
PID	$\frac{1,2}{K} * \left(\frac{t_0}{\tau}\right)^{-1}$	$2,0 * t_0$	$\frac{t_0}{2}$

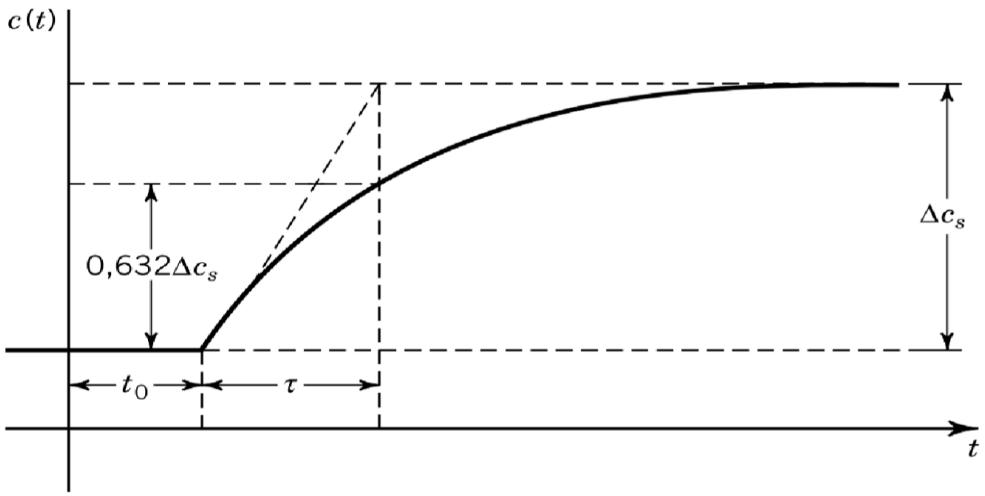
Fonte: SMITH, CORRIPIO (2008, p.219)

Considerando K_p como o ganho proporcional, T_i o tempo integral e T_d o tempo derivativo. O valor de K é obtido com a Equação 6, sendo ΔC_s o intervalo da variável observada e Δm o degrau da fonte de mudança da perturbação:

$$K = \frac{\Delta C_s}{\Delta m} \tag{6}$$

Os valores de t_0 e T são obtidos através do gráfico da variável medida pelo tempo (Figura 15).

Figura 6 - Análise do gráfico para obtenção dos parâmetros



Fonte: SMITH, CORRIPIO (2008, p.215)

4.1 Hardware

Para a montagem do hardware foram utilizados os componentes presentes no Quadro 1.

Quadro 1 - Hardware do módulo

Componente	Descrição
Arduino Nano	Microcontrolador responsável pelo controle do módulo.
Sensor HC-SR04	Sensor ultrassônico responsável pela medição da altura da coluna de líquido no tanque.
Módulo Hx711	Módulo responsável pela amplificação do sinal de leitura das células de carga.
Células de carga	Componente responsável pela medição de massa do tanque.
Transistor TIP120	Responsável pelo controle de rotação da bomba.

Fonte: Autoria Própria (2020)

O Arduino Nano foi escolhido para esse projeto pelo baixo custo apresentado e pelo tamanho compacto, facilitando a montagem da placa ilhada. O Arduino Nano possui as mesmas configurações de hardware que o Arduino UNO, porém com tamanho reduzido.

O hardware do módulo foi montado pensando em modificações futuras, o Arduino Nano foi colocado em uma base removível para facilitar a substituição. As células de carga foram conectadas com jumpers assim como o sensor HC-SR04. O único componente que foi soldado direto na placa é o módulo Hx711 e o transistor TIP120, os demais componentes são de fácil remoção para facilitar alterações futuras no módulo.

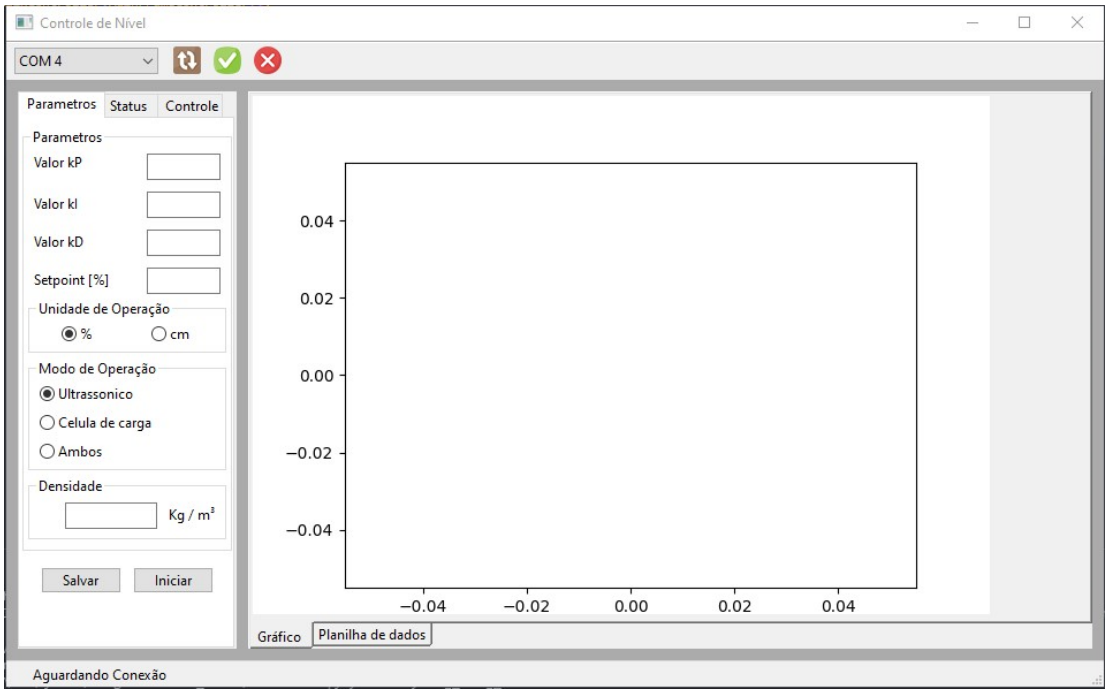
4.2 Software

O *software* foi construído utilizando a linguagem Python, a interface gráfica (Figura 6) foi construída com a biblioteca wxPython. Com o software foi possível visualizar os dados de forma gráfica ou em forma de tabela, com a possibilidade de

exportar os dados para estudos posteriores. A comunicação serial com o Arduino foi feita com um cabo USB-A/Mini USB. O código consta no Apêndice B.

O *software* realiza a comunicação com o microcontrolador, ou seja, é o responsável por enviar/receber informações de uma maneira intuitiva para o usuário. Além de enviar todos os parâmetros necessários para o funcionamento do módulo, através dele é possível configurar o zero do tanque para a medição de altura de líquido, e a tara do tanque para a medição de massa, também é possível estimar a densidade do líquido no tanque através da relação entre volume e massa.

Figura 7 - Interface gráfica



Fonte: Autoria Própria (2021)

4.3 Microcontrolador

O código do microcontrolador foi desenvolvido em C++, através da IDE Arduino. O microcontrolador conta com uma biblioteca para realizar o cálculo do PID. Todos os parâmetros do controlador ficam salvos na EEPROM do embarcado, dessa maneira, toda a configuração feita no dispositivo fica salva na memória para a próxima utilização. O código consta no Apêndice A.

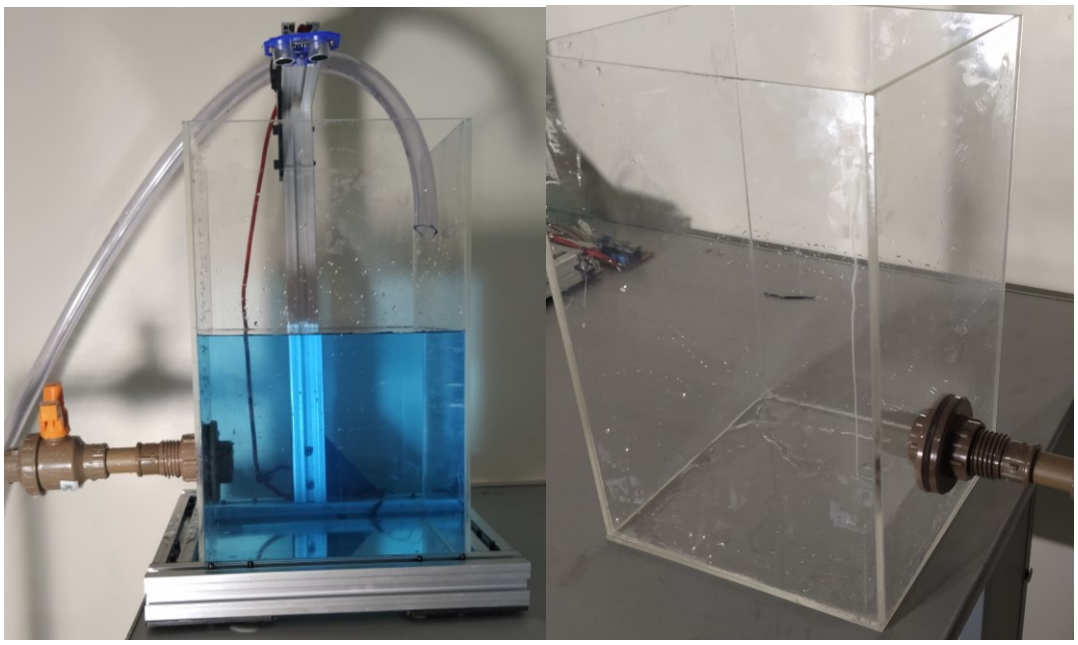
4.4 Estrutura

A estrutura é constituída por três partes principais: tanque superior, estrutura de alumínio e tanque inferior.

O tanque superior foi construído utilizando acrílico. A princípio a ideia era cortar todas as peças do tanque na CNC do FabLab da UTFPR, porém vários imprevistos surgiram na tentativa de realizar o corte. O principal problema foi a perda de passo da CNC, o que ocasionou irregularidades na lateral das peças, o que posteriormente poderia ocasionar a má fixação da cola, impedindo a montagem do tanque. A solução foi realizar o corte das peças em uma CNC laser, serviço prestado por uma empresa localizada em Francisco Beltrão - PR.

O tanque (Figura 8) possui uma abertura circular para o sistema de encanamento. O sistema conta com um flange para realizar a conexão do tanque com os canos, uma válvula para realizar a perturbação e canos para realizar o direcionamento do líquido para o tanque inferior.

Figura 8 - Tanque superior

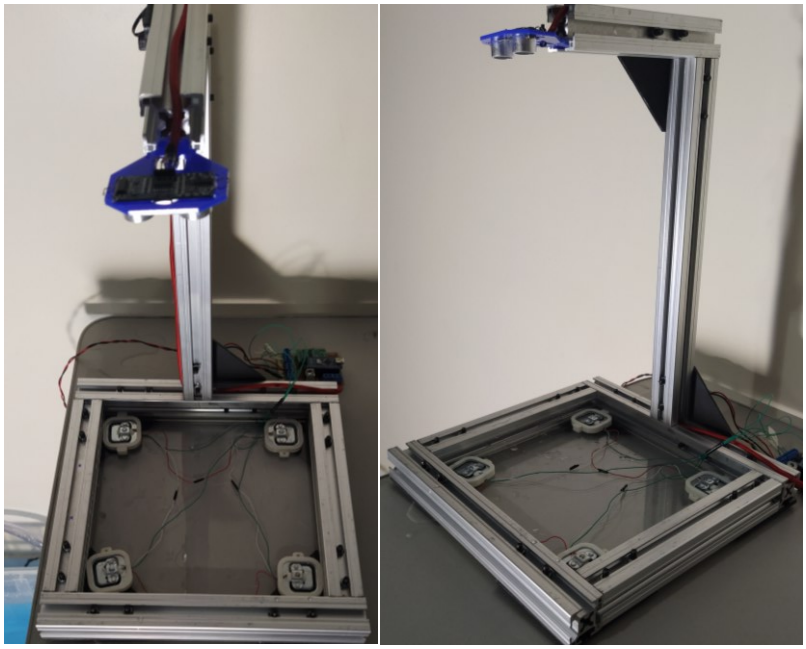


Fonte: Autoria Própria (2021)

A estrutura que suporta o tanque superior, Figura 9, foi construída no FabLab da UTFPR, utilizando alumínio estrutural. A estrutura serve como suporte para o sensor ultrassônico e possibilita que o tanque permaneça em cima das células de

carga. Foi projetada para ser de fácil transporte, possibilitando assim o deslocamento do módulo para uma sala de aula, caso seja do interesse do usuário.

Figura 9 - Estrutura de apoio



Fonte: Autoria Própria (2021)

O tanque inferior, representado na Figura 10, foi feito utilizando uma caixa organizadora de 26 Litros. Essa opção foi escolhida por ser de baixo custo em relação a montagem de outro tanque de acrílico. O tanque inferior serviu apenas como reservatório, permitindo o fluxo circular de líquido no sistema.

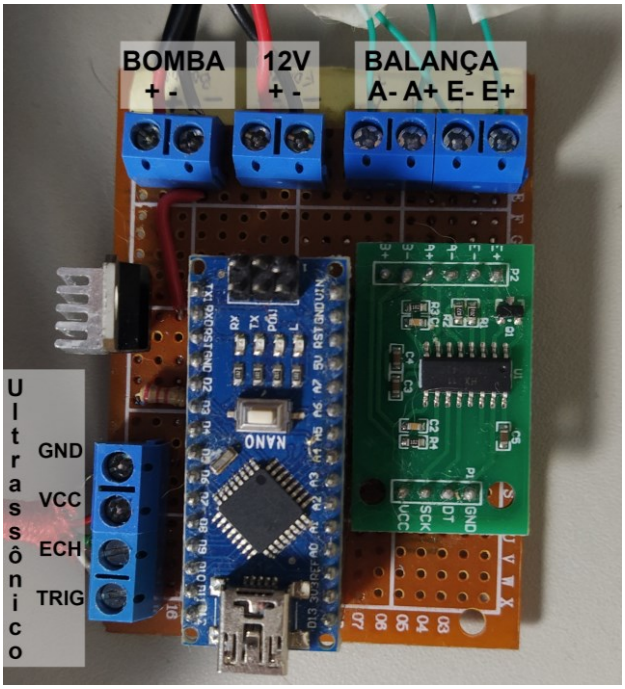
Figura 10 - Tanque inferior

Fonte: Autoria Própria (2021)

5 RESULTADOS E DISCUSSÕES

A estrutura elétrica foi feita utilizando uma placa ilhada para facilitar a montagem do circuito. O Arduino Nano foi instalado em conectores, com intenção de evitar a soldagem na placa para realizar a troca do componente caso necessário, ou para facilitar a manutenção do código. A placa ilhada possui conectores para todas as entradas e saídas, facilitando a montagem do módulo. Na Figura 11 é possível observar o resultado da montagem do circuito elétrico.

Figura 11 - Análise do gráfico para obtenção dos parâmetros



Fonte: Autoria Própria (2021)

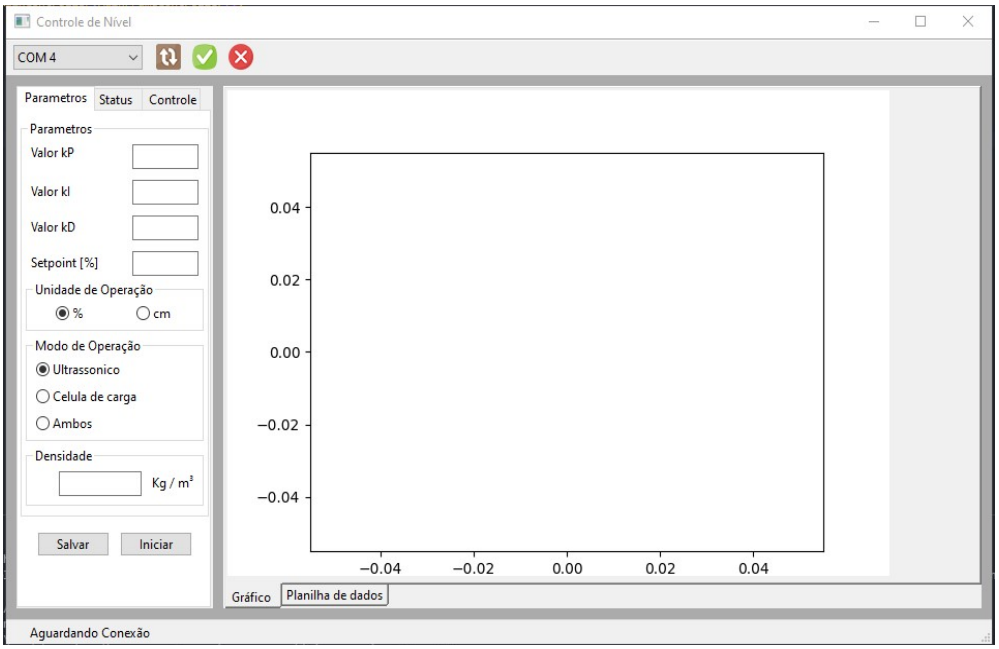
O código desenvolvido em Python serviu como uma interface para a utilização das funções do microcontrolador, sendo capaz de: ler e enviar comandos via porta serial, visualizar os resultados graficamente, salvar os dados em uma planilha interna, sendo possível copiar os dados e exportá-los posteriormente para análise em um software de planilhas.

O código em Python utiliza a biblioteca wxPython para realizar a construção e controle da interface gráfica. A intenção da interface foi de facilitar o uso do módulo, tornando-o intuitivo, de fácil utilização.

A comunicação foi feita através da porta serial, com *baudrate* [6] de 9600, essa velocidade foi escolhida por ser suficiente para o envio dos dados, evitando o engarrafamento de dados da interface com o microcontrolador.

A Figura 12 representa a tela inicial do *software*, a configuração da comunicação serial com o módulo fica na barra superior no canto esquerdo.

Figura 12 - Interface gráfica na aba "Parâmetros"



Fonte: Autoria Própria (2021)

Da esquerda para a direita: caixa de seleção da porta serial, botão para recarregar portas serial, botão para conectar na porta selecionada e botão para fechar conexão serial.

Na tela inicial do programa, na direita, fica localizado o gráfico, abaixo do gráfico fica o elemento responsável por alternar a visualização entre gráfico ou planilha de dados. O gráfico pode ser visto durante a reprodução do experimento, a planilha fica inutilizada nesse momento, pois a adição dos dados resulta no recarregamento da planilha, impedindo a movimentação pelos dados.

Na esquerda do layout, fica localizado a seção de abas responsáveis pelo controle do módulo. A aba "Parâmetros" contém os campos necessários para realizar o envio dos parâmetros ao microcontrolador, como os valores de Kp, Ki e Kd, assim como o *setpoint,* unidade de operação (porcentagem ou altura de líquido) e densidade.

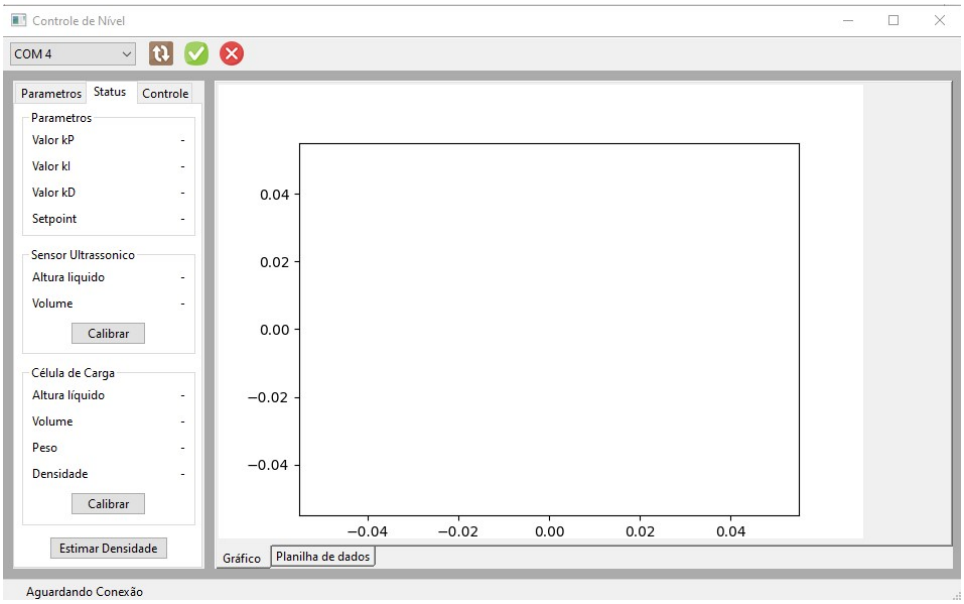
---

[6] Baud rate é um termo utilizado para medir a velocidade entre a comunicação entre dispositivos que utilizam a comunicação serial.

O botão “Salvar” envia os dados dos campos para o módulo e o botão “Iniciar” inicia o experimento, utilizando os parâmetros contidos no módulo (é utilizar o botão “Salvar” caso o desejo seja iniciar o experimento com os parâmetros definidos na interface).

A aba “Status”, Figura 13, apresenta todos os parâmetros contidos no módulo, a fim de confirmar se todos os parâmetros desejados constam na rotina do módulo. Na aba “Status” ficam contidos os botões responsáveis por realizar a calibração do tanque, o botão “Calibrar” localizado na seção “Sensor Ultrassônico” realizar a calibração da altura do tanque, obtendo a leitura de distância do sensor até a altura do líquido, configurando como zero do tanque. Dessa maneira, toda leitura do sensor ultrassônico irá descontar a leitura feita na calibração, a fim de obter a leitura da altura da coluna de líquido.

Figura 13 - Interface gráfica na aba “Status”



Fonte: Autoria Própria (2021)

O botão “Calibrar” na seção “Célula de Carga” realiza o mesmo procedimento, porém utilizando medidas de massa. A intenção é obter a medida de massa e utilizar como tara da balança, para medir apenas a massa da coluna de água. A conversão de massa para altura de líquido é feita através da relação entre fórmulas físicas.

Com a densidade (Equação 5) é possível obter o volume de líquido no tanque.

$$D = \frac{m}{V} \tag{5}$$

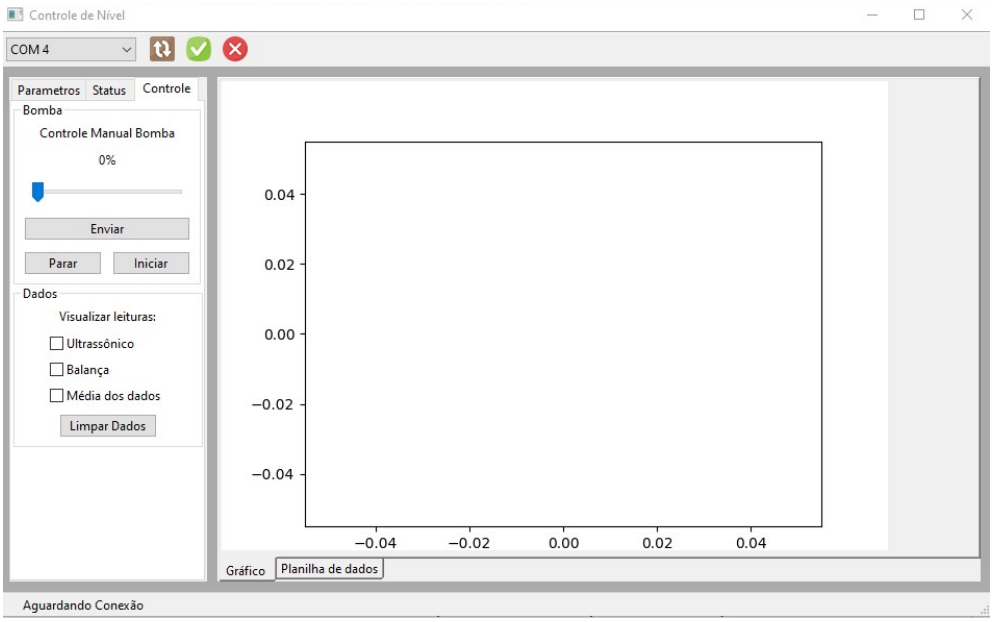
O formato retangular do tanque permite encontrar a altura facilmente com a fórmula do volume do retângulo (Equação 6).

$$V = \text{Área da base} * \text{Altura} \quad (6)$$

O botão “Estimar Densidade” localizado na aba “Status” envia um comando para o módulo responsável por obter uma estimativa da densidade. O volume do líquido é estimado utilizando a Equação 6 com a leitura de altura de líquido obtido pelo sensor ultrassônico. A estimativa da densidade é obtida utilizando a Equação 5 com a leitura de massa da balança.

A aba “Controle”, Figura 14, realiza tanto o controle da bomba como o da visualização dos dados. O slide é responsável por selecionar a potência da bomba, o botão “Enviar” envia a potência da bomba para o microcontrolador. O botão “Iniciar” liga a bomba e o “Parar” desliga a bomba. Na seção “Dados” é possível selecionar quais dados serão exibidos no gráfico através das *checkbox*. O botão “Limpar Dados” apaga os dados de leitura do experimento.

Figura 14 - Interface gráfica na aba “Controle”



Fonte: Autoria Própria (2021)

O código para o microcontrolador foi o responsável por conter todas as rotinas e funções necessárias para o funcionamento do módulo, como por exemplo: cálculo da altura de nível através da diferença entre altura do tanque e leitura do sensor ultrassônico, cálculo do nível de líquido através de parâmetros de peso e densidade, função para estimar a densidade do líquido utilizando a leitura da balança em conjunto com a leitura de volume do sensor ultrassônico, dentre outras funções.

O microcontrolador realiza o cálculo do PID através da biblioteca “PID”, facilitando a implementação desse controlador em dispositivos embarcados. Sendo necessário citar apenas os parâmetros do controlador, *setpoint,* estilo de comportamento do PID (normal ou inverso) e configurar a saída do atuador responsável por modificar a variável.

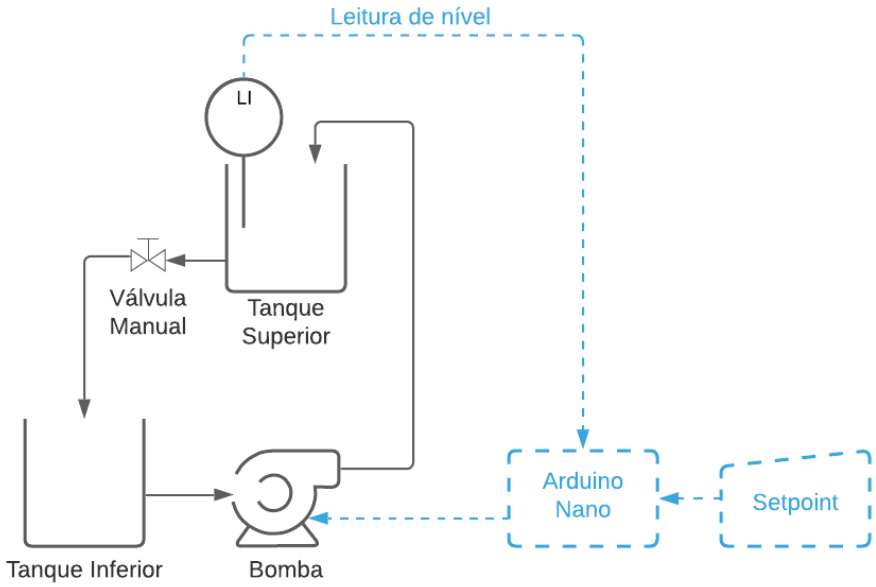
Através da Figura 15 é possível visualizar o resultado da construção do módulo didático de controle de nível e o fluxo de funcionamento.

Figura 15 - Análise do gráfico para obtenção dos parâmetros

Fonte: Autoria Própria (2021)

A Figura 16 representa um esquema do funcionamento do módulo. Basicamente o tanque superior possui uma vazão de entrada e uma vazão de saída, a vazão de saída é regulada pela válvula, responsável por realizar a perturbação no sistema. Todo o líquido que sai do tanque superior é destinado ao tanque inferior, cujo papel é apenas de servir como reservatório para manter o sistema cíclico. A bomba é conectada no tanque inferior e controlada pelo sistema do microcontrolador, que é regulado com os parâmetros escolhidos pelo usuário. A bomba envia o líquido para o tanque superior, completando o ciclo.

Figura 16 - Esquema de funcionamento do módulo



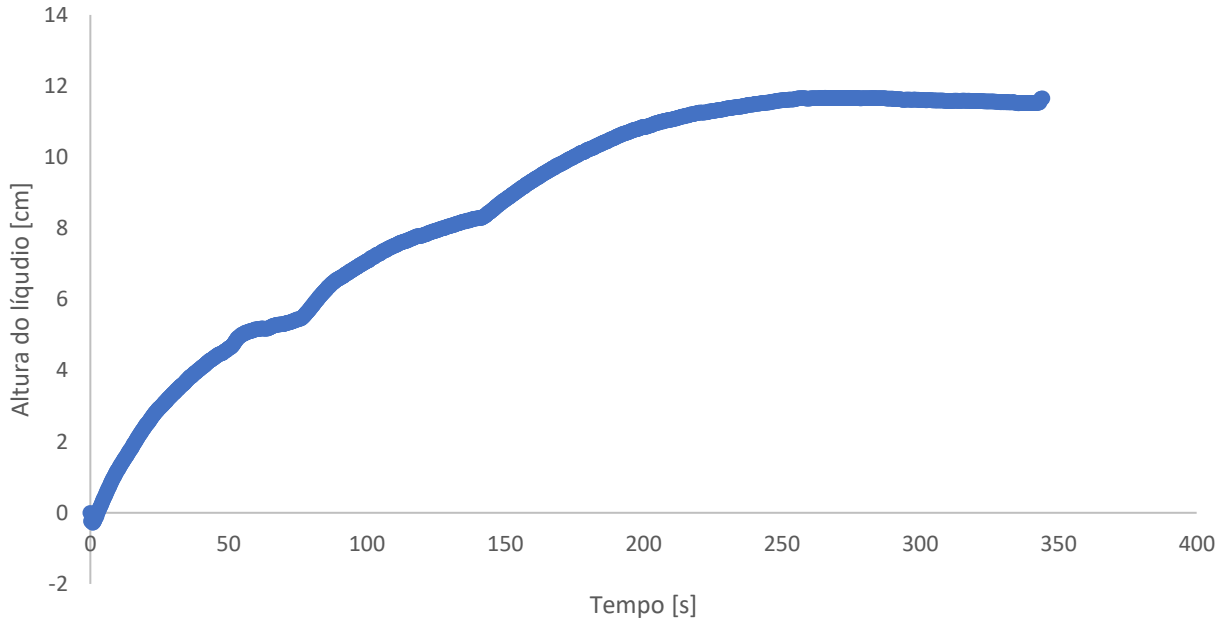
Fonte: Autoria Própria (2021)

O custo total estimado para a produção do módulo foi de aproximadamente R$ 750,00, custo 65 vezes menor em comparação com o modelo já citado nesse trabalho, de valor R$ 49.671,10. No caso desse trabalho, muito dos materiais foram reutilizados de projetos antigos do FabLab, como o alumínio estrutural e as placas de acrílico, reduzindo ainda mais o custo do módulo para aproximadamente R$ 300,00.

O módulo opera de duas maneiras, realizando o controle para problemas servo e regulador. O problema servo consiste em uma alteração do *setpoint* atual, fazendo com que o módulo altere a vazão da bomba para que o nível se desloque para o novo *setpoint*, definido pelo usuário. Já o problema regulador consiste em adaptar a vazão de entrada com base em perturbações da vazão de saída, fazendo com que o nível do tanque permaneça no *setpoint* configurado, com o mínimo de variação possível, eliminando o efeito da perturbação na variável controlada.

A Figura 17 representa o gráfico obtido experimentalmente, os dados foram obtidos obtendo a média entre as leituras do sensor ultrassônico e as leituras da balança, utilizando uma vazão constante de entrada e saída, a fim de elevar a altura de líquido em um nível onde a vazão de saída fosse compensada pela vazão de entrada.

Figura 17 - Gráfico experimental para obtenção dos parâmetros



Fonte: Autoria Própria (2021)

A Tabela 2 representa os resultados obtidos experimentalmente.

Tabela 2 - Parametros obtidos experimentalmente para sintonia PID

Parâmetro	Equação	Valor
t_0	-	10
T	-	90
K	$\frac{\Delta C_s}{\Delta m}$	2,4
Kp	$\frac{1,2}{K} * \left(\frac{t_0}{\tau}\right)^{-1}$	4,5
Ki	$2,0 * t_0$	0,225
Kd	$\frac{t_0}{2}$	22,5

Fonte: Autoria Própria (2021)

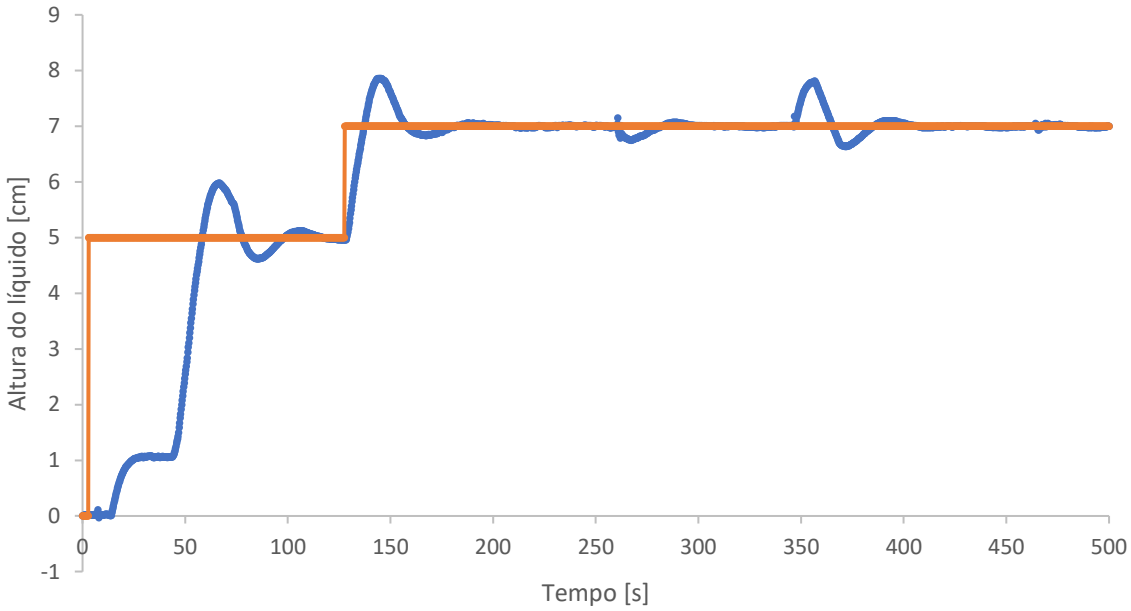
Os parâmetros sintonizados foram testados, porém sem sucesso. O sistema não respondeu corretamente. Por esse motivo, a sintonia manual foi escolhida para ser utilizar no sistema. Embora os valores obtidos não tenham funcionado, eles serviram de base de ordem de grandeza para a escolha dos valores que funcionariam para o controle do módulo de nível.

Uma das prováveis causas do insucesso da sintonia pode ter sido a consistência dimensional da malha, uma vez que não foi estudada uma malha teórica previamente. Outro possível motivo pode ser relacionado a um intervalo de resposta para a bomba. O sinal PWM do sistema varia de 0 a 255, essa variação resulta em uma voltagem enviada para a bomba para operar de maneira controlada. Porém, através de experimentos, foi possível observar que a bomba só começa a responder após um sinal PWM de 100, talvez isso não tenha sido considerado no cálculo dos parâmetros. Obtendo parâmetros que resultam em valores PWM menor que 100, impedindo o funcionamento do sistema.

Os parâmetros foram obtidos de maneira manual. Essa obtenção de parâmetros foi feita por meio de testes. O objetivo foi encontrar parâmetros que realizassem a manutenção da variável em um intervalo de tempo aceitável. O sistema iniciou com os parâmetros zerados, o termo K_p foi sendo elevado de maneira gradual, até obter a resposta do sistema. Assim, o sistema começou a responder e realizar a manutenção do nível para o *setpoint*. O parâmetro K_i foi estimado de maneira gradual, verificando o intervalo de tempo necessário para realizar o ajuste para o *setpoint*. O parâmetro K_d não foi estimado, pois o sistema em questão é um sistema turbulento, ou seja, possui ruídos na entrada das leituras de nível, nesse caso o parâmetro K_d é zerado para evitar erros.

Na Figura 18, é possível observar o experimento com a manutenção do sistema servo e regulador utilizando os parâmetros K_p, K_i e K_d, sintonizados manualmente. Os parâmetros sintonizados foram: $K_p = 10$, $K_i = 5$ e $K_d = 0$.

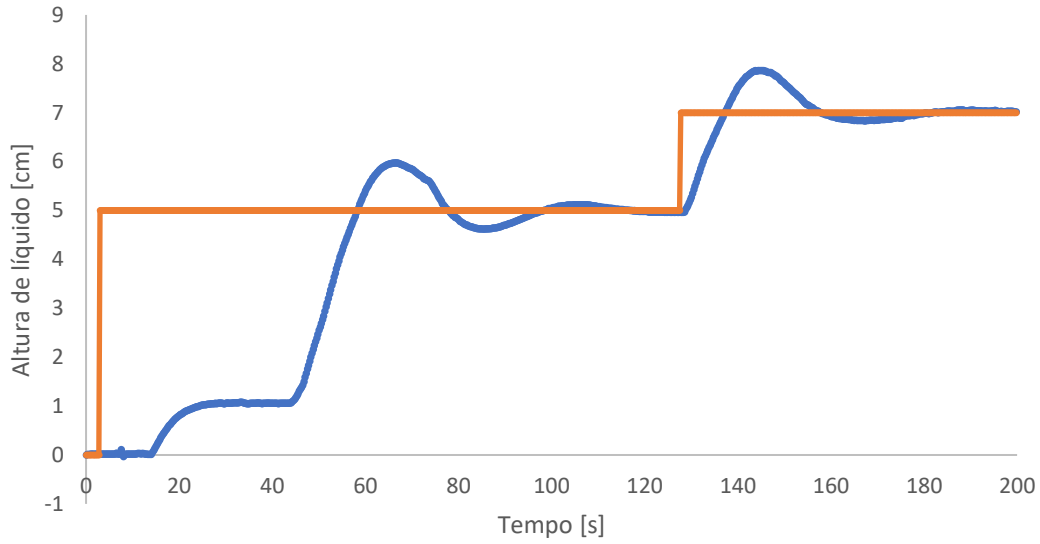
Figura 18 - Experimento realizado com o módulo



Fonte: Autoria Própria (2021)

A Figura 19 representa o intervalo de tempo em que o problema servo foi aplicado. Alternando o *setpoint* do sistema de 5 cm para 7 cm. É possível observar que o comportamento do sistema converge para o *setpoint*, apresentando comportamento típico de uma malha de realimentação respondendo a um comando do controlador. É possível visualizar a presença de *overshoot*, que é o salto dado pela variável acima do *setpoint*, caracterizado pelo aumento da vazão de entrada, na tentativa de atingir a nova altura de líquido. Esse salto atinge valor máximo próximo a 8 cm, porém, a malha de realimentação continua trabalhando, diminuindo a vazão de entrada para que a altura de 7 cm possa ser atingida, que é o objetivo estabelecido para o problema servo. Cerca de 10 segundos depois da altura de 8 cm ter sido atingida, o valor de 7 cm definido como o novo *setpoint*, é alcançado e o nível de líquido no tanque é mantido nesse novo valor

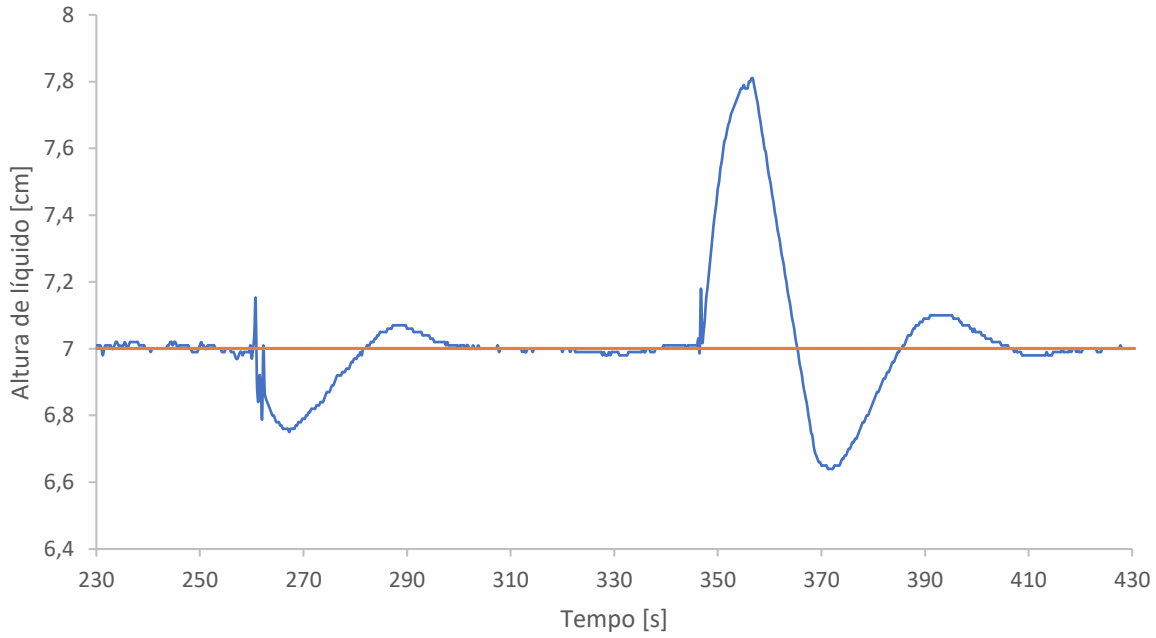
Figura 19 - Módulo reagindo ao problema servo



Fonte: Autoria Própria (2021)

A Figura 20 representa o intervalo de tempo em que o problema regulador foi aplicado. Em primeiro momento, no tempo aproximado de 250 segundos, a válvula de saída do tanque teve a vazão aumentada, resultando na queda da altura de líquido e consequentemente no sistema trabalhando para eliminar essa perturbação. No momento aproximado de 340 segundos, a válvula teve sua vazão reduzida, proporcionando ao sistema um acúmulo de líquido, até o momento da correção feita pelo controlador. Vale ressaltar que depois da primeira perturbação imposta no sistema, a altura de líquido mínima atingida foi de 6,65 cm, aproximadamente. Após alguns segundos a malha atuou no sistema fazendo com que o nível do tanque fosse reestabelecido ao valor de 7 cm. A segunda perturbação imposta foi mais intensa, fazendo com que inicialmente o tanque atingisse 7,8 cm de nível de líquido. A malha reage a essa perturbação diminuindo a vazão de entrada, fazendo com que o nível de líquido comece a cair. Em dado momento, a nova vazão de entrada é redefinida pela malha, necessitando ser aumentada, pois a altura de líquido no tanque atingiu um valor menor que 7 cm (6,6 cm aproximadamente). Alternando entre aumento e diminuição da vazão de entrada, é possível observar que depois de algum tempo a malha de realimentação reestabelece o nível de líquido do tanque para 7 cm, eliminando as perturbações que tiraram o nível de líquido de seu *setpoint*.

Figura 20 - Módulo reagindo ao problema regulador

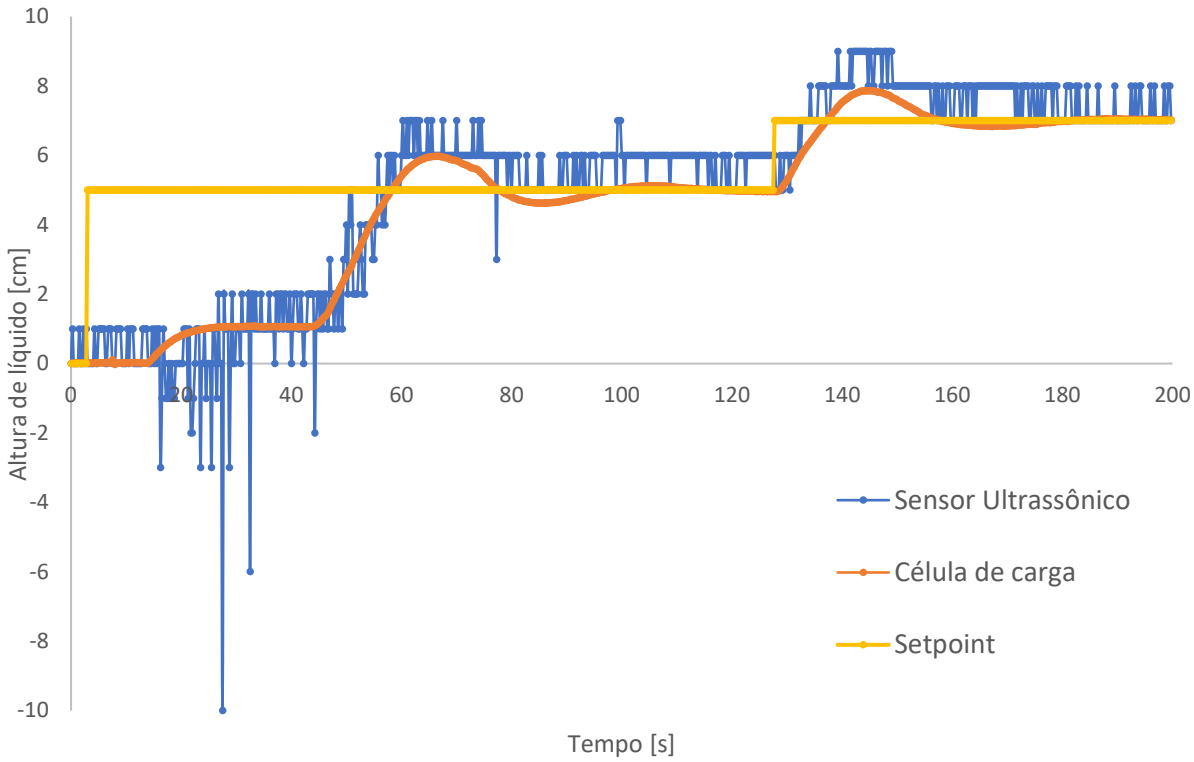


Fonte: Autoria Própria (2021)

Por meio das Figuras 18, 19 e 20 é possível observar o controlador reagindo ao problema servo e ao problema regulador utilizando os parâmetros para o controlador PID encontrados manualmente, reagindo de maneira esperada aos problemas.

A Figura 21 representa a comparação entre os sensores ultrassônico e a balança no experimento realizado. É possível observar a falta de precisão do sensor ultrassônico, além da sensibilidade de leitura ser de aproximadamente 1 cm tornando a leitura imprecisa, o sensor oscila em alguns momentos, o que pode prejudicar o ajuste de nível. Apesar da falta de precisão, o sensor ultrassônico foi utilizado como ferramenta de segurança para o módulo, evitando casos de transbordamento do tanque.

Figura 21 - Leituras dos sensores no experimento



Fonte: Autoria Própria (2021)

6 CONSIDERAÇÕES FINAIS

O código escrito em Python apresentou resultados satisfatórios, assim como o código desenvolvido para o Arduino. Ambos conseguiram realizar as rotinas previstas em código.

O módulo possibilitou a configuração dos parâmetros do PID, assim como a configuração do *setpoint.* Apesar da falta de precisão do sensor ultrassônico, o mesmo se mostrou útil como ferramenta de segurança para evitar o transbordamento do tanque.

Apesar da falha na sintonia com os parâmetros de malha aberta, o módulo, configurado com os parâmetros manuais, manteve o nível no *setpoint* desejado, permitindo a alteração do *setpoint,* assim como, eliminando variações causadas pela perturbação na válvula.

Portanto, é possível concluir que o módulo desenvolvido no presente trabalho pode ser utilizado para abordar os principais conceitos presentes na disciplina de Controle de Processos, aplicando-se a teoria em um problema de controle de nível de líquidos em tanques. O módulo pode ser utilizado para complementar experimentos na disciplina de Laboratório para Engenharia Química 3, que prevê a reprodução experimental de teorias aprendidas na disciplina de Controle de Processos.

7 RECOMENDAÇÕES PARA TRABALHOS FUTUROS

A seguir, algumas recomendações para aperfeiçoar a experiência na utilização do módulo:

- Produzir uma placa com circuito impresso para o circuito eletrônico.

- Adição de outra variável a ser controlada que seja de interesse, como por exemplo: temperatura ou vazão.

- Construir um tanque inferior de acrílico, e utilizá-lo para observar os efeitos que a modificação da altura de nível no tanque superior causa.

- Criar rotinas de experimentos para a realização das aulas práticas.

- Estudar a aplicação do método de malha aberta para a sintonia dos controladores.

REFERÊNCIAS

BARR, Michael. Introduction to Pulse Width Modulation (PWM). **Barrgroup**, 2001. Disponível em: <https://barrgroup.com/embedded-systems/how-to/pwm-pulse-width-modulation>. Acesso em: 14 de jun. de 2020.

BAŞÇI, Abdullah; DERDIYOK, Adnan. Implementation of an adaptive fuzzy compensator for coupled tank liquid level control system. **Measurement: Journal of the International Measurement Confederation**, *[S. l.]*, v. 91, p. 12–18, 2016. DOI: 10.1016/j.measurement.2016.05.026.

BIANCHINI, David; GOMES, Francisco de Salles C. A Simulação como Ferramenta Didática no Ensino de Engenharia. **XXXV Congresso Brasileiro de Educação e engenharia**, Campinas, SP, p. 1–5, 2007. Disponível em: http://www.abenge.org.br/CobengeAnteriores/2007/artigos/368-David Bianchini.pdf.

BISTA, Dinesh. Understanding and Design of an Arduino- based PID Controller. Richmond, VA, 2016.

BRASIL, MINISTÉRIO DA EDUCAÇÃO. Conselho Nacional de Educação. Resolução n° 2, de 24 de abril de 2019. Institui as Diretrizes Curriculares Nacionais do Curso de Graduação em Engenharia. Brasília, 24 abr. 2019. Disponível em: https://www.in.gov.br/web/dou/-/resolu%C3%87%C3%83o-n%C2%BA-2-de-24-de-abril-de-2019-85344528. Acesso em: 30 de jun. de 2021.

COUTINHO, Thiago. Entenda o que é Python e pra que ele serve. **Voitto**, 2019. Disponível em: <https://www.voitto.com.br/blog/artigo/python>. Acesso em: 13 de out. de 2020.

COUTO, Francisco Pazzini. ATIVIDADES EXPERIMENTAIS EM AULAS DE FÍSICA : REPERCUSSÕES NA MOTIVAÇÃO DOS ESTUDANTES , NA DIALOGIA E NOS PROCESSOS DE MODELAGEM Belo Horizonte ATIVIDADES EXPERIMENTAIS EM AULAS DE FÍSICA : REPERCUSSÕES NA MOTIVAÇÃO DOS ESTUDANTES , NA DIALOGIA E NOS. **Physics**, *[S. l.]*, 2009.

DE SOUZA, Anderson R.; PAIXÃO, Alexsander C.; UZÊDA, Diego D.; DIAS, Marco A.; DUARTE, Sergio; DE AMORIM, Helio S. A placa Arduino: uma opção de baixo custo para experiencias de física assistidas pelo PC. **Revista Brasileira de Ensino de Fisica**, Rio de Janeiro, RJ, v. 33, n. 1, p. 1–5, 2011. DOI: 10.1590/S1806-11172011000100026.

DUARTE, Bruce Paiva. Controle de temperatura de um processo fotocatalítico com uso da plataforma arduino. *[S. l.]*, 2019.

FILIPEFLOP. Sensor de Peso 50Kg Célula de Carga. **Filipeflop**, 2020. Disponível em: <https://www.filipeflop.com/produto/sensor-de-peso-50kg-celula-de-carga/#tab-description>. Acesso em: 13 de out. de 2020.

GOSMANN, HUGO LEONARDO. Um sistema multivariável de tanques acoplados para avaliação de técnicas de controle. Brasília, DF, 2002.

GUERRA, W. A. Implementação de Controle Proporcional , Integral e Derivativo

Digital em Controladores Lógico Programáveis. Recife, PE, p. 1–37, 2009.

HECK, GUILHERME DE SOUZA. DESENVOLVIMENTO DE UM MÓDULO DIDÁTICO PARA ENSINO DE AUTOMAÇÃO INDUSTRIAL. *[S. l.]*, 2017.

JOVIC, F. **Process control systems: Principles of design and operation**. Gulf publishing company. Houston, Texas: 1986.

JUNIOR, Vivaldo Alexandre da Silva. Implementação de controle de temperatura do tipo splirange (faixa dividida) em um sistema de refrigeração. Natal, RN, 2010.

MAITELLI, André Laurindo; CARVALHO, Fábio Câmara Araújo De. Programa computacional interativo para simulação e otimização de controladores PID. *[S. l.]*, 2003.

MIR, Arsheen; SWARNALATHA, R. Implementation of an industrial automation system model using an Arduino. **Journal of Engineering Science and Technology**, *[S. l.]*, v. 13, p. 4131–4144, 2018.

MUELLER, J. P. Começando a programar em Python para leigos. Rio de Janeiro: Alta Books, 2020. ISBN 9788550815855. Disponível em: https://search.ebscohost.com/login.aspx?direct=true&db=edsmib&AN=edsmib.000020984&lang=pt-br&site=eds-live&scope=site. Acesso em: 14 dez. 2021.

NAIR, Aiswarya Lakshmi Sasidharan; MARY, S. Anitha Janet; LINSELY, J. Arul. Modeling and control of level control process - A comparative study. **2017 Innovations in Power and Advanced Computing Technologies, i-PACT 2017**, Thuckalay, p. 1–4, 2017. DOI: 10.1109/IPACT.2017.8245161.

NEGREIROS, Bruno Azevedo Ferraz De. Desenvolvimento de módulo didático de baixo custo para ensino de absorção reativa por intermédio da aprendizagem significativa. *[S. l.]*, 2019.

RAMOS, Adriano Peixoto; WENSE, Gabriel lula barros. Sistema Didático De Nível De Líquidos. Brasília, DF, 2008.

RODRIGUES, Maria João Mortágua. PID Control of Water in a tank. *[S. l.]*, 2011.

ROGGIA, Leandro; FUENTES, Rodrigo Cardozo. **Automação Industrial**. Santa Maria, RS.

SBARBARO, Daniel; ORTEGA, Romeo. Averaging level control: An approach based on mass balance. **Journal of Process Control**, *[S. l.]*, v. 17, p. 621–629, 2007. DOI: 10.1016/j.jprocont.2007.01.005.

SILVA, Rodrigo Allan. Desenvolvimento de módulo de controle de nível. Cornélio Procópio, PR, 2014.

SIMONELLI, George; MAI, Breno Frigini; VICHELO, Deiver Robson Saccani; DE MARCHI, Humberto Frigini; DE CARVALHO, Romero Florentino. Simulação Do Controle De Uma Coluna De Destilação Descontínua Utilizando O Scilab. **Engevista**, *[S. l.]*, v. 19, p. 498–519, 2017. DOI: 10.22409/engevista.v19i2.870.

SOUZA, Fabio. Arduino MEGA 2650. **Embarcados**, 2020. Disponível em: <https://www.embarcados.com.br/arduino-mega-2560/>. Acesso em: 10 de jun. de 2020.

SOUZA, Fabio. Arduino UNO. **Embarcados**, 2020. Disponível em: <https://www.embarcados.com.br/arduino-mega-2560/>. Acesso em: 10 de jun. de 2020.

SMITH, Carlos A.; CORRIPIO, Armando. **Princípios e Prática do Controle Automático de Processo.** [s.l.] : LTC, 2008. Disponível em: http://search.ebscohost.com/login.aspx?direct=true&db=edsmib&AN=edsmib.000003311&lang=pt-br&site=eds-live&scope=site.

STEVAN, Sergio Luiz. **Automação e instrumentação industrial com Arduino : teoria e projetos.** São José dos Campos, SP: Erica, 2015. Disponível em: http://search.ebscohost.com/login.aspx?direct=true&db=edsmib&AN=edsmib.000009255&lang=pt-br&site=eds-live&scope=site.

THOMSEN, Adilson. O que é Arduino? **Filipeflop**, 2014. Disponível em: <https://www.filipeflop.com/blog/o-que-e-arduino/>. Acesso em: 10 de jun. de 2020.

THOMSEN, Adilson. Como conectar o Sensor Ultrassônico HC-SR04 ao Arduino? **Filipeflop**, 2011. Disponível em: <https://www.filipeflop.com/blog/sensor-ultrassonico-hc-sr04-ao-arduino/>. Acesso em: 14 de jun. de 2020.

WAWGINIAKS, Michael Lanes. Desenvolvimento do protótipo de uma bancada didática para experimentos educacionais envolvendo controle de temperatura. Panambi, RS, p. 0–89, 2017.

ZHOU, Feng; PENG, Hui; ZENG, Xiaoyong; TIAN, Xiaoying; WU, Jun. Modeling and Control Approach to Coupled Tanks Liquid Level System Based on Function-Type Weight RBF-ARX Model. **Asian Journal of Control**, *[S. l.]*, v. 19, n. 2, p. 692–707, 2017. DOI: 10.1002/asjc.1393.

APÊNDICE A - Código Microcontrolador

main.ino

```
#include <Wire.h>
#include "Pinout.h"
#include "Nivel.h"
#include <PID_v1.h>
#include <EEPROM.h>
#include "HX711.h"


#define DOUT  A0
#define CLK  A1

//Variaveis PID
double PID_Setpoint, PID_Input,PID_Output;
bool parametros_enviados = false;


//Variaveis do processo
float nivelLiq = 0;
float nivelLiq_ = 0;
float zero_liquido = 0;

float peso = 0;

float altura_ultra = 0;
float porcen_ultra = 0;
float altura_media = 0;
float porcen_media = 0;
float altura_balan = 0;
float porcen_balan = 0;


bool modo_PID = false;
bool Pump_state = false;
bool manual = false;
float PWM_pump = 0;

bool enviarDados = false;

unsigned long time;
unsigned long timer;

//Parametros
double Kp, Ki, Kd, densidade, area, tara, altura_tanque;

// Endereços EEPROM
int Kp_addr = 0, Ki_addr = 20, Kd_addr = 40, mode_addr = 60, PID_Setpoint_addr
= 80, tara_addr = 100, area_addr = 120, densidade_addr = 140,
```

```
unidade_medida_addr = 160, altura_tanque_addr = 180, calibracao_balanca_addr =
200;
int read_mode = 0; // 0->Ultrassonico, 1-> Balança, 2-> Media entre balança e
ultrassonico
int unidade_medida = 0; // 0-> %, 1-> Cm



PID myPID(&PID_Input, &PID_Output, &PID_Setpoint, Kp, Ki, Kd, DIRECT);

HX711 balanca;
float calibration_factor = 21970.00;
Nivel nivel;


//Eventos comunicação serial
String inputString = "";
boolean stringComplete = false;
boolean isConnected = false;

void setup()
{
  timer = time;
  Serial.begin(9600);
  PID_Setpoint = 20;
  getEEPROM();
  myPID.SetTunings(Kp, Ki, Kd); //trocar parametros PID
  myPID.SetMode(AUTOMATIC);
  balanca.begin(DOUT, CLK);                          // inicializa a balança
  balanca.set_scale(calibration_factor);             // ajusta fator de
calibração
  balanca.tare();

  while (!Serial.available() && parametros_enviados == false)
  {
    sendData();
    parametros_enviados = true;
  }
}

void loop()
{

  time = millis();

  //COMUNICAÇÃO SERIAL
  if (stringComplete)
  {
    stringComplete = false;
```

```
  byte inicio = 0;
  byte final = 0;

  for (int i = 0; i < inputString.length(); i++) {

    if (inputString[i] == '#')
    {
      inicio = i;
    }

    if (inputString[i] == '*')
    {
      final = i;
      String comando = inputString.substring(inicio, final);
      realizarComando(comando);
    }
  }
  inputString = "";
}



//DELAY PARA O CODIGO FLUIR
if ((time - timer) > 250)
{
  nivel.loop();
  readLevel();

  if (manual == true) {
    if (Pump_state == true)
    {
      if (nivel.distance > 8)
      {
        analogWrite(Pinout::pumpPin, PWM_pump);
      } else {
        analogWrite(Pinout::pumpPin, 0);
        delay(2000);
      }
    }
    if (Pump_state == false)
    {
      analogWrite(Pinout::pumpPin, 0);
    }
  }
  if (manual == false)
  {
    if (Pump_state == true)
    {
      //SETAR A VARIAVEL DE INPUT
```

```
        PID_Input = nivelLiq;
        myPID.Compute();

        if (nivel.distance > 8)
        {
          analogWrite(Pinout::pumpPin, PID_Output);
        } else {
          analogWrite(Pinout::pumpPin, 0);
          delay(2000);
        }
      }
    }
    if (Pump_state == false)
    {
      analogWrite(Pinout::pumpPin, 0);
    }
    timer = time;
  }
}




void readLevel() {

  altura_ultra = nivel.readSonar();
  porcen_ultra = (altura_ultra / nivel.tankHeight) * 100;
  peso = balanca.get_units(5);
  altura_balan = peso*2.7700831;
  porcen_balan = (altura_balan / nivel.tankHeight) * 100;
  altura_media = (altura_ultra + altura_balan) / 2;
  porcen_media = (altura_media / nivel.tankHeight) * 100;


  if (unidade_medida == 1)
  {
    if (read_mode == 0)
    {
      nivelLiq = altura_ultra;
    }
    else if (read_mode == 1)
    {
      nivelLiq = altura_balan;
    }
    else if (read_mode == 2)
    {
      nivelLiq = altura_media;
    }
```

```
    else
    {
      nivelLiq = altura_media;
    }

    if (enviarDados == true)
    {
      Serial.print("*");
      Serial.print(altura_ultra);
      Serial.print("#");
      Serial.print(altura_balan);
      Serial.print("#");
      Serial.print(altura_media);
      Serial.print("#");
      Serial.print(nivel.distance);
      Serial.print("#");
      Serial.print(peso);
      Serial.println("*");
    }

  }

  if (unidade_medida == 0)
  {
    if (read_mode == 0)
    {
      nivelLiq = porcen_ultra;
    }
    else if (read_mode == 1)
    {
      nivelLiq = porcen_balan;
    }
    else if (read_mode == 2)
    {
      nivelLiq = porcen_media;
    }
    else
    {
      nivelLiq = porcen_ultra;
    }

    if (enviarDados == true)
    {
      Serial.print("*");
      Serial.print(porcen_ultra);
      Serial.print("#");
      Serial.print(porcen_balan);
      Serial.print("#");
      Serial.print(porcen_media);
```

```
      Serial.print("#");
      Serial.print(nivel.distance);
      Serial.print("#");
      Serial.print(peso);
      Serial.println("*");
    }
  }
}


void getEEPROM() //CARREGAR VARIAVEIS DA EEPROM
{
  EEPROM.get(PID_Setpoint_addr, PID_Setpoint);
  EEPROM.get(Kp_addr,
Kp);
  EEPROM.get(Ki_addr, Ki);
  EEPROM.get(Kd_addr, Kd);
  EEPROM.get(tara_addr, nivel.tara);
  EEPROM.get(area_addr, nivel.areaBase);
  EEPROM.get(altura_tanque_addr, nivel.tankHeight);
  EEPROM.get(densidade_addr, nivel.density);
  EEPROM.get(mode_addr, read_mode);
  EEPROM.get(unidade_medida_addr, unidade_medida);
}


void updateEEPROM() //SALVAR AS VARIAVEIS EEPROM
{
  EEPROM.put(PID_Setpoint_addr, PID_Setpoint);
  EEPROM.put(Kp_addr, Kp);
  EEPROM.put(Ki_addr, Ki);
  EEPROM.put(Kd_addr, Kd);
  EEPROM.put(tara_addr, nivel.tara);
  EEPROM.put(area_addr,  nivel.areaBase);
  EEPROM.put(altura_tanque_addr, nivel.tankHeight);
  EEPROM.put(densidade_addr, nivel.density);
  EEPROM.put(mode_addr, read_mode);
  EEPROM.put(unidade_medida_addr, unidade_medida);
}


void sendData()
{
  Serial.println("#KP=" + String(Kp) + "*");
  Serial.println("#KI=" + String(Ki) + "*");
  Serial.println("#KD=" + String(Kd) + "*");
  delay(200);
  Serial.println("#SP=" + String(PID_Setpoint) + "*");
  Serial.println("#UM=" + String(unidade_medida) + "*");
```

```
  Serial.println("#MD=" + String(read_mode) + "*");
  delay(200);
  Serial.println("#TR=" + String(nivel.tara) + "*");
  Serial.println("#AR=" + String(nivel.areaBase) + "*");
  Serial.println("#DS=" + String(nivel.density) + "*");
  delay(200);
  Serial.println("#FIM*");
}


void realizarComando(String input) {

  String inputString = input;
  String comandoPrincipal = "";
  String comandoSecundario = "";

  if (inputString.length() > 0)
  {
    comandoPrincipal = inputString.substring(1, 4);
    comandoSecundario = "";
  }

  if (comandoPrincipal.equals("STR"))
  {
    enviarDados = true;
    Pump_state = true;
  }

  if (comandoPrincipal.equals("STP"))
  {
    Pump_state = false;
    enviarDados = false;
  }
  if (comandoPrincipal.equals("PMP"))
  {
    String value = inputString.substring(4, inputString.length());
    if (value == "150")
    {
      Pump_state = false;
      enviarDados = false;
      manual = false;
    }
    else if (value == "200")
    {
      Pump_state = true;
      enviarDados = true;
      manual = true;
    }
    else
```

```
    {
      PWM_pump = (value.toFloat() / 100) * 255;
    }
  }

  if (comandoPrincipal.equals("SAV"))
  {
    updateEEPROM();
  }

  if (comandoPrincipal.equals("LOA"))
  {
    getEEPROM();
  }

  if (comandoPrincipal.equals("SND"))
  {
    getEEPROM();
    delay(50);
    sendData();
  }

  if (comandoPrincipal.equals("SET"))
  {
    comandoSecundario = inputString.substring(4, 6);
    if (comandoSecundario.equals("KI"))
    {
      String value = inputString.substring(6, inputString.length());
      Ki = value.toDouble();
      myPID.SetTunings(Kp, Ki, Kd); //trocar parametros PID
    }
    if (comandoSecundario.equals("KP"))
    {
      String value = inputString.substring(6, inputString.length());
      Kp = value.toDouble();
      myPID.SetTunings(Kp, Ki, Kd); //trocar parametros PID
    }

    if (comandoSecundario.equals("KD"))
    {
      String value = inputString.substring(6, inputString.length());
      Kd = value.toDouble();
      myPID.SetTunings(Kp, Ki, Kd); //trocar parametros PID
    }

    if (comandoSecundario.equals("SP"))
    {
      String value = inputString.substring(6, inputString.length());
      PID_Setpoint = value.toDouble();
```

```
    }
    if (comandoSecundario.equals("MD"))
    {
      String value = inputString.substring(6, inputString.length());
      read_mode = value.toInt();
    }

    if (comandoSecundario.equals("DS"))
    {
      String value = inputString.substring(6, inputString.length());
      nivel.density = densidade;
      EEPROM.put(densidade_addr,  nivel.density);
    }

    if (comandoSecundario.equals("ED")) //Estimar Densidade
    {
      nivel.density = estimar_densidade();
      EEPROM.put(densidade_addr, nivel.density);
    }

    if (comandoSecundario.equals("TR"))
    {
      balanca.tare();
    }

    if (comandoSecundario.equals("AR"))
    {
      String value = inputString.substring(6, inputString.length());
      EEPROM.put(area_addr, nivel.areaBase);
    }

    if (comandoSecundario.equals("AT"))
    {
      nivel.setTankHeigth();
      Serial.print("Altura calibrada tanque :");
      Serial.println(nivel.tankHeight);
      EEPROM.put(altura_tanque_addr, nivel.tankHeight);

    }

    if (comandoSecundario.equals("UM"))
    {
      String value = inputString.substring(6, inputString.length());
      unidade_medida = value.toInt();
      EEPROM.put(unidade_medida_addr, unidade_medida);
    }
  }
}
```

```
void serialEvent() //EVENTOS SERIAL
{
  while (Serial.available()) {
    char inChar = (char)Serial.read();
    inputString += inChar;
    if (inChar == '\n') {
      stringComplete = true;
    }
  }
}



float estimar_densidade()
{
  int i = 0;
  float leituras_ultra = 0;
  float leituras_balan = 0;

  while (i<=10){
    leituras_ultra += nivel.readSonar();
    leituras_balan += balanca.get_units(5);
    i+=1;
  }

  float media_ultra = leituras_ultra/i;
  float media_balan = leituras_balan/i;

  return media_balan/(media_ultra*361);

}
```

pinout.cpp

```cpp
#include "Pinout.h"

namespace Pinout
{
  const int echoPin {11};
  const int triggerPin {12};
  const int pumpPin {3};
};
```

pinout.h

```cpp
#ifndef PINOUT_H
#define PINOUT_H

namespace Pinout
{
  extern const int echoPin;
  extern const int triggerPin;
  extern const int pumpPin;
}
```

nivel.h

```
#ifndef NIVEL_H
#define NIVEL_H
#include <NewPing.h>
#include "HX711.h"
#include "Pinout.h"
// #include <Ultrasonic.h>

class Nivel
{

  public:


    void loop();
    float readSonar();
    float setTankHeigth();


    float distance;
    float peso;
    float areaBase;
    float density;
    float volume;
    float tankHeight;
    float tara;


  NewPing sonar {Pinout::triggerPin, Pinout::echoPin, 100};

  private:
  float levelNivelSonar;
};

#endif
```

nivel.cpp

```
#include "Arduino.h"
#include "Nivel.h"
#include "Pinout.h"


float Nivel::readSonar()
{
    levelNivelSonar = abs(tankHeight - distance);
    return levelNivelSonar;
};



float Nivel::setTankHeigth()
{

  int i = 0;
  float leituras = 0;

  while (i<=10){

  if (sonar.ping_cm() != 0) {
    leituras += sonar.ping_cm();
    i+=1;
  }
  }
  tankHeight = leituras/i;
};



void Nivel::loop()
{

  if (sonar.ping_cm() != 0) {
    distance = sonar.ping_cm();
  }
};
```

NewPing.h

```
// ---------------------------------------------------------------------------
// NewPing Library - v1.9.1 - 07/15/2018
//
// AUTHOR/LICENSE:
// Created by Tim Eckel - teckel@leethost.com
// Copyright 2018 License: Forks and derivitive works are NOT permitted
without
// permission. Permission is only granted to use as-is for private and
// commercial use. If you wish to contribute, make changes, or enhancements,
// please create a pull request. I get a TON of support issues from this
// library, most of which comes from old versions, buggy forks or sites
without
// proper documentation, just trying to wrangle this project together.
//
// LINKS:
// Project home: https://bitbucket.org/teckel12/arduino-new-ping/wiki/Home
// Blog: http://arduino.cc/forum/index.php/topic,106043.0.html
//
// DISCLAIMER:
// This software is furnished "as is", without technical support, and with no
// warranty, express or implied, as to its usefulness for any purpose.
//
// BACKGROUND:
// When I first received an ultrasonic sensor I was not happy with how poorly
// it worked. Quickly I realized the problem wasn't the sensor, it was the
// available ping and ultrasonic libraries causing the problem. The NewPing
// library totally fixes these problems, adds many new features, and breaths
// new life into these very affordable distance sensors.
//
// FEATURES:
// * Works with many different ultrasonic sensors: SR04, SRF05, SRF06, DYP-
ME007, URM37 & Parallax PING)))™.
// * Compatible with the entire Arduino line-up (and clones), Teensy family
(including $19.80 96Mhz 32 bit Teensy 3.2) and non-AVR microcontrollers.
// * Interface with all but the SRF06 sensor using only one Arduino pin.
// * Doesn't lag for a full second if no ping/echo is received.
// * Ping sensors consistently and reliably at up to 30 times per second.
// * Timer interrupt method for event-driven sketches.
// * Built-in digital filter method ping_median() for easy error correction.
// * Uses port registers for a faster pin interface and smaller code size.
// * Allows you to set a maximum distance where pings beyond that distance are
read as no ping "clear".
// * Ease of using multiple sensors (example sketch with 15 sensors).
// * More accurate distance calculation (cm, inches & uS).
// * Doesn't use pulseIn, which is slow and gives incorrect results with some
ultrasonic sensor models.
// * Actively developed with features being added and bugs/issues addressed.
//
```

```
// CONSTRUCTOR:
//   NewPing sonar(trigger_pin, echo_pin [, max_cm_distance])
//     trigger_pin & echo_pin - Arduino pins connected to sensor trigger and
echo.
//       NOTE: To use the same Arduino pin for trigger and echo, specify the
same pin for both values.
//     max_cm_distance - [Optional] Maximum distance you wish to sense.
Default=500cm.
//
// METHODS:
//   sonar.ping([max_cm_distance]) - Send a ping and get the echo time (in
microseconds) as a result. [max_cm_distance] allows you to optionally set a
new max distance.
//   sonar.ping_in([max_cm_distance]) - Send a ping and get the distance in
whole inches. [max_cm_distance] allows you to optionally set a new max
distance.
//   sonar.ping_cm([max_cm_distance]) - Send a ping and get the distance in
whole centimeters. [max_cm_distance] allows you to optionally set a new max
distance.
//   sonar.ping_median(iterations [, max_cm_distance]) - Do multiple pings
(default=5), discard out of range pings and return median in microseconds.
[max_cm_distance] allows you to optionally set a new max distance.
//   NewPing::convert_in(echoTime) - Convert echoTime from microseconds to
inches (rounds to nearest inch).
//   NewPing::convert_cm(echoTime) - Convert echoTime from microseconds to
centimeters (rounds to nearest cm).
//   sonar.ping_timer(function [, max_cm_distance]) - Send a ping and call
function to test if ping is complete. [max_cm_distance] allows you to
optionally set a new max distance.
//   sonar.check_timer() - Check if ping has returned within the set distance
limit.
//   NewPing::timer_us(frequency, function) - Call function every frequency
microseconds.
//   NewPing::timer_ms(frequency, function) - Call function every frequency
milliseconds.
//   NewPing::timer_stop() - Stop the timer.
//
// HISTORY:
// 07/15/2018 v1.9.1 - Added support for ATtiny441 and ATtiny841
//   microcontrollers.
// 12/09/2017 v1.9.0 - Added support for the ARM-based Particle devices. If
//   other ARM-based microcontrollers adopt a similar timer method that the
//   Particle and Teensy 3.x share, support for other ARM-based
microcontrollers
//   could be possible. Attempt to add to Arduino library manager. License
//   changed.
// 07/30/2016 v1.8 - Added support for non-AVR microcontrollers. For non-AVR
//   microcontrollers, advanced ping_timer() timer methods are disabled due to
//   inconsistencies or no support at all between platforms. However, standard
```

```
//   ping methods are all supported. Added new optional variable to ping(),
//   ping_in(), ping_cm(), ping_median(), and ping_timer() methods which
allows
//   you to set a new maximum distance for each ping. Added support for the
//   ATmega16, ATmega32 and ATmega8535 microcontrollers. Changed convert_cm()
//   and convert_in() methods to static members. You can now call them without
//   an object. For example: cm = NewPing::convert_cm(distance);
//
// 09/29/2015 v1.7 - Removed support for the Arduino Due and Zero because
//   they're both 3.3 volt boards and are not 5 volt tolerant while the HC-
SR04
//   is a 5 volt sensor.  Also, the Due and Zero don't support pin
manipulation
//   compatibility via port registers which can be done (see the Teensy 3.2).
//
// 06/17/2014 v1.6 - Corrected delay between pings when using ping_median()
//   method. Added support for the URM37 sensor (must change URM37_ENABLED
from
//   false to true). Added support for Arduino microcontrollers like the $20
//   32 bit ARM Cortex-M4 based Teensy 3.2. Added automatic support for the
//   Atmel ATtiny family of microcontrollers. Added timer support for the
//   ATmega8 microcontroller. Rounding disabled by default, reduces compiled
//   code size (can be turned on with ROUNDING_ENABLED switch). Added
//   TIMER_ENABLED switch to get around compile-time "__vector_7" errors when
//   using the Tone library, or you can use the toneAC, NewTone or
//   TimerFreeTone libraries: https://bitbucket.org/teckel12/arduino-toneac/
//   Other speed and compiled size optimizations.
//
// 08/15/2012 v1.5 - Added ping_median() method which does a user specified
//   number of pings (default=5) and returns the median ping in microseconds
//   (out of range pings ignored). This is a very effective digital filter.
//   Optimized for smaller compiled size (even smaller than sketches that
//   don't use a library).
//
// 07/14/2012 v1.4 - Added support for the Parallax PING)))☺ sensor.
Interface
//   with all but the SRF06 sensor using only one Arduino pin. You can also
//   interface with the SRF06 using one pin if you install a 0.1uf capacitor
//   on the trigger and echo pins of the sensor then tie the trigger pin to
//   the Arduino pin (doesn't work with Teensy). To use the same Arduino pin
//   for trigger and echo, specify the same pin for both values. Various bug
//   fixes.
//
// 06/08/2012 v1.3 - Big feature addition, event-driven ping! Uses Timer2
//   interrupt, so be mindful of PWM or timing conflicts messing with Timer2
//   may cause (namely PWM on pins 3 & 11 on Arduino, PWM on pins 9 and 10 on
//   Mega, and Tone library). Simple to use timer interrupt functions you can
//   use in your sketches totally unrelated to ultrasonic sensors (don't use
if
```

```
//   you're also using NewPing's ping_timer because both use Timer2
interrupts).
//   Loop counting ping method deleted in favor of timing ping method after
//   inconsistent results kept surfacing with the loop timing ping method.
//   Conversion to cm and inches now rounds to the nearest cm or inch. Code
//   optimized to save program space and fixed a couple minor bugs here and
//   there. Many new comments added as well as line spacing to group code
//   sections for better source readability.
//
// 05/25/2012 v1.2 - Lots of code clean-up thanks to Arduino Forum members.
//   Rebuilt the ping timing code from scratch, ditched the pulseIn code as it
//   doesn't give correct results (at least with ping sensors). The NewPing
//   library is now VERY accurate and the code was simplified as a bonus.
//   Smaller and faster code as well. Fixed some issues with very close ping
//   results when converting to inches. All functions now return 0 only when
//   there's no ping echo (out of range) and a positive value for a successful
//   ping. This can effectively be used to detect if something is out of range
//   or in-range and at what distance. Now compatible with Arduino 0023.
//
// 05/16/2012 v1.1 - Changed all I/O functions to use low-level port registers
//   for ultra-fast and lean code (saves from 174 to 394 bytes). Tested on
both
//   the Arduino Uno and Teensy 2.0 but should work on all Arduino-based
//   platforms because it calls standard functions to retrieve port registers
//   and bit masks. Also made a couple minor fixes to defines.
//
// 05/15/2012 v1.0 - Initial release.
// ------------------------------------------------------------------------

#ifndef NewPing_h

    #define NewPing_h

    #if defined (ARDUINO) && ARDUINO >= 100
        #include <Arduino.h>
    #else
        #include <WProgram.h>
        #if defined (PARTICLE)
            #include <SparkIntervalTimer.h>
        #else
            #include <pins_arduino.h>
        #endif
    #endif

    #if defined (__AVR__)
        #include <avr/io.h>
        #include <avr/interrupt.h>
    #endif
```

```
    // Shouldn't need to change these values unless you have a specific need
to do so.
    #define MAX_SENSOR_DISTANCE 500 // Maximum sensor distance can be as high
as 500cm, no reason to wait for ping longer than sound takes to travel this
distance and back. Default=500
    #define US_ROUNDTRIP_CM 57      // Microseconds (uS) it takes sound to
travel round-trip 1cm (2cm total), uses integer to save compiled code space.
Default=57
    #define US_ROUNDTRIP_IN 146     // Microseconds (uS) it takes sound to
travel round-trip 1 inch (2 inches total), uses integer to save compiled code
space. Defalult=146
    #define ONE_PIN_ENABLED true    // Set to "false" to disable one pin mode
which saves around 14-26 bytes of binary size. Default=true
    #define ROUNDING_ENABLED false  // Set to "true" to enable distance
rounding which also adds 64 bytes to binary size. Default=false
    #define URM37_ENABLED false     // Set to "true" to enable support for the
URM37 sensor in PWM mode. Default=false
    #define TIMER_ENABLED true      // Set to "false" to disable the timer ISR
(if getting "__vector_7" compile errors set this to false). Default=true

    // Probably shouldn't change these values unless you really know what
you're doing.
    #define NO_ECHO 0               // Value returned if there's no ping echo
within the specified MAX_SENSOR_DISTANCE or max_cm_distance. Default=0
    #define MAX_SENSOR_DELAY 5800   // Maximum uS it takes for sensor to start
the ping. Default=5800
    #define ECHO_TIMER_FREQ 24      // Frequency to check for a ping echo
(every 24uS is about 0.4cm accuracy). Default=24
    #define PING_MEDIAN_DELAY 29000 // Microsecond delay between pings in the
ping_median method. Default=29000
    #define PING_OVERHEAD 5         // Ping overhead in microseconds (uS).
Default=5
    #define PING_TIMER_OVERHEAD 13  // Ping timer overhead in microseconds
(uS). Default=13
    #if URM37_ENABLED == true
        #undef  US_ROUNDTRIP_CM
        #undef  US_ROUNDTRIP_IN
        #define US_ROUNDTRIP_CM 50  // Every 50uS PWM signal is low indicates
1cm distance. Default=50
        #define US_ROUNDTRIP_IN 127 // If 50uS is 1cm, 1 inch would be 127uS
(50 x 2.54 = 127). Default=127
    #endif

    // Conversion from uS to distance (round result to nearest cm or inch).
    #define NewPingConvert(echoTime, conversionFactor) (max(((unsigned
int)echoTime + conversionFactor / 2) / conversionFactor, (echoTime ? 1 : 0)))

    // Detect non-AVR microcontrollers (Teensy 3.x, Arduino DUE, etc.) and
don't use port registers or timer interrupts as required.
```

```
    #if (defined (__arm__) && (defined (TEENSYDUINO) || defined (PARTICLE)))
        #undef  PING_OVERHEAD
        #define PING_OVERHEAD 1
        #undef  PING_TIMER_OVERHEAD
        #define PING_TIMER_OVERHEAD 1
        #define DO_BITWISE true
    #elif !defined (__AVR__)
        #undef  PING_OVERHEAD
        #define PING_OVERHEAD 1
        #undef  PING_TIMER_OVERHEAD
        #define PING_TIMER_OVERHEAD 1
        #undef  TIMER_ENABLED
        #define TIMER_ENABLED false
        #define DO_BITWISE false
    #else
        #define DO_BITWISE true
    #endif

    // Disable the timer interrupts when using ATmega128 and all ATtiny
microcontrollers.
    #if defined (__AVR_ATmega128__) || defined (__AVR_ATtiny24__) || defined
(__AVR_ATtiny44__) || defined (__AVR_ATtiny441__) || defined
(__AVR_ATtiny84__) || defined (__AVR_ATtiny841__) || defined
(__AVR_ATtiny25__) || defined (__AVR_ATtiny45__) || defined (__AVR_ATtiny85__)
|| defined (__AVR_ATtiny261__) || defined (__AVR_ATtiny461__) || defined
(__AVR_ATtiny861__) || defined (__AVR_ATtiny43U__)
        #undef  TIMER_ENABLED
        #define TIMER_ENABLED false
    #endif
    // Define timers when using ATmega8, ATmega16, ATmega32 and ATmega8535
microcontrollers.
    #if defined (__AVR_ATmega8__) || defined (__AVR_ATmega16__) || defined
(__AVR_ATmega32__) || defined (__AVR_ATmega8535__)
        #define OCR2A OCR2
        #define TIMSK2 TIMSK
        #define OCIE2A OCIE2
    #endif
    class NewPing {
        public:
            NewPing(uint8_t trigger_pin, uint8_t echo_pin, unsigned int
max_cm_distance = MAX_SENSOR_DISTANCE);
            unsigned int ping(unsigned int max_cm_distance = 0);
            unsigned long ping_cm(unsigned int max_cm_distance = 0);
            unsigned long ping_in(unsigned int max_cm_distance = 0);
            unsigned long ping_median(uint8_t it = 5, unsigned int
max_cm_distance = 0);
            static unsigned int convert_cm(unsigned int echoTime);
            static unsigned int convert_in(unsigned int echoTime);
    #if TIMER_ENABLED == true
```

```
            void ping_timer(void (*userFunc)(void), unsigned int
max_cm_distance = 0);
            boolean check_timer();
            unsigned long ping_result;
            static void timer_us(unsigned int frequency, void
(*userFunc)(void));
            static void timer_ms(unsigned long frequency, void
(*userFunc)(void));
            static void timer_stop();
    #endif
        private:
            boolean ping_trigger();
            void set_max_distance(unsigned int max_cm_distance);
    #if TIMER_ENABLED == true
            boolean ping_trigger_timer(unsigned int trigger_delay);
            boolean ping_wait_timer();
            static void timer_setup();
            static void timer_ms_cntdwn();
    #endif
    #if DO_BITWISE == true
            uint8_t _triggerBit;
            uint8_t _echoBit;
        #if defined(PARTICLE)
            #if !defined(portModeRegister)
                #if defined (STM32F10X_MD)
                    #define portModeRegister(port)     ( &(port->CRL) )
                #elif defined (STM32F2XX)
                    #define portModeRegister(port)     ( &(port->MODER) )
                #endif
            #endif
            volatile uint32_t *_triggerOutput;
            volatile uint32_t *_echoInput;
            volatile uint32_t *_triggerMode;
        #else
            volatile uint8_t *_triggerOutput;
            volatile uint8_t *_echoInput;
            volatile uint8_t *_triggerMode;
        #endif
    #else
            uint8_t _triggerPin;
            uint8_t _echoPin;
    #endif
            unsigned int _maxEchoTime;
            unsigned long _max_time;
    };


#endif
```

NewPing.cpp

```
// ---------------------------------------------------------------------------
// Created by Tim Eckel - teckel@leethost.com
//
// See NewPing.h for license, purpose, syntax, version history, links, etc.
// ---------------------------------------------------------------------------

#include "NewPing.h"


// ---------------------------------------------------------------------------
// NewPing constructor
// ---------------------------------------------------------------------------

NewPing::NewPing(uint8_t trigger_pin, uint8_t echo_pin, unsigned int
max_cm_distance) {
#if DO_BITWISE == true
    _triggerBit = digitalPinToBitMask(trigger_pin); // Get the port register
bitmask for the trigger pin.
    _echoBit = digitalPinToBitMask(echo_pin);       // Get the port register
bitmask for the echo pin.

    _triggerOutput = portOutputRegister(digitalPinToPort(trigger_pin)); // Get
the output port register for the trigger pin.
    _echoInput = portInputRegister(digitalPinToPort(echo_pin));         // Get
the input port register for the echo pin.

    _triggerMode = (uint8_t *)
portModeRegister(digitalPinToPort(trigger_pin)); // Get the port mode register
for the trigger pin.
#else
    _triggerPin = trigger_pin;
    _echoPin = echo_pin;
#endif

    set_max_distance(max_cm_distance); // Call function to set the max sensor
distance.

#if (defined (__arm__) && (defined (TEENSYDUINO) || defined(PARTICLE))) ||
DO_BITWISE != true
    pinMode(echo_pin, INPUT);     // Set echo pin to input (on Teensy 3.x
(ARM), pins default to disabled, at least one pinMode() is needed for GPIO
mode).
    pinMode(trigger_pin, OUTPUT); // Set trigger pin to output (on Teensy 3.x
(ARM), pins default to disabled, at least one pinMode() is needed for GPIO
mode).
#endif

#if defined (ARDUINO_AVR_YUN)
```

```
    pinMode(echo_pin, INPUT);     // Set echo pin to input for the Arduino
Yun, not sure why it doesn't default this way.
#endif

#if ONE_PIN_ENABLED != true && DO_BITWISE == true
    *_triggerMode |= _triggerBit; // Set trigger pin to output.
#endif
}


// ---------------------------------------------------------------------------
// Standard ping methods
// ---------------------------------------------------------------------------

unsigned int NewPing::ping(unsigned int max_cm_distance) {
    if (max_cm_distance > 0) set_max_distance(max_cm_distance); // Call
function to set a new max sensor distance.

    if (!ping_trigger()) return NO_ECHO; // Trigger a ping, if it returns
false, return NO_ECHO to the calling function.

#if URM37_ENABLED == true
    #if DO_BITWISE == true
        while (!(*_echoInput & _echoBit))            // Wait for the ping
echo.
    #else
        while (!digitalRead(_echoPin))               // Wait for the ping
echo.
    #endif
            if (micros() > _max_time) return NO_ECHO; // Stop the loop and
return NO_ECHO (false) if we're beyond the set maximum distance.
#else
    #if DO_BITWISE == true
        while (*_echoInput & _echoBit)               // Wait for the ping
echo.
    #else
        while (digitalRead(_echoPin))                // Wait for the ping
echo.
    #endif
            if (micros() > _max_time) return NO_ECHO; // Stop the loop and
return NO_ECHO (false) if we're beyond the set maximum distance.
#endif

    return (micros() - (_max_time - _maxEchoTime) - PING_OVERHEAD); //
Calculate ping time, include overhead.
}


unsigned long NewPing::ping_cm(unsigned int max_cm_distance) {
```

```
    unsigned long echoTime = NewPing::ping(max_cm_distance); // Calls the ping
method and returns with the ping echo distance in uS.
#if ROUNDING_ENABLED == false
    return (echoTime / US_ROUNDTRIP_CM);              // Call the ping method
and returns the distance in centimeters (no rounding).
#else
    return NewPingConvert(echoTime, US_ROUNDTRIP_CM); // Convert uS to
centimeters.
#endif
}


unsigned long NewPing::ping_in(unsigned int max_cm_distance) {
    unsigned long echoTime = NewPing::ping(max_cm_distance); // Calls the ping
method and returns with the ping echo distance in uS.
#if ROUNDING_ENABLED == false
    return (echoTime / US_ROUNDTRIP_IN);              // Call the ping method
and returns the distance in inches (no rounding).
#else
    return NewPingConvert(echoTime, US_ROUNDTRIP_IN); // Convert uS to inches.
#endif
}


unsigned long NewPing::ping_median(uint8_t it, unsigned int max_cm_distance) {
    unsigned int uS[it], last;
    uint8_t j, i = 0;
    unsigned long t;
    uS[0] = NO_ECHO;

    while (i < it) {
        t = micros();                // Start ping timestamp.
        last = ping(max_cm_distance);  // Send ping.

        if (last != NO_ECHO) {         // Ping in range, include as part of
median.
            if (i > 0) {               // Don't start sort till second ping.
                for (j = i; j > 0 && uS[j - 1] < last; j--) // Insertion sort
loop.
                    uS[j] = uS[j - 1];                     // Shift ping
array to correct position for sort insertion.
            } else j = 0;              // First ping is sort starting point.
            uS[j] = last;              // Add last ping to array in sorted
position.
            i++;                       // Move to next ping.
        } else it--;                   // Ping out of range, skip and don't
include as part of median.

        if (i < it && micros() - t < PING_MEDIAN_DELAY)
```

```
            delay((PING_MEDIAN_DELAY + t - micros()) / 1000); // Millisecond
delay between pings.

    }
    return (uS[it >> 1]); // Return the ping distance median.
}


// ---------------------------------------------------------------------------
// Standard and timer interrupt ping method support functions (not called
directly)
// ---------------------------------------------------------------------------

boolean NewPing::ping_trigger() {
#if DO_BITWISE == true
    #if ONE_PIN_ENABLED == true
        *_triggerMode |= _triggerBit;  // Set trigger pin to output.
    #endif

    *_triggerOutput &= ~_triggerBit;   // Set the trigger pin low, should
already be low, but this will make sure it is.
    delayMicroseconds(4);              // Wait for pin to go low.
    *_triggerOutput |= _triggerBit;    // Set trigger pin high, this tells the
sensor to send out a ping.
    delayMicroseconds(10);             // Wait long enough for the sensor to
realize the trigger pin is high. Sensor specs say to wait 10uS.
    *_triggerOutput &= ~_triggerBit;   // Set trigger pin back to low.

    #if ONE_PIN_ENABLED == true
        *_triggerMode &= ~_triggerBit; // Set trigger pin to input (when using
one Arduino pin, this is technically setting the echo pin to input as both are
tied to the same Arduino pin).
    #endif

    #if URM37_ENABLED == true
        if (!(*_echoInput & _echoBit)) return false;           // Previous
ping hasn't finished, abort.
        _max_time = micros() + _maxEchoTime + MAX_SENSOR_DELAY; // Maximum
time we'll wait for ping to start (most sensors are <450uS, the SRF06 can take
up to 34,300uS!)
        while (*_echoInput & _echoBit)                         // Wait for
ping to start.
            if (micros() > _max_time) return false;            // Took too
long to start, abort.
    #else
        if (*_echoInput & _echoBit) return false;              // Previous
ping hasn't finished, abort.
```

```
        _max_time = micros() + _maxEchoTime + MAX_SENSOR_DELAY; // Maximum
time we'll wait for ping to start (most sensors are <450uS, the SRF06 can take
up to 34,300uS!)
        while (!(*_echoInput & _echoBit))                       // Wait for
ping to start.
            if (micros() > _max_time) return false;             // Took too
long to start, abort.
    #endif
#else
    #if ONE_PIN_ENABLED == true
        pinMode(_triggerPin, OUTPUT); // Set trigger pin to output.
    #endif

    digitalWrite(_triggerPin, LOW);   // Set the trigger pin low, should
already be low, but this will make sure it is.
    delayMicroseconds(4);             // Wait for pin to go low.
    digitalWrite(_triggerPin, HIGH);  // Set trigger pin high, this tells the
sensor to send out a ping.
    delayMicroseconds(10);            // Wait long enough for the sensor to
realize the trigger pin is high. Sensor specs say to wait 10uS.
    digitalWrite(_triggerPin, LOW);   // Set trigger pin back to low.

    #if ONE_PIN_ENABLED == true
        pinMode(_triggerPin, INPUT);  // Set trigger pin to input (when using
one Arduino pin, this is technically setting the echo pin to input as both are
tied to the same Arduino pin).
    #endif

    #if URM37_ENABLED == true
        if (!digitalRead(_echoPin)) return false;               // Previous
ping hasn't finished, abort.
        _max_time = micros() + _maxEchoTime + MAX_SENSOR_DELAY; // Maximum
time we'll wait for ping to start (most sensors are <450uS, the SRF06 can take
up to 34,300uS!)
        while (digitalRead(_echoPin))                           // Wait for
ping to start.
            if (micros() > _max_time) return false;             // Took too
long to start, abort.
    #else
        if (digitalRead(_echoPin)) return false;                // Previous
ping hasn't finished, abort.
        _max_time = micros() + _maxEchoTime + MAX_SENSOR_DELAY; // Maximum
time we'll wait for ping to start (most sensors are <450uS, the SRF06 can take
up to 34,300uS!)
        while (!digitalRead(_echoPin))                          // Wait for
ping to start.
            if (micros() > _max_time) return false;             // Took too
long to start, abort.
    #endif
```

```
#endif

    _max_time = micros() + _maxEchoTime; // Ping started, set the time-out.
    return true;                         // Ping started successfully.
}


void NewPing::set_max_distance(unsigned int max_cm_distance) {
#if ROUNDING_ENABLED == false
    _maxEchoTime = min(max_cm_distance + 1, (unsigned int) MAX_SENSOR_DISTANCE
+ 1) * US_ROUNDTRIP_CM; // Calculate the maximum distance in uS (no rounding).
#else
    _maxEchoTime = min(max_cm_distance, (unsigned int) MAX_SENSOR_DISTANCE) *
US_ROUNDTRIP_CM + (US_ROUNDTRIP_CM / 2); // Calculate the maximum distance in
uS.
#endif
}


#if TIMER_ENABLED == true && DO_BITWISE == true

    // ----------------------------------------------------------------------
----
    // Timer interrupt ping methods (won't work with ATmega128, ATtiny and
most non-AVR microcontrollers)
    // ----------------------------------------------------------------------
----

    void NewPing::ping_timer(void (*userFunc)(void), unsigned int
max_cm_distance) {
        if (max_cm_distance > 0) set_max_distance(max_cm_distance); // Call
function to set a new max sensor distance.

        if (!ping_trigger()) return;         // Trigger a ping, if it returns
false, return without starting the echo timer.
        timer_us(ECHO_TIMER_FREQ, userFunc); // Set ping echo timer check
every ECHO_TIMER_FREQ uS.
    }


    boolean NewPing::check_timer() {
        if (micros() > _max_time) { // Outside the time-out limit.
            timer_stop();           // Disable timer interrupt
            return false;           // Cancel ping timer.
        }

    #if URM37_ENABLED == false
        if (!(*_echoInput & _echoBit)) { // Ping echo received.
    #else
```

```
        if (*_echoInput & _echoBit) {    // Ping echo received.
    #endif
            timer_stop();                // Disable timer interrupt
            ping_result = (micros() - (_max_time - _maxEchoTime) -
PING_TIMER_OVERHEAD); // Calculate ping time including overhead.
            return true;                 // Return ping echo true.
        }

        return false; // Return false because there's no ping echo yet.
    }


    // ---------------------------------------------------------------------
----
    // Timer2/Timer4 interrupt methods (can be used for non-ultrasonic needs)
    // ---------------------------------------------------------------------
----

    // Variables used for timer functions
    void (*intFunc)();
    void (*intFunc2)();
    unsigned long _ms_cnt_reset;
    volatile unsigned long _ms_cnt;
    #if defined(__arm__) && (defined (TEENSYDUINO) || defined(PARTICLE))
        IntervalTimer itimer;
    #endif


    void NewPing::timer_us(unsigned int frequency, void (*userFunc)(void)) {
        intFunc = userFunc; // User's function to call when there's a timer
event.
        timer_setup();      // Configure the timer interrupt.

    #if defined (__AVR_ATmega32U4__) // Use Timer4 for ATmega32U4
(Teensy/Leonardo).
        OCR4C = min((frequency>>2) - 1, 255); // Every count is 4uS, so divide
by 4 (bitwise shift right 2) subtract one, then make sure we don't go over 255
limit.
        TIMSK4 = (1<<TOIE4);                  // Enable Timer4 interrupt.
    #elif defined (__arm__) && defined (TEENSYDUINO) // Timer for Teensy 3.x
        itimer.begin(userFunc, frequency);           // Really simple on the
Teensy 3.x, calls userFunc every 'frequency' uS.
    #elif defined (__arm__) && defined (PARTICLE)    // Timer for Particle
devices
        itimer.begin(userFunc, frequency, uSec);     // Really simple on the
Particle, calls userFunc every 'frequency' uS.
    #else
```

```
        OCR2A = min((frequency>>2) - 1, 255); // Every count is 4uS, so divide
by 4 (bitwise shift right 2) subtract one, then make sure we don't go over 255
limit.
        TIMSK2 |= (1<<OCIE2A);               // Enable Timer2 interrupt.
    #endif
    }


    void NewPing::timer_ms(unsigned long frequency, void (*userFunc)(void)) {
        intFunc = NewPing::timer_ms_cntdwn;  // Timer events are sent here
once every ms till user's frequency is reached.
        intFunc2 = userFunc;                 // User's function to call when
user's frequency is reached.
        _ms_cnt = _ms_cnt_reset = frequency; // Current ms counter and reset
value.
        timer_setup();                       // Configure the timer interrupt.

    #if defined (__AVR_ATmega32U4__) // Use Timer4 for ATmega32U4
(Teensy/Leonardo).
        OCR4C = 249;           // Every count is 4uS, so 1ms = 250 counts - 1.
        TIMSK4 = (1<<TOIE4);   // Enable Timer4 interrupt.
    #elif defined (__arm__) && defined (TEENSYDUINO)        // Timer for
Teensy 3.x
        itimer.begin(NewPing::timer_ms_cntdwn, 1000);       // Set timer to
1ms (1000 uS).
    #elif defined (__arm__) && defined (PARTICLE)           // Timer for
Particle
        itimer.begin(NewPing::timer_ms_cntdwn, 1000, uSec); // Set timer to
1ms (1000 uS).
    #else
        OCR2A = 249;           // Every count is 4uS, so 1ms = 250 counts - 1.
        TIMSK2 |= (1<<OCIE2A); // Enable Timer2 interrupt.
    #endif
    }


    void NewPing::timer_stop() { // Disable timer interrupt.
    #if defined (__AVR_ATmega32U4__) // Use Timer4 for ATmega32U4
(Teensy/Leonardo).
        TIMSK4 = 0;
    #elif defined (__arm__) && (defined (TEENSYDUINO) || defined (PARTICLE))
// Timer for Teensy 3.x & Particle
        itimer.end();
    #else
        TIMSK2 &= ~(1<<OCIE2A);
    #endif
    }
```

```
    // ---------------------------------------------------------------------
----
    // Timer2/Timer4 interrupt method support functions (not called directly)
    // ---------------------------------------------------------------------
----

    void NewPing::timer_setup() {
    #if defined (__AVR_ATmega32U4__) // Use Timer4 for ATmega32U4
(Teensy/Leonardo).
        timer_stop(); // Disable Timer4 interrupt.
        TCCR4A = TCCR4C = TCCR4D = TCCR4E = 0;
        TCCR4B = (1<<CS42) | (1<<CS41) | (1<<CS40) | (1<<PSR4); // Set Timer4
prescaler to 64 (4uS/count, 4uS-1020uS range).
        TIFR4 = (1<<TOV4);
        TCNT4 = 0;    // Reset Timer4 counter.
    #elif defined (__AVR_ATmega8__) || defined (__AVR_ATmega16__) || defined
(__AVR_ATmega32__) || defined (__AVR_ATmega8535__) // Alternate timer commands
for certain microcontrollers.
        timer_stop();                // Disable Timer2 interrupt.
        ASSR &= ~(1<<AS2);           // Set clock, not pin.
        TCCR2 = (1<<WGM21 | 1<<CS22); // Set Timer2 to CTC mode, prescaler to
64 (4uS/count, 4uS-1020uS range).
        TCNT2 = 0;                   // Reset Timer2 counter.
    #elif defined (__arm__) && (defined (TEENSYDUINO) || defined (PARTICLE))
        timer_stop(); // Stop the timer.
    #else
        timer_stop();        // Disable Timer2 interrupt.
        ASSR &= ~(1<<AS2);   // Set clock, not pin.
        TCCR2A = (1<<WGM21); // Set Timer2 to CTC mode.
        TCCR2B = (1<<CS22);  // Set Timer2 prescaler to 64 (4uS/count, 4uS-
1020uS range).
        TCNT2 = 0;           // Reset Timer2 counter.
    #endif
    }


    void NewPing::timer_ms_cntdwn() {
        if (!_ms_cnt--) {            // Count down till we reach zero.
            intFunc2();              // Scheduled time reached, run the main
timer event function.
            _ms_cnt = _ms_cnt_reset; // Reset the ms timer.
        }
    }

    #if defined (__AVR_ATmega32U4__) // Use Timer4 for ATmega32U4
(Teensy/Leonardo).
    ISR(TIMER4_OVF_vect) {
        intFunc(); // Call wrapped function.
    }
```

```
    #elif defined (__AVR_ATmega8__) || defined (__AVR_ATmega16__) || defined
(__AVR_ATmega32__) || defined (__AVR_ATmega8535__) // Alternate timer commands
for certain microcontrollers.
    ISR(TIMER2_COMP_vect) {
        intFunc(); // Call wrapped function.
    }
    #elif defined (__arm__)
        // Do nothing...
    #else
    ISR(TIMER2_COMPA_vect) {
        intFunc(); // Call wrapped function.
    }
    #endif


#endif


// ---------------------------------------------------------------------------
// Conversion methods (rounds result to nearest cm or inch).
// ---------------------------------------------------------------------------

unsigned int NewPing::convert_cm(unsigned int echoTime) {
#if ROUNDING_ENABLED == false
    return (echoTime / US_ROUNDTRIP_CM);              // Convert uS to
centimeters (no rounding).
#else
    return NewPingConvert(echoTime, US_ROUNDTRIP_CM); // Convert uS to
centimeters.
#endif
}


unsigned int NewPing::convert_in(unsigned int echoTime) {
#if ROUNDING_ENABLED == false
    return (echoTime / US_ROUNDTRIP_IN);              // Convert uS to inches
(no rounding).
#else
    return NewPingConvert(echoTime, US_ROUNDTRIP_IN); // Convert uS to inches.
#endif
}
```

PID_v1.h

```
#ifndef PID_v1_h
#define PID_v1_h
#define LIBRARY_VERSION 1.2.1

class PID
{


  public:

  //Constants used in some of the functions below
  #define AUTOMATIC 1
  #define MANUAL  0
  #define DIRECT  0
  #define REVERSE  1
  #define P_ON_M 0
  #define P_ON_E 1

  //commonly used functions
**************************************************************************
    PID(double*, double*, double*,        // * constructor.  links the PID to
the Input, Output, and
        double, double, double, int, int);//   Setpoint.  Initial tuning
parameters are also set here.
                                          //   (overload for specifying
proportional mode)

    PID(double*, double*, double*,        // * constructor.  links the PID to
the Input, Output, and
        double, double, double, int);     //   Setpoint.  Initial tuning
parameters are also set here

    void SetMode(int Mode);               // * sets PID to either Manual (0)
or Auto (non-0)

    bool Compute();                       // * performs the PID
calculation.  it should be
                                          //   called every time loop()
cycles. ON/OFF and
                                          //   calculation frequency can be
set using SetMode
                                          //   SetSampleTime respectively

    void SetOutputLimits(double, double); // * clamps the output to a specific
range. 0-255 by default, but
                                          //   it's likely the user will want
to change this depending on
                                          //   the application
```

```
  //available but not commonly used functions
*************************************************************
    void SetTunings(double, double,       // * While most users will set the
tunings once in the
                    double);              //   constructor, this function
gives the user the option
                                          //   of changing tunings during
runtime for Adaptive control
    void SetTunings(double, double,       // * overload for specifying
proportional mode
                    double, int);

  void SetControllerDirection(int);   // * Sets the Direction, or "Action" of
the controller. DIRECT
                      //   means the output will increase when error is
positive. REVERSE
                      //   means the opposite.  it's very unlikely that this
will be needed
                      //   once it is set in the constructor.
    void SetSampleTime(int);              // * sets the frequency, in
Milliseconds, with which
                                          //   the PID calculation is
performed.  default is 100



  //Display functions
****************************************************************
  double GetKp();             // These functions query the pid for interal
values.
  double GetKi();             //  they were created mainly for the pid front-
end,
  double GetKd();             // where it's important to know what is actually
  int GetMode();              //  inside the PID.
  int GetDirection();           //

  private:
  void Initialize();

  double dispKp;        // * we'll hold on to the tuning parameters in user-
entered
  double dispKi;        //   format for display purposes
  double dispKd;        //

  double kp;                  // * (P)roportional Tuning Parameter
    double ki;                  // * (I)ntegral Tuning Parameter
```

```
    double kd;                  // * (D)erivative Tuning Parameter

  int controllerDirection;
  int pOn;

    double *myInput;              // * Pointers to the Input, Output, and
Setpoint variables
    double *myOutput;             //   This creates a hard link between the
variables and the
    double *mySetpoint;           //   PID, freeing the user from having to
constantly tell us
                                  //   what these values are.  with pointers
we'll just know.

  unsigned long lastTime;
  double outputSum, lastInput;

  unsigned long SampleTime;
  double outMin, outMax;
  bool inAuto, pOnE;
};
#endif
```

PID_V1.cpp

```
/**********************************************************************
*****************
 * Arduino PID Library - Version 1.2.1
 * by Brett Beauregard <br3ttb@gmail.com> brettbeauregard.com
 *
 * This Library is licensed under the MIT License
 **********************************************************************
****************/

#if ARDUINO >= 100
  #include "Arduino.h"
#else
  #include "WProgram.h"
#endif

#include <PID_v1.h>

/*Constructor (...)*********************************************************
 *    The parameters specified here are those for for which we can't set up
 *    reliable defaults, so we need to have the user set them.
 *********************************************************************/
PID::PID(double* Input, double* Output, double* Setpoint,
        double Kp, double Ki, double Kd, int POn, int ControllerDirection)
{
    myOutput = Output;
    myInput = Input;
    mySetpoint = Setpoint;
    inAuto = false;

    PID::SetOutputLimits(0, 255);            //default output limit
corresponds to
                                    //the arduino pwm limits

    SampleTime = 100;                    //default Controller Sample Time is
0.1 seconds

    PID::SetControllerDirection(ControllerDirection);
    PID::SetTunings(Kp, Ki, Kd, POn);

    lastTime = millis()-SampleTime;
}

/*Constructor (...)*********************************************************
 *    To allow backwards compatability for v1.1, or for people that just want
 *    to use Proportional on Error without explicitly saying so
 *********************************************************************/

PID::PID(double* Input, double* Output, double* Setpoint,
```

```
        double Kp, double Ki, double Kd, int ControllerDirection)
    :PID::PID(Input, Output, Setpoint, Kp, Ki, Kd, P_ON_E,
ControllerDirection)
{

}


/* Compute()
**********************************************************************
 *     This, as they say, is where the magic happens.  this function should be
called
 *   every time "void loop()" executes.  the function will decide for itself
whether a new
 *   pid Output needs to be computed.  returns true when the output is
computed,
 *   false when nothing has been done.
 **********************************************************************
*****/
bool PID::Compute()
{
   if(!inAuto) return false;
   unsigned long now = millis();
   unsigned long timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      /*Compute all the working error variables*/
      double input = *myInput;
      double error = *mySetpoint - input;
      double dInput = (input - lastInput);
      outputSum+= (ki * error);

      /*Add Proportional on Measurement, if P_ON_M is specified*/
      if(!pOnE) outputSum-= kp * dInput;

      if(outputSum > outMax) outputSum= outMax;
      else if(outputSum < outMin) outputSum= outMin;

      /*Add Proportional on Error, if P_ON_E is specified*/
      double output;
      if(pOnE) output = kp * error;
      else output = 0;

      /*Compute Rest of PID Output*/
      output += outputSum - kd * dInput;

       if(output > outMax) output = outMax;
      else if(output < outMin) output = outMin;
       *myOutput = output;
```

```
      /*Remember some variables for next time*/
      lastInput = input;
      lastTime = now;
       return true;
   }
   else return false;
}

/*
SetTunings(...)*************************************************************
 * This function allows the controller's dynamic performance to be adjusted.
 * it's called automatically from the constructor, but tunings can also
 * be adjusted on the fly during normal operation
 ******************************************************************************
*/
void PID::SetTunings(double Kp, double Ki, double Kd, int POn)
{
   if (Kp<0 || Ki<0 || Kd<0) return;

   pOn = POn;
   pOnE = POn == P_ON_E;

   dispKp = Kp; dispKi = Ki; dispKd = Kd;

   double SampleTimeInSec = ((double)SampleTime)/1000;
   kp = Kp;
   ki = Ki * SampleTimeInSec;
   kd = Kd / SampleTimeInSec;

  if(controllerDirection ==REVERSE)
   {
      kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
   }
}

/*
SetTunings(...)*************************************************************
 * Set Tunings using the last-rembered POn setting
 ******************************************************************************
*/
void PID::SetTunings(double Kp, double Ki, double Kd){
    SetTunings(Kp, Ki, Kd, pOn);
}

/* SetSampleTime(...)
*********************************************************
```

```
 * sets the period, in Milliseconds, at which the calculation is performed
 ******************************************************************************
*/
void PID::SetSampleTime(int NewSampleTime)
{
   if (NewSampleTime > 0)
   {
      double ratio  = (double)NewSampleTime
                      / (double)SampleTime;
      ki *= ratio;
      kd /= ratio;
      SampleTime = (unsigned long)NewSampleTime;
   }
}

/* SetOutputLimits(...)****************************************************
 *     This function will be used far more often than SetInputLimits.  while
 *  the input to the controller will generally be in the 0-1023 range (which
is
 *  the default already,)  the output will be a little different.  maybe
they'll
 *  be doing a time window and will need 0-8000 or something.  or maybe
they'll
 *  want to clamp it from 0-125.  who knows.  at any rate, that can all be
done
 *  here.
 **************************************************************************/
void PID::SetOutputLimits(double Min, double Max)
{
   if(Min >= Max) return;
   outMin = Min;
   outMax = Max;

   if(inAuto)
   {
      if(*myOutput > outMax) *myOutput = outMax;
      else if(*myOutput < outMin) *myOutput = outMin;

      if(outputSum > outMax) outputSum= outMax;
      else if(outputSum < outMin) outputSum= outMin;
   }
}

/*
SetMode(...)****************************************************************
 * Allows the controller Mode to be set to manual (0) or Automatic (non-zero)
 * when the transition from manual to auto occurs, the controller is
 * automatically initialized
```

```
 ******************************************************************************
*/
void PID::SetMode(int Mode)
{
    bool newAuto = (Mode == AUTOMATIC);
    if(newAuto && !inAuto)
    {  /*we just went from manual to auto*/
        PID::Initialize();
    }
    inAuto = newAuto;
}

/*
Initialize()****************************************************************
 * does all the things that need to happen to ensure a bumpless transfer
 *  from manual to automatic mode.
 ******************************************************************************
*/
void PID::Initialize()
{
   outputSum = *myOutput;
   lastInput = *myInput;
   if(outputSum > outMax) outputSum = outMax;
   else if(outputSum < outMin) outputSum = outMin;
}

/*
SetControllerDirection(...)*************************************************
 * The PID will either be connected to a DIRECT acting process (+Output leads
 * to +Input) or a REVERSE acting process(+Output leads to -Input.)  we need
to
 * know which one, because otherwise we may increase the output when we should
 * be decreasing.  This is called from the constructor.
 ******************************************************************************
*/
void PID::SetControllerDirection(int Direction)
{
   if(inAuto && Direction !=controllerDirection)
   {
       kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
   }
   controllerDirection = Direction;
}

/* Status
Funcions*******************************************************************
 * Just because you set the Kp=-1 doesn't mean it actually happened.  these
```

```
 * functions query the internal state of the PID.  they're here for display
 * purposes.  this are the functions the PID Front-end uses for example
 ******************************************************************************
*/
double PID::GetKp(){ return  dispKp; }
double PID::GetKi(){ return  dispKi;}
double PID::GetKd(){ return  dispKd;}
int PID::GetMode(){ return  inAuto ? AUTOMATIC : MANUAL;}
int PID::GetDirection(){ return controllerDirection;}
```

APÊNDICE B - Código Interface

Main.py

```
# -*- coding: utf-8 -*-
import wx
import TCCMainView
import TCCMainController

# Creating the application
app = wx.App()

# Creating the main Window
frm = TCCMainView.MainFrame(None, TCCMainController.MainFrameController())


# Showing the window
frm.Show()

# Starting the main loop
app.MainLoop()
```

TCCMainController.py

```
import wx
import TCCMainView
import TCCMainTemplate
from serial import Serial, serialutil
import serial.tools.list_ports
from sys import platform
import numpy as np


from database import DataBase
import plot_grafico

class MainFrameController:
    """

    MainFrameController controls the MainFrame using the Model-View-Controller
    design pattern.

    """

    # Initializes the class
    def __init__(self):
        DataBase.__init__(self)
        self.unidadeMedida = ""
        # # Inner variables
        self.connection = None

        self.running = False

        return

    def SetView(self, view):
        self.view = view

    def ViewInitialization(self):
        self.deviceConnectionTimer = wx.Timer(self.view, id=wx.ID_ANY)
        self.view.Bind(wx.EVT_TIMER,self.ReceiveMessages,self.deviceConnection
Timer)
        self.attStatusBar("Aguardando Conexão")

        self.x = np.array([])
        self.y = np.array([])
        self.y2 = np.array([])
        self.y3 = np.array([])
        self.setpoint = np.array([])
        self.x_counter = 0
        self.dados_grafico = [0,0,0]
```

```
        self.plot_forca = plot_grafico.CanvasPanel(self.view.m_panel6)
        self.addPointChart(0,0,0,0)

        print("VIEW INITI")
        return

    def attStatusBar(self, texto):
        self.view.lb_statusBar.SetStatusText("       "+texto, i=0)

    def addPointChart(self, y ,y2, y3 , setpoint):
        self.y = np.append(self.y, float(y))
        self.y2 = np.append(self.y2, float(y2))
        self.y3 = np.append(self.y3, float(y3))
        self.x = np.append(self.x, self.x_counter)
        self.setpoint = np.append(self.setpoint, float(setpoint))
        self.x_counter += 0.25
        self.attGrid()
        if len(self.x) > 20:
            self.plot_forca.draw(self.dados_grafico,self.x[-20:], self.y[-
20:],self.y2[-20:],self.y3[-20:],self.setpoint[-20:])
        else:
            self.plot_forca.draw(self.dados_grafico,self.x,
self.y,self.y2,self.y3,self.setpoint)

    def attStatus(self):
        self.view.lb_status_kp.LabelText = str(self.PARAMETROS["KP"])
        self.view.lb_status_ki.LabelText = str(self.PARAMETROS["KI"])
        self.view.lb_status_kd.LabelText = str(self.PARAMETROS["KD"])
        self.view.lb_status_setpoint.LabelText = str(self.PARAMETROS["SP"])

        # self.view.lb_status_alturaLiquido.LabelText = "Calculo altura do
liquido (altura do tanque - altura da leitura atual)"
        self.view.lb_status_areaBase.LabelText = str(self.PARAMETROS["AR"])
        # self.view.lb_status_volume.LabelText = "Calculo volume do liquido
(altura do liquido * area da base)"

        self.view.lb_status_tara.LabelText = str(self.PARAMETROS["TR"])
        self.view.lb_status_densidade.LabelText = str(self.PARAMETROS["DS"])
        # self.view.lb_status_volumeBalanca.LabelText = "Calculo volume do
liquido (massa * densidade)"
        # self.view.lb_status_pesoBalanca.LabelText = "leitura do peso da
balança"

    def attParameters(self):

        self.view.tb_kp.SetValue(str(self.PARAMETROS["KP"]))
        self.view.tb_ki.SetValue(str(self.PARAMETROS["KI"]))
```

```
        self.view.tb_kd.SetValue(str(self.PARAMETROS["KD"]))
        self.view.tb_setpoint.SetValue(str(self.PARAMETROS["SP"]))
        self.view.m_textCtrl10.SetValue(str(self.PARAMETROS["DS"]))

        # self.view.tb_kp.write(str(self.PARAMETROS["KP"]))
        # self.view.tb_ki.write(str(self.PARAMETROS["KI"]))
        # self.view.tb_kd.write(str(self.PARAMETROS["KD"]))
        # self.view.tb_setpoint.write(str(self.PARAMETROS["SP"]))
        # self.view.m_textCtrl10.write(str(self.PARAMETROS["DS"]))


        if self.PARAMETROS["UM"] == 0:
            self.view.unidadeMedida = "Altura do liquido (%)"
            self.view.lb_setpoint.LabelText = "Setpoint [%]"
            self.view.m_radioBtn1.SetValue(True)
            # self.m_radioBtn2.SetValue = False

        else:
            self.view.unidadeMedida = "Altura do liquido (cm)"
            self.view.lb_setpoint.LabelText = "Setpoint [cm]"
            self.view.m_radioBtn2.SetValue(True)
            # self.m_radioBtn1.SetValue = False

        if self.PARAMETROS["MD"] == 0:
            self.view.m_radioBtn11.SetValue(True)
        if self.PARAMETROS["MD"] == 1:
            self.view.m_radioBtn21.SetValue(True)
        if self.PARAMETROS["MD"] == 2:
            self.view.m_radioBtn211.SetValue(True)
        self.attStatus()

    def saveNewParameters(self):

        self.PARAMETROS["KP"] = self.view.tb_kp.GetLineText(0)
        self.PARAMETROS["KI"] = self.view.tb_ki.GetLineText(0)
        self.PARAMETROS["KD"] = self.view.tb_kd.GetLineText(0)
        self.PARAMETROS["SP"] = self.view.tb_setpoint.GetLineText(0)
        self.PARAMETROS["DS"] = self.view.m_textCtrl10.GetLineText(0)

        #ENVIAR PARA O MICROCONTROLADOR

#Comuinicação Serial
    def IsConnected(self):
        """ Returns the current connection state """
        #print(self.connection)
        if self.connection is not None:
            if self.connection.is_open:
                return True
            else:
```

```
                return False
        else:
            return False

    def Close(self):
        """ Closes the connection """
        # Stopping timer
        self.deviceConnectionTimer.Stop()
        self.view.aba_principal.Enable(False)

        if self.IsConnected():
            self.connection.close()
            # self.view.btn_cancel.Enable(False)
            # self.view.btn_check.Enable(True)
            # self.view.btn_transfer.Enable(True)
            self.view.cb_portasDisponiveis.Enable(True)
            self.attStatusBar("Conexão fechada - Aguardando nova conexão")
            self.connection = None
            self.running = False
            self.view.m_button25.SetLabel("Iniciar")

    def SendMessage(self, message):
        """ Sends message to the microcontroller """

        send_message = "#"+message+'*\n'
        send_message = send_message.encode(encoding='UTF-8',errors='replace')
        if self.IsConnected():
            self.connection.write(send_message)
            print("ENVIAR ->",send_message)
        return

    def ReceiveMessages(self, event):
        """ Listen to messagens from the microcontroller """
        if self.IsConnected():
            data = ""
            data = self.connection.readline().decode('UTF-
8').strip().strip("\r").strip("\n")
            if len(data) > 0:
                self.decodeMessage(data)
            return

    def decodeMessage(self, message):
        # print(message[0],"-",message[1:-1],"-",message[-1])

        if message[0] == "*" and message[-1] == "*":
            print("Nivel :",message.strip("*"))
            message = message.strip("*").strip(" ").split("#")
            self.addPointChart(float(message[0]),float(message[1]),float(messa
ge[2]),float(self.PARAMETROS["SP"]))
```

```
            print(message)

        if message[0] == "#" and message[-1] == "*":
            new_message = message.strip("#").strip("*")
            print("Parametro ->",new_message)
            if new_message != "FIM":
                submessage = new_message.split("=")
                print(submessage)
                self.PARAMETROS[submessage[0]] = float(submessage[1])
                # print(submessage[0] , float(submessage[1]))
            else:
                self.attParameters()

    def Reconnect(self):
        """ Try to reconnect after a connection exception """

        dial = wx.MessageDialog(self.view, 'Conexão perdida. Reconectar?',
'Falha de conexão',
            wx.YES_NO | wx.NO_DEFAULT | wx.ICON_QUESTION)
        dial.SetYesNoLabels("Sim", "Não")
        if dial.ShowModal() == wx.ID_YES:
            # Trying to reconnect
            self.Open(self.connectionPort, self.baudRate, self.timeOut,
self.readFreq, self.msgPerRead)
            # If this trial fails, shows the connection manager
            if not self.IsConnected():
                wx.MessageBox("Falha ao reestabelecer a conexão.", "Falha de
conexão",
                wx.OK | wx.ICON_ERROR)

    def onChoicePortCOM(self):
        self.view.port =
self.view.cb_portasDisponiveis.GetString(self.view.cb_portasDisponiveis.GetSel
ection())

        print("onChoicePortCOM", self.view.port)

    def onButtonRefreshCOM(self):

        self.view.cb_portasDisponiveis.Clear()
        self.attStatusBar("Aguarde - Carregando portas COM")
        numberOfPorts = 20
        if platform == 'linux':
            prefix = str('/dev/ttyUSB')
        else:
            prefix = str('COM')

        portList = []
```

```
        for n in range(0, numberOfPorts):
            # self.view.gaugeSearch.SetValue(n)
            port = prefix + str(n)
            if n == 10:
                port = '/dev/ttyACM0'
            try:
                getConnection = Serial(port, 9800, timeout=5)
                portList.append(port)
                getConnection.close()
                self.view.cb_portasDisponiveis.Append(port)
                print(port)

            except serialutil.SerialException:
                pass


        # ports = serial.tools.list_ports.comports()
        # for p in ports:
        #     print(p.device)
        self.attStatusBar("Aguardando Conexão")

    def onButtonConnectCOM(self):
        self.view.port =
self.view.cb_portasDisponiveis.GetString(self.view.cb_portasDisponiveis.GetSel
ection())
        # self.SendMessage("GET")
        if self.IsConnected() == False:

            self.Open(self.view.port)

    def onButtonCancelCOM(self):
        if self.IsConnected():
            self.Close()

    def Open(self, port=None, baudRate=9600, timeOut = 0.0, readFreq = 10,
msgPerRead = 20):
        """ Sets the current connection state and port """
        self.connectionPort = port
        self.baudRate = baudRate
        self.timeOut = timeOut # seconds
        self.readFreq = readFreq
        self.msgPerRead = msgPerRead
        self.bufferSize = 1024

        if self.connectionPort is None:
            self.connection = None
            return
        try:
```

```
            self.connection =
Serial(port=self.connectionPort,baudrate=self.baudRate,timeout=self.timeOut)
        except:
            self.connection = None
            return
        if self.IsConnected:
            self.deviceConnectionTimer.Start(1000.0/readFreq)
            self.attStatusBar("Porta "+ self.view.port +" Conectada")

            self.SendMessage("SND")
            self.view.aba_principal.Enable(True)

            # self.view.btn_cancel.Enable(True)
            # self.view.btn_check.Enable(False)
            # self.view.btn_transfer.Enable(False)
            self.view.cb_portasDisponiveis.Enable(False)


            # self.NotifyObservers()
        # self.connection =
Serial(port=self.view.port,baudrate=9800,timeout=2)
        # self.SerialConnect()

        print("onButtonConnectCOM")



    def clearGrid(self):
        self.view.gr_planilhaDados.ClearGrid()

    def attGrid(self):
        self.view.gr_planilhaDados.ClearGrid()
        self.view.gr_planilhaDados.SetColLabelValue(0,"Tempo")
        self.view.gr_planilhaDados.SetColLabelValue(1,"Ultra")
        self.view.gr_planilhaDados.SetColLabelValue(2,"Balanc")
        self.view.gr_planilhaDados.SetColLabelValue(3,"Media")
        self.view.gr_planilhaDados.SetColLabelValue(4,"Setpoint")
        # self.view.gr_planilhaDados.SetColLabelValue(2,"Deslocamento")
        # self.view.gr_planilhaDados.SetColLabelValue(3," ")
        # self.view.gr_planilhaDados.SetColLabelValue(4,"Propriedade")
        # self.view.gr_planilhaDados.SetColLabelValue(5,"Valor")


        self.view.gr_planilhaDados.DeleteRows(pos=0, numRows=-1)

        self.view.gr_planilhaDados.InsertRows(pos=0, numRows=len(self.x))
        # self.view.gr_planilhaDados.DeleteRows(pos=0, numRows=-1)
        # if prop_len >= data_len:
        #     self.view.gr_planilhaDados.InsertRows(pos=0, numRows=prop_len)
```

```
        # else:
        #     self.view.gr_planilhaDados.InsertRows(pos=0, numRows=data_len)

        index = 0
        for i in range(0,len(self.x)):
            self.view.gr_planilhaDados.SetCellValue(index,0,str(self.x[i]))
            self.view.gr_planilhaDados.SetCellValue(index,1,str(self.y[i]))
            self.view.gr_planilhaDados.SetCellValue(index,2,str(self.y2[i]))
            self.view.gr_planilhaDados.SetCellValue(index,3,str(self.y3[i]))
            self.view.gr_planilhaDados.SetCellValue(index,4,str(self.setpoint[
i]))
            index+=1

#Botões interface
    def onButtonCalibrarUltrassonico(self):

        dial = wx.MessageDialog(self.view, 'Setar altura de liquido atual como
0 do tanque?', 'Calibração do sensor Ultrassonico',wx.YES_NO | wx.NO_DEFAULT |
wx.ICON_QUESTION)
        dial.SetYesNoLabels("Sim", "Não")
        if dial.ShowModal() == wx.ID_YES:
            self.SendMessage("SETAT")
            print("Realizando calibração")

        print("Realizando calibração")

    def onButtonCalibrarCelulaCarga(self):

        dial = wx.MessageDialog(self.view, 'Setar peso de liquido atual como
tara?', 'Calibração Balança',wx.YES_NO | wx.NO_DEFAULT | wx.ICON_QUESTION)
        dial.SetYesNoLabels("Sim", "Não")
        if dial.ShowModal() == wx.ID_YES:
            self.SendMessage("SETTR")
            print("Realizando calibração")

        print("Realizando calibração")

    def radioPorcentagem(self):
        self.PARAMETROS["UM"] = 0

    def radioLitros(self):
        self.PARAMETROS["UM"] = 1

    def radioUltrassonico( self):
        self.PARAMETROS["MD"] = 0

    def radioCelulaCarga( self):
        self.PARAMETROS["MD"] = 1
```

```
    def radioAmbos( self):
        self.PARAMETROS["MD"] = 2

    def onButtonDensidade( self):
        self.SendMessage("PID")

    def onButtonMudarSetpoint( self):

        pass

    def onButtonLimparDados(self):
        dial = wx.MessageDialog(self.view, 'Gostaria de apagar os dados das
leituras?', 'Dados',wx.YES_NO | wx.NO_DEFAULT | wx.ICON_QUESTION)
        dial.SetYesNoLabels("Sim", "Não")
        if dial.ShowModal() == wx.ID_YES:
            self.x = np.array([])
            self.y = np.array([])
            self.y2 = np.array([])
            self.y3 = np.array([])
            self.setpoint = np.array([])
            self.x_counter = 0
            self.addPointChart(0,0,0,0)
            self.clearGrid()

    def onButtonSave( self):
        self.saveNewParameters()
        if self.running == False:
            for key in self.PARAMETROS.keys():
                self.SendMessage("SET"+key+str(self.PARAMETROS[key]))
            self.SendMessage("SAV")
        else:
            self.PARAMETROS["SP"] = self.view.tb_setpoint.GetLineText(0)
            self.SendMessage("SETSP"+str(self.PARAMETROS["SP"]))
            self.SendMessage("SETUM"+str(self.PARAMETROS["UM"]))
            self.SendMessage("SAV")

    def onButtonStart(self):
        if self.running:
            self.view.m_button25.SetLabel("Iniciar")
            self.SendMessage("STP")
            self.running = False
            self.view.tb_kp.Enable(True)
            self.view.tb_ki.Enable(True)
            self.view.tb_kd.Enable(True)
            self.view.m_textCtrl10.Enable(True)
            # self.view.lb_setpoint.Enable(True)
            self.view.m_radioBtn11.Enable(True)
            self.view.m_radioBtn21.Enable(True)
            self.view.m_radioBtn211.Enable(True)
```

```
            self.view.m_button23.Enable(True)
            self.view.btn_status_calibrarUltrassonico.Enable(True)
            self.view.btn_status_calibrarCelulaCarga.Enable(True)
            self.view.lb_kp.Enable(True)
            self.view.lb_kd.Enable(True)
            self.view.lb_ki.Enable(True)
        else:
            self.view.m_button25.SetLabel("Parar")
            # self.SendMessage("PID")
            self.SendMessage("STR")
            self.running = True
            self.view.tb_kp.Enable(False)
            self.view.tb_ki.Enable(False)
            self.view.tb_kd.Enable(False)
            self.view.m_textCtrl10.Enable(False)
            # self.view.lb_setpoint.Enable(False)
            self.view.lb_kp.Enable(False)
            self.view.lb_kd.Enable(False)
            self.view.lb_ki.Enable(False)
            self.view.m_radioBtn11.Enable(False)
            self.view.m_radioBtn21.Enable(False)
            self.view.m_radioBtn211.Enable(False)
            self.view.m_button23.Enable(False)
            self.view.btn_status_calibrarUltrassonico.Enable(False)
            self.view.btn_status_calibrarCelulaCarga.Enable(False)

    def onScroll( self ):
        self.view.m_staticText311.LabelText =
str(self.view.m_slider1.GetValue()) + "%"

    def onScrollChanged( self ):
        print("onScrollChanged")


    def onButtonSetPump( self ):
        self.SendMessage("PMP"+str(self.view.m_slider1.GetValue()))

    def onButtonStopPump( self ):
        self.SendMessage("PMP150")

    def onButtonStartPump( self ):
        self.SendMessage("PMP200")

    def onBoxDataUltra( self ):
            # self.SendMessage("SND")
        if self.view.m_checkBox4.GetValue():
            self.dados_grafico[0] = 1
        else:
            self.dados_grafico[0] = 0
```

```
        # pass

    def onBoxDataBalance( self ):
        if self.view.m_checkBox5.GetValue():
            self.dados_grafico[1] = 1
        else:
            self.dados_grafico[1] = 0


    def onBoxBoth( self ):
        if self.view.m_checkBox6.GetValue():
            self.dados_grafico[2] = 1
        else:
            self.dados_grafico[2] = 0
```

TCCMainView.py

```
import wx
import wx.adv

import TCCMainTemplate



class MainFrame ( TCCMainTemplate.MainFrame ):

    def __init__( self, parent, controller ):
        TCCMainTemplate.MainFrame.__init__ ( self, parent)
        self.controller = controller
        self.controller.SetView(self)
        self.controller.ViewInitialization()
        self.port = None
        # Initializes this view

    def onChoicePortCOM(self, event):
        self.controller.onChoicePortCOM()

    def onButtonRefreshCOM(self, event):
        self.controller.onButtonRefreshCOM()

    def onButtonCancelCOM( self, event ):
        self.controller.onButtonCancelCOM()

    def onButtonConnectCOM(self, event):
        self.controller.onButtonConnectCOM()

    def onButtonCalibrarUltrassonico(self, event):
        self.controller.onButtonCalibrarUltrassonico()

    def onButtonCalibrarCelulaCarga(self, event):
        self.controller.onButtonCalibrarCelulaCarga()

    def radioPorcentagem( self, event ):
        self.controller.radioPorcentagem()

    def radioLitros( self, event ):
        self.controller.radioLitros()

    def radioUltrassonico( self, event ):
        self.controller.radioUltrassonico()

    def radioCelulaCarga( self, event ):
        self.controller.radioCelulaCarga()
```

```
    def radioAmbos( self, event ):
        self.controller.radioAmbos()

    def onButtonCalibrarUltrassonico( self, event ):
        self.controller.onButtonCalibrarUltrassonico()

    def onButtonCalibrarCelulaCarga( self, event ):
        self.controller.onButtonCalibrarCelulaCarga()

    def onButtonDensidade( self, event ):
        self.controller.onButtonDensidade()

    def onButtonSave( self, event ):
        self.controller.onButtonSave()

    def onButtonStart( self, event ):
        self.controller.onButtonStart()

    def onButtonLimparDados( self, event ):
        self.controller.onButtonLimparDados()

    def onButtonMudarSetpoint( self, event ):
        self.controller.onButtonMudarSetpoint()

    def onScroll( self, event ):
        self.controller.onScroll()

    def onScrollChanged( self, event ):
        self.controller.onScrollChanged()

    def onButtonSetPump( self, event ):
        self.controller.onButtonSetPump()

    def onButtonStopPump( self, event ):
        self.controller.onButtonStopPump()

    def onButtonStartPump( self, event ):
        self.controller.onButtonStartPump()

    def onBoxDataUltra( self, event ):
        self.controller.onBoxDataUltra()

    def onBoxDataBalance( self, event ):
        self.controller.onBoxDataBalance()

    def onBoxBoth( self, event ):
        self.controller.onBoxBoth()
```

plot_grafico.py

```
import wx
import matplotlib
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as
FigureCanvas
from matplotlib.figure import Figure
import serial
import time


class CanvasPanel(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent,size=wx.Size(806, 450))

        self.figure = Figure()
        self.axes = self.figure.add_subplot(111)
        self.canvas = FigureCanvas(self, -1, self.figure)
        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.canvas, 1, wx.EXPAND)
        self.SetSizer(self.sizer)
        self.axes.set_xlabel("Pontos")
        self.axes.set_ylabel("Nivel")


    def draw(self,dados_grafico,x,y,y2,y3, setpoint):
        self.axes.clear()
        if dados_grafico[0] == 1:
            self.axes.plot(x,y, '-o')
        if dados_grafico[1] == 1:
            self.axes.plot(x,y2, '-o')
        if dados_grafico[2] == 1:
            self.axes.plot(x,y3, '-o')
        self.axes.plot(x,setpoint)
        self.canvas.draw()
```

TCCMainTemplate.py

```
# -*- coding: utf-8 -*-

###########################################################################
## Python code generated with wxFormBuilder (version Oct 26 2018)
## http://www.wxformbuilder.org/
##
## PLEASE DO *NOT* EDIT THIS FILE!
###########################################################################

import wx
import wx.xrc
import wx.grid

###########################################################################
## Class MainFrame
###########################################################################

class MainFrame ( wx.Frame ):

    def __init__( self, parent ):
        wx.Frame.__init__ ( self, parent, id = wx.ID_ANY, title = u"Controle
de Nível", pos = wx.DefaultPosition, size = wx.Size( 969,600 ), style =
wx.DEFAULT_FRAME_STYLE|wx.TAB_TRAVERSAL )

        self.SetSizeHints( wx.Size( -1,-1 ), wx.Size( -1,-1 ) )

        self.m_toolBar1 = self.CreateToolBar(
wx.TB_HORIZONTAL|wx.TB_HORZ_LAYOUT, wx.ID_ANY )
        self.m_toolBar1.SetToolBitmapSize( wx.Size( 24,24 ) )
        self.m_toolBar1.SetToolSeparation( 16 )
        self.m_staticText2 = wx.StaticText( self.m_toolBar1, wx.ID_ANY,
wx.EmptyString, wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText2.Wrap( -1 )

        self.m_toolBar1.AddControl( self.m_staticText2 )
        self.m_staticText79 = wx.StaticText( self.m_toolBar1, wx.ID_ANY,
wx.EmptyString, wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText79.Wrap( -1 )

        self.m_toolBar1.AddControl( self.m_staticText79 )
        cb_portasDisponiveisChoices = [ u"COM 4", u"COM 5" ]
        self.cb_portasDisponiveis = wx.Choice( self.m_toolBar1, wx.ID_ANY,
wx.DefaultPosition, wx.Size( 125,-1 ), cb_portasDisponiveisChoices, 0 )
        self.cb_portasDisponiveis.SetSelection( 0 )
        self.m_toolBar1.AddControl( self.cb_portasDisponiveis )
        self.m_toolBar1.AddSeparator()
```

```
        self.btn_transfer = self.m_toolBar1.AddLabelTool( wx.ID_ANY, u"tool",
wx.Bitmap( u"./icons/transfer.png", wx.BITMAP_TYPE_ANY ), wx.NullBitmap,
wx.ITEM_NORMAL, wx.EmptyString, wx.EmptyString, None )

        self.btn_check = self.m_toolBar1.AddLabelTool( wx.ID_ANY, u"tool",
wx.Bitmap( u"./icons/checked.png", wx.BITMAP_TYPE_ANY ), wx.NullBitmap,
wx.ITEM_NORMAL, wx.EmptyString, wx.EmptyString, None )

        self.btn_cancel = self.m_toolBar1.AddLabelTool( wx.ID_ANY, u"tool",
wx.Bitmap( u"./icons/cancel.png", wx.BITMAP_TYPE_ANY ), wx.NullBitmap,
wx.ITEM_NORMAL, wx.EmptyString, wx.EmptyString, None )

        self.m_toolBar1.Realize()

        bSizer1 = wx.BoxSizer( wx.HORIZONTAL )

        bSizer2 = wx.BoxSizer( wx.VERTICAL )

        self.aba_principal = wx.Notebook( self, wx.ID_ANY, wx.DefaultPosition,
wx.Size( 190,-1 ), 0 )
        self.pnl_parametros = wx.Panel( self.aba_principal, wx.ID_ANY,
wx.DefaultPosition, wx.Size( -1,-1 ), wx.TAB_TRAVERSAL )
        bSizer8 = wx.BoxSizer( wx.VERTICAL )

        sbSizer6 = wx.StaticBoxSizer( wx.StaticBox( self.pnl_parametros,
wx.ID_ANY, u"Parametros" ), wx.VERTICAL )

        bSizer9 = wx.BoxSizer( wx.HORIZONTAL )

        bSizer98 = wx.BoxSizer( wx.VERTICAL )

        self.lb_kp = wx.StaticText( sbSizer6.GetStaticBox(), wx.ID_ANY,
u"Valor kP", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.lb_kp.Wrap( -1 )

        bSizer98.Add( self.lb_kp, 0, wx.ALL, 5 )


        bSizer9.Add( bSizer98, 0, 0, 5 )

        bSizer41 = wx.BoxSizer( wx.VERTICAL )

        self.tb_kp = wx.TextCtrl( sbSizer6.GetStaticBox(), wx.ID_ANY,
wx.EmptyString, wx.Point( -1,-1 ), wx.Size( 64,-1 ), wx.TE_RIGHT )
        bSizer41.Add( self.tb_kp, 0, wx.ALL|wx.ALIGN_RIGHT, 5 )


        bSizer9.Add( bSizer41, 1, wx.EXPAND, 5 )
```

```
        sbSizer6.Add( bSizer9, 0, wx.EXPAND, 5 )

        bSizer91 = wx.BoxSizer( wx.HORIZONTAL )

        bSizer99 = wx.BoxSizer( wx.VERTICAL )

        self.lb_ki = wx.StaticText( sbSizer6.GetStaticBox(), wx.ID_ANY,
u"Valor kI", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.lb_ki.Wrap( -1 )

        bSizer99.Add( self.lb_ki, 0, wx.ALL, 5 )


        bSizer91.Add( bSizer99, 0, wx.ALIGN_CENTER_VERTICAL, 5 )

        bSizer40 = wx.BoxSizer( wx.VERTICAL )

        self.tb_ki = wx.TextCtrl( sbSizer6.GetStaticBox(), wx.ID_ANY,
wx.EmptyString, wx.DefaultPosition, wx.Size( 64,-1 ), wx.TE_RIGHT )
        bSizer40.Add( self.tb_ki, 0, wx.ALL|wx.ALIGN_RIGHT, 5 )


        bSizer91.Add( bSizer40, 1, wx.EXPAND, 5 )


        sbSizer6.Add( bSizer91, 0, wx.EXPAND, 5 )

        bSizer92 = wx.BoxSizer( wx.HORIZONTAL )

        bSizer100 = wx.BoxSizer( wx.VERTICAL )

        self.lb_kd = wx.StaticText( sbSizer6.GetStaticBox(), wx.ID_ANY,
u"Valor kD", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.lb_kd.Wrap( -1 )

        bSizer100.Add( self.lb_kd, 0, wx.ALL, 5 )


        bSizer92.Add( bSizer100, 0, wx.ALIGN_CENTER_VERTICAL, 5 )

        bSizer39 = wx.BoxSizer( wx.VERTICAL )

        self.tb_kd = wx.TextCtrl( sbSizer6.GetStaticBox(), wx.ID_ANY,
wx.EmptyString, wx.DefaultPosition, wx.Size( 64,-1 ), wx.TE_RIGHT )
        bSizer39.Add( self.tb_kd, 0, wx.ALL|wx.ALIGN_RIGHT, 5 )


        bSizer92.Add( bSizer39, 1, 0, 5 )
```

```
        sbSizer6.Add( bSizer92, 0, wx.EXPAND, 5 )

        bSizer93 = wx.BoxSizer( wx.HORIZONTAL )

        bSizer101 = wx.BoxSizer( wx.VERTICAL )

        self.lb_setpoint = wx.StaticText( sbSizer6.GetStaticBox(), wx.ID_ANY,
u"Setpoint [%]", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.lb_setpoint.Wrap( -1 )

        bSizer101.Add( self.lb_setpoint, 0, wx.ALL, 5 )


        bSizer93.Add( bSizer101, 0, wx.ALIGN_CENTER_VERTICAL, 5 )

        bSizer38 = wx.BoxSizer( wx.VERTICAL )

        self.tb_setpoint = wx.TextCtrl( sbSizer6.GetStaticBox(), wx.ID_ANY,
wx.EmptyString, wx.DefaultPosition, wx.Size( 64,-1 ), wx.TE_RIGHT )
        bSizer38.Add( self.tb_setpoint, 0, wx.ALL|wx.ALIGN_RIGHT, 5 )


        bSizer93.Add( bSizer38, 1, wx.EXPAND, 5 )


        sbSizer6.Add( bSizer93, 0, wx.EXPAND, 5 )

        sbSizer8 = wx.StaticBoxSizer( wx.StaticBox( sbSizer6.GetStaticBox(),
wx.ID_ANY, u"Unidade de Operação" ), wx.HORIZONTAL )

        bSizer99 = wx.BoxSizer( wx.VERTICAL )

        self.m_radioBtn1 = wx.RadioButton( sbSizer8.GetStaticBox(), wx.ID_ANY,
u"%", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_radioBtn1.SetValue( True )
        bSizer99.Add( self.m_radioBtn1, 0, wx.ALIGN_CENTER_HORIZONTAL|wx.ALL,
5 )


        sbSizer8.Add( bSizer99, 1, wx.EXPAND, 5 )

        bSizer100 = wx.BoxSizer( wx.VERTICAL )

        self.m_radioBtn2 = wx.RadioButton( sbSizer8.GetStaticBox(), wx.ID_ANY,
u"cm", wx.DefaultPosition, wx.DefaultSize, 0 )
        bSizer100.Add( self.m_radioBtn2, 0, wx.ALIGN_CENTER|wx.ALL, 5 )


        sbSizer8.Add( bSizer100, 1, wx.EXPAND, 5 )
```

```
        sbSizer6.Add( sbSizer8, 0, wx.EXPAND|wx.BOTTOM, 5 )

        sbSizer81 = wx.StaticBoxSizer( wx.StaticBox( sbSizer6.GetStaticBox(),
wx.ID_ANY, u"Modo de Operação" ), wx.VERTICAL )

        bSizer992 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_radioBtn11 = wx.RadioButton( sbSizer81.GetStaticBox(),
wx.ID_ANY, u"Ultrassonico", wx.DefaultPosition, wx.DefaultSize, wx.RB_GROUP )
        self.m_radioBtn11.SetValue( True )
        self.m_radioBtn11.SetToolTip( u"Utilizar apenas o sensor ultrassonico
para realizar a leitura do volume do liquido" )

        bSizer992.Add( self.m_radioBtn11, 0, wx.ALL, 5 )


        sbSizer81.Add( bSizer992, 1, wx.EXPAND, 5 )

        bSizer1001 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_radioBtn21 = wx.RadioButton( sbSizer81.GetStaticBox(),
wx.ID_ANY, u"Celula de carga", wx.DefaultPosition, wx.DefaultSize, 0 )
        bSizer1001.Add( self.m_radioBtn21, 0, wx.ALIGN_CENTER|wx.ALL, 5 )


        sbSizer81.Add( bSizer1001, 1, wx.EXPAND, 5 )

        bSizer10011 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_radioBtn211 = wx.RadioButton( sbSizer81.GetStaticBox(),
wx.ID_ANY, u"Ambos", wx.DefaultPosition, wx.DefaultSize, 0 )
        bSizer10011.Add( self.m_radioBtn211, 0, wx.ALIGN_CENTER|wx.ALL, 5 )


        sbSizer81.Add( bSizer10011, 1, wx.EXPAND, 5 )


        sbSizer6.Add( sbSizer81, 0, wx.EXPAND|wx.BOTTOM, 5 )

        sbSizer82 = wx.StaticBoxSizer( wx.StaticBox( sbSizer6.GetStaticBox(),
wx.ID_ANY, u"Densidade" ), wx.HORIZONTAL )

        bSizer993 = wx.BoxSizer( wx.VERTICAL )

        bSizer112 = wx.BoxSizer( wx.HORIZONTAL )


        bSizer112.Add( ( 10, 0), 1, 0, 5 )
```

```
        self.m_textCtrl10 = wx.TextCtrl( sbSizer82.GetStaticBox(), wx.ID_ANY,
wx.EmptyString, wx.DefaultPosition, wx.Size( 80,-1 ), wx.TE_RIGHT )
        bSizer112.Add( self.m_textCtrl10, 0, wx.ALL|wx.ALIGN_CENTER_VERTICAL,
5 )

        self.m_staticText39 = wx.StaticText( sbSizer82.GetStaticBox(),
wx.ID_ANY, u"Kg / m³", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText39.Wrap( -1 )

        bSizer112.Add( self.m_staticText39, 0,
wx.ALL|wx.ALIGN_CENTER_VERTICAL, 5 )


        bSizer993.Add( bSizer112, 1, wx.EXPAND, 5 )


        sbSizer82.Add( bSizer993, 1, wx.EXPAND, 5 )


        sbSizer6.Add( sbSizer82, 0, wx.EXPAND|wx.BOTTOM, 5 )


        bSizer8.Add( sbSizer6, 0, wx.EXPAND|wx.TOP|wx.BOTTOM, 8 )

        bSizer113 = wx.BoxSizer( wx.VERTICAL )

        bSizer114 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_button24 = wx.Button( self.pnl_parametros, wx.ID_ANY,
u"Salvar", wx.DefaultPosition, wx.Size( 70,-1 ), 0 )
        bSizer114.Add( self.m_button24, 0, wx.ALL, 5 )

        self.m_button25 = wx.Button( self.pnl_parametros, wx.ID_ANY,
u"Iniciar", wx.DefaultPosition, wx.Size( 70,-1 ), 0 )
        bSizer114.Add( self.m_button25, 0, wx.ALL, 5 )


        bSizer113.Add( bSizer114, 1, wx.ALIGN_CENTER, 5 )


        bSizer8.Add( bSizer113, 0, wx.EXPAND, 5 )


        self.pnl_parametros.SetSizer( bSizer8 )
        self.pnl_parametros.Layout()
        bSizer8.Fit( self.pnl_parametros )
        self.aba_principal.AddPage( self.pnl_parametros, u"Parametros", True )
        self.pnl_status = wx.Panel( self.aba_principal, wx.ID_ANY,
wx.DefaultPosition, wx.DefaultSize, wx.TAB_TRAVERSAL )
```

```
        bSizer64 = wx.BoxSizer( wx.VERTICAL )

        sbSizer9 = wx.StaticBoxSizer( wx.StaticBox( self.pnl_status,
wx.ID_ANY, u"Parametros" ), wx.VERTICAL )

        bSizer9911 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText6911 = wx.StaticText( sbSizer9.GetStaticBox(),
wx.ID_ANY, u"Valor kP", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText6911.Wrap( -1 )

        bSizer9911.Add( self.m_staticText6911, 0, wx.ALL, 5 )

        bSizer41111 = wx.BoxSizer( wx.VERTICAL )

        self.lb_status_kp = wx.StaticText( sbSizer9.GetStaticBox(), wx.ID_ANY,
u"-", wx.DefaultPosition, wx.Size( 50,-1 ), wx.ALIGN_RIGHT )
        self.lb_status_kp.Wrap( -1 )

        bSizer41111.Add( self.lb_status_kp, 0, wx.ALIGN_RIGHT|wx.ALL, 5 )


        bSizer9911.Add( bSizer41111, 1, 0, 5 )


        sbSizer9.Add( bSizer9911, 0, wx.EXPAND, 5 )

        bSizer9211 = wx.BoxSizer( wx.HORIZONTAL )

        self.lb_status_ki = wx.StaticText( sbSizer9.GetStaticBox(), wx.ID_ANY,
u"Valor kI", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.lb_status_ki.Wrap( -1 )

        bSizer9211.Add( self.lb_status_ki, 0, wx.ALL, 5 )

        bSizer3911 = wx.BoxSizer( wx.VERTICAL )

        bSizer3911.SetMinSize( wx.Size( 50,-1 ) )
        self.lb_status_ki = wx.StaticText( sbSizer9.GetStaticBox(), wx.ID_ANY,
u"-", wx.DefaultPosition, wx.Size( 50,-1 ), wx.ALIGN_RIGHT )
        self.lb_status_ki.Wrap( -1 )

        bSizer3911.Add( self.lb_status_ki, 0, wx.ALIGN_RIGHT|wx.ALL, 5 )


        bSizer9211.Add( bSizer3911, 1, 0, 5 )


        sbSizer9.Add( bSizer9211, 0, wx.EXPAND, 5 )
```

```
        bSizer9111 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText6111 = wx.StaticText( sbSizer9.GetStaticBox(),
wx.ID_ANY, u"Valor kD", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText6111.Wrap( -1 )

        bSizer9111.Add( self.m_staticText6111, 0, wx.ALL, 5 )

        bSizer4011 = wx.BoxSizer( wx.VERTICAL )

        self.lb_status_kd = wx.StaticText( sbSizer9.GetStaticBox(), wx.ID_ANY,
u"-", wx.DefaultPosition, wx.Size( 50,-1 ), wx.ALIGN_RIGHT )
        self.lb_status_kd.Wrap( -1 )

        bSizer4011.Add( self.lb_status_kd, 0, wx.ALIGN_RIGHT|wx.ALL, 5 )


        bSizer9111.Add( bSizer4011, 1, 0, 5 )


        sbSizer9.Add( bSizer9111, 0, wx.EXPAND, 5 )

        bSizer91112 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText61112 = wx.StaticText( sbSizer9.GetStaticBox(),
wx.ID_ANY, u"Setpoint", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText61112.Wrap( -1 )

        bSizer91112.Add( self.m_staticText61112, 0, wx.ALL, 5 )

        bSizer40112 = wx.BoxSizer( wx.VERTICAL )

        self.lb_status_setpoint = wx.StaticText( sbSizer9.GetStaticBox(),
wx.ID_ANY, u"-", wx.DefaultPosition, wx.Size( 50,-1 ), wx.ALIGN_RIGHT )
        self.lb_status_setpoint.Wrap( -1 )

        bSizer40112.Add( self.lb_status_setpoint, 0, wx.ALIGN_RIGHT|wx.ALL, 5
)


        bSizer91112.Add( bSizer40112, 1, 0, 5 )


        sbSizer9.Add( bSizer91112, 1, wx.EXPAND, 5 )


        bSizer64.Add( sbSizer9, 0, wx.EXPAND|wx.ALL, 5 )
```

```
        sbSizer91 = wx.StaticBoxSizer( wx.StaticBox( self.pnl_status,
wx.ID_ANY, u"Sensor Ultrassonico" ), wx.VERTICAL )

        bSizer99111 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText69111 = wx.StaticText( sbSizer91.GetStaticBox(),
wx.ID_ANY, u"Altura liquido", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText69111.Wrap( -1 )

        bSizer99111.Add( self.m_staticText69111, 0, wx.ALL, 5 )

        bSizer411111 = wx.BoxSizer( wx.VERTICAL )

        self.lb_status_alturaLiquido = wx.StaticText(
sbSizer91.GetStaticBox(), wx.ID_ANY, u"-", wx.DefaultPosition, wx.Size( 50,-1
), wx.ALIGN_RIGHT )
        self.lb_status_alturaLiquido.Wrap( -1 )

        bSizer411111.Add( self.lb_status_alturaLiquido, 0,
wx.ALL|wx.ALIGN_RIGHT, 5 )


        bSizer99111.Add( bSizer411111, 1, 0, 5 )


        sbSizer91.Add( bSizer99111, 0, wx.EXPAND, 5 )

        bSizer91111 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText61111 = wx.StaticText( sbSizer91.GetStaticBox(),
wx.ID_ANY, u"Volume", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText61111.Wrap( -1 )

        bSizer91111.Add( self.m_staticText61111, 0,
wx.ALL|wx.ALIGN_CENTER_VERTICAL, 5 )

        bSizer40111 = wx.BoxSizer( wx.VERTICAL )

        self.lb_status_volume = wx.StaticText( sbSizer91.GetStaticBox(),
wx.ID_ANY, u"-", wx.DefaultPosition, wx.Size( 50,-1 ), wx.ALIGN_RIGHT )
        self.lb_status_volume.Wrap( -1 )

        bSizer40111.Add( self.lb_status_volume, 0, wx.ALL|wx.ALIGN_RIGHT, 5 )


        bSizer91111.Add( bSizer40111, 1, 0, 5 )


        sbSizer91.Add( bSizer91111, 0, wx.EXPAND, 5 )
```

```
        self.btn_status_calibrarUltrassonico = wx.Button(
sbSizer91.GetStaticBox(), wx.ID_ANY, u"Calibrar", wx.DefaultPosition,
wx.DefaultSize, 0 )
        sbSizer91.Add( self.btn_status_calibrarUltrassonico, 0,
wx.ALL|wx.ALIGN_CENTER_HORIZONTAL, 5 )


        bSizer64.Add( sbSizer91, 0, wx.EXPAND|wx.ALL, 5 )

        sbSizer912 = wx.StaticBoxSizer( wx.StaticBox( self.pnl_status,
wx.ID_ANY, u"Célula de Carga" ), wx.VERTICAL )

        bSizer92111 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText62111 = wx.StaticText( sbSizer912.GetStaticBox(),
wx.ID_ANY, u"Altura líquido", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText62111.Wrap( -1 )

        bSizer92111.Add( self.m_staticText62111, 0,
wx.ALL|wx.ALIGN_CENTER_VERTICAL, 5 )

        bSizer39111 = wx.BoxSizer( wx.VERTICAL )

        self.lb_status_areaBase = wx.StaticText( sbSizer912.GetStaticBox(),
wx.ID_ANY, u"-", wx.DefaultPosition, wx.Size( 50,-1 ), wx.ALIGN_RIGHT )
        self.lb_status_areaBase.Wrap( -1 )

        bSizer39111.Add( self.lb_status_areaBase, 0, wx.ALL|wx.ALIGN_RIGHT, 5
)


        bSizer92111.Add( bSizer39111, 1, 0, 5 )


        sbSizer912.Add( bSizer92111, 0, wx.EXPAND, 5 )

        bSizer911112 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText611112 = wx.StaticText( sbSizer912.GetStaticBox(),
wx.ID_ANY, u"Volume", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText611112.Wrap( -1 )

        bSizer911112.Add( self.m_staticText611112, 0,
wx.ALL|wx.ALIGN_CENTER_VERTICAL, 5 )

        bSizer401112 = wx.BoxSizer( wx.VERTICAL )
```

```
        self.lb_status_volumeBalanca = wx.StaticText(
sbSizer912.GetStaticBox(), wx.ID_ANY, u"-", wx.DefaultPosition, wx.Size( 50,-1
), wx.ALIGN_RIGHT )
        self.lb_status_volumeBalanca.Wrap( -1 )

        bSizer401112.Add( self.lb_status_volumeBalanca, 0,
wx.ALL|wx.ALIGN_RIGHT, 5 )


        bSizer911112.Add( bSizer401112, 1, 0, 5 )


        sbSizer912.Add( bSizer911112, 0, wx.EXPAND, 5 )

        bSizer9111121 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText6111121 = wx.StaticText( sbSizer912.GetStaticBox(),
wx.ID_ANY, u"Peso", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText6111121.Wrap( -1 )

        bSizer9111121.Add( self.m_staticText6111121, 0,
wx.ALL|wx.ALIGN_CENTER_VERTICAL, 5 )

        bSizer4011121 = wx.BoxSizer( wx.VERTICAL )

        self.lb_status_pesoBalanca = wx.StaticText( sbSizer912.GetStaticBox(),
wx.ID_ANY, u"-", wx.DefaultPosition, wx.Size( 50,-1 ), wx.ALIGN_RIGHT )
        self.lb_status_pesoBalanca.Wrap( -1 )

        bSizer4011121.Add( self.lb_status_pesoBalanca, 0,
wx.ALL|wx.ALIGN_RIGHT, 5 )


        bSizer9111121.Add( bSizer4011121, 1, 0, 5 )


        sbSizer912.Add( bSizer9111121, 0, wx.EXPAND, 5 )

        bSizer921112 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_staticText621112 = wx.StaticText( sbSizer912.GetStaticBox(),
wx.ID_ANY, u"Densidade", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText621112.Wrap( -1 )

        bSizer921112.Add( self.m_staticText621112, 0,
wx.ALL|wx.ALIGN_CENTER_VERTICAL, 5 )

        bSizer391112 = wx.BoxSizer( wx.VERTICAL )
```

```
        self.lb_status_densidade = wx.StaticText( sbSizer912.GetStaticBox(),
wx.ID_ANY, u"-", wx.DefaultPosition, wx.Size( 50,-1 ), wx.ALIGN_RIGHT )
        self.lb_status_densidade.Wrap( -1 )

        bSizer391112.Add( self.lb_status_densidade, 0, wx.ALL|wx.ALIGN_RIGHT,
5 )


        bSizer921112.Add( bSizer391112, 1, 0, 5 )


        sbSizer912.Add( bSizer921112, 0, wx.EXPAND, 5 )

        self.btn_status_calibrarCelulaCarga = wx.Button(
sbSizer912.GetStaticBox(), wx.ID_ANY, u"Calibrar", wx.DefaultPosition,
wx.DefaultSize, 0 )
        sbSizer912.Add( self.btn_status_calibrarCelulaCarga, 0,
wx.ALL|wx.ALIGN_CENTER_HORIZONTAL, 5 )


        bSizer64.Add( sbSizer912, 1, wx.EXPAND|wx.ALL, 5 )

        self.m_button23 = wx.Button( self.pnl_status, wx.ID_ANY, u"Estimar
Densidade", wx.DefaultPosition, wx.DefaultSize, 0 )
        bSizer64.Add( self.m_button23, 0, wx.ALL|wx.ALIGN_CENTER_HORIZONTAL, 5
)


        self.pnl_status.SetSizer( bSizer64 )
        self.pnl_status.Layout()
        bSizer64.Fit( self.pnl_status )
        self.aba_principal.AddPage( self.pnl_status, u"Status", False )
        self.m_panel5 = wx.Panel( self.aba_principal, wx.ID_ANY,
wx.DefaultPosition, wx.DefaultSize, wx.TAB_TRAVERSAL )
        bSizer58 = wx.BoxSizer( wx.VERTICAL )

        bSizer59 = wx.BoxSizer( wx.VERTICAL )

        sbSizer92 = wx.StaticBoxSizer( wx.StaticBox( self.m_panel5, wx.ID_ANY,
u"Bomba" ), wx.VERTICAL )

        self.m_staticText31 = wx.StaticText( sbSizer92.GetStaticBox(),
wx.ID_ANY, u"Controle Manual Bomba", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText31.Wrap( -1 )

        sbSizer92.Add( self.m_staticText31, 0,
wx.ALIGN_CENTER_HORIZONTAL|wx.ALL, 5 )
```

```
        self.m_staticText311 = wx.StaticText( sbSizer92.GetStaticBox(),
wx.ID_ANY, u"0%", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText311.Wrap( -1 )

        sbSizer92.Add( self.m_staticText311, 0,
wx.ALIGN_CENTER_HORIZONTAL|wx.ALL, 5 )

        self.m_slider1 = wx.Slider( sbSizer92.GetStaticBox(), wx.ID_ANY, 0, 0,
100, wx.DefaultPosition, wx.DefaultSize, wx.SL_HORIZONTAL )
        sbSizer92.Add( self.m_slider1, 0, wx.ALL|wx.EXPAND, 5 )

        self.m_button14 = wx.Button( sbSizer92.GetStaticBox(), wx.ID_ANY,
u"Enviar", wx.DefaultPosition, wx.DefaultSize, 0 )
        sbSizer92.Add( self.m_button14, 0, wx.ALL|wx.EXPAND, 5 )

        bSizer62 = wx.BoxSizer( wx.HORIZONTAL )

        self.m_button15 = wx.Button( sbSizer92.GetStaticBox(), wx.ID_ANY,
u"Parar", wx.DefaultPosition, wx.Size( 75,-1 ), 0 )
        bSizer62.Add( self.m_button15, 0, wx.ALL, 5 )

        self.m_button16 = wx.Button( sbSizer92.GetStaticBox(), wx.ID_ANY,
u"Iniciar", wx.DefaultPosition, wx.Size( 75,-1 ), 0 )
        bSizer62.Add( self.m_button16, 0, wx.ALL, 5 )


        sbSizer92.Add( bSizer62, 1, 0, 5 )


        bSizer59.Add( sbSizer92, 1, 0, 5 )


        bSizer58.Add( bSizer59, 1, 0, 5 )

        bSizer60 = wx.BoxSizer( wx.VERTICAL )

        sbSizer83 = wx.StaticBoxSizer( wx.StaticBox( self.m_panel5, wx.ID_ANY,
u"Dados" ), wx.VERTICAL )

        self.m_staticText32 = wx.StaticText( sbSizer83.GetStaticBox(),
wx.ID_ANY, u"Visualizar leituras:", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_staticText32.Wrap( -1 )

        sbSizer83.Add( self.m_staticText32, 0, wx.ALIGN_CENTER|wx.ALL, 5 )

        bSizer63 = wx.BoxSizer( wx.VERTICAL )

        self.m_checkBox4 = wx.CheckBox( sbSizer83.GetStaticBox(), wx.ID_ANY,
u"Ultrassônico", wx.DefaultPosition, wx.DefaultSize, 0 )
```

```
        bSizer63.Add( self.m_checkBox4, 0, wx.ALIGN_LEFT|wx.ALL, 5 )

        self.m_checkBox5 = wx.CheckBox( sbSizer83.GetStaticBox(), wx.ID_ANY,
u"Balança", wx.DefaultPosition, wx.DefaultSize, 0 )
        self.m_checkBox5.SetValue(True)
        bSizer63.Add( self.m_checkBox5, 0, wx.ALIGN_LEFT|wx.ALL, 5 )

        self.m_checkBox6 = wx.CheckBox( sbSizer83.GetStaticBox(), wx.ID_ANY,
u"Média dos dados", wx.DefaultPosition, wx.DefaultSize, 0 )
        bSizer63.Add( self.m_checkBox6, 0, wx.ALIGN_LEFT|wx.ALL, 5 )


        sbSizer83.Add( bSizer63, 1, wx.ALIGN_CENTER_HORIZONTAL, 5 )

        self.m_button13 = wx.Button( sbSizer83.GetStaticBox(), wx.ID_ANY,
u"Limpar Dados", wx.DefaultPosition, wx.DefaultSize, 0 )
        sbSizer83.Add( self.m_button13, 0, wx.ALIGN_CENTER|wx.ALL, 5 )
        bSizer60.Add( sbSizer83, 1, wx.EXPAND, 5 )
        bSizer58.Add( bSizer60, 1, wx.EXPAND, 5 )
        bSizer58.Add( ( 0, 0), 1, wx.EXPAND, 5 )
        self.m_panel5.SetSizer( bSizer58 )
        self.m_panel5.Layout()
        bSizer58.Fit( self.m_panel5 )
        self.aba_principal.AddPage( self.m_panel5, u"Controle", False )

        bSizer2.Add( self.aba_principal, 1, wx.ALL|wx.EXPAND, 5 )


        bSizer1.Add( bSizer2, 0, wx.EXPAND|wx.TOP|wx.BOTTOM|wx.LEFT, 5 )

        bSizer3 = wx.BoxSizer( wx.VERTICAL )

        self.m_notebook2 = wx.Notebook( self, wx.ID_ANY, wx.DefaultPosition,
wx.Size( -1,-1 ), wx.NB_BOTTOM )
        self.m_panel6 = wx.Panel( self.m_notebook2, wx.ID_ANY,
wx.DefaultPosition, wx.Size( 600,500 ), wx.TAB_TRAVERSAL )
        self.m_notebook2.AddPage( self.m_panel6, u"Gráfico ", True )
        self.m_panel8 = wx.Panel( self.m_notebook2, wx.ID_ANY,
wx.DefaultPosition, wx.DefaultSize, wx.TAB_TRAVERSAL )
        bSizer42 = wx.BoxSizer( wx.VERTICAL )

        self.gr_planilhaDados = wx.grid.Grid( self.m_panel8, wx.ID_ANY,
wx.DefaultPosition, wx.DefaultSize, 0 )

        # Grid
        self.gr_planilhaDados.CreateGrid( 24, 8 )
        self.gr_planilhaDados.EnableEditing( True )
        self.gr_planilhaDados.EnableGridLines( True )
        self.gr_planilhaDados.EnableDragGridSize( True )
```

```
        self.gr_planilhaDados.SetMargins( 0, 0 )

        # Columns
        self.gr_planilhaDados.EnableDragColMove( False )
        self.gr_planilhaDados.EnableDragColSize( True )
        self.gr_planilhaDados.SetColLabelSize( 30 )
        self.gr_planilhaDados.SetColLabelAlignment( wx.ALIGN_CENTER,
wx.ALIGN_CENTER )

        # Rows
        self.gr_planilhaDados.EnableDragRowSize( True )
        self.gr_planilhaDados.SetRowLabelSize( 80 )
        self.gr_planilhaDados.SetRowLabelAlignment( wx.ALIGN_CENTER,
wx.ALIGN_CENTER )

        # Label Appearance

        # Cell Defaults
        self.gr_planilhaDados.SetDefaultCellAlignment( wx.ALIGN_LEFT,
wx.ALIGN_TOP )
        bSizer42.Add( self.gr_planilhaDados, 0, wx.ALL, 5 )


        self.m_panel8.SetSizer( bSizer42 )
        self.m_panel8.Layout()
        bSizer42.Fit( self.m_panel8 )
        self.m_notebook2.AddPage( self.m_panel8, u"Planilha de dados", False )

        bSizer3.Add( self.m_notebook2, 1, wx.EXPAND |wx.ALL, 5 )


        bSizer1.Add( bSizer3, 5, wx.EXPAND|wx.TOP|wx.BOTTOM|wx.RIGHT, 5 )


        self.SetSizer( bSizer1 )
        self.Layout()
        self.lb_statusBar = self.CreateStatusBar( 1, wx.STB_SIZEGRIP,
wx.ID_ANY )

        self.Centre( wx.BOTH )

        # Connect Events
        self.cb_portasDisponiveis.Bind( wx.EVT_CHOICE, self.onChoicePortCOM )
        self.Bind( wx.EVT_TOOL, self.onButtonRefreshCOM, id =
self.btn_transfer.GetId() )
        self.Bind( wx.EVT_TOOL, self.onButtonConnectCOM, id =
self.btn_check.GetId() )
        self.Bind( wx.EVT_TOOL, self.onButtonCancelCOM, id =
self.btn_cancel.GetId() )
```

```
        self.m_radioBtn1.Bind( wx.EVT_RADIOBUTTON, self.radioPorcentagem )
        self.m_radioBtn2.Bind( wx.EVT_RADIOBUTTON, self.radioLitros )
        self.m_radioBtn11.Bind( wx.EVT_RADIOBUTTON, self.radioUltrassonico )
        self.m_radioBtn21.Bind( wx.EVT_RADIOBUTTON, self.radioCelulaCarga )
        self.m_radioBtn211.Bind( wx.EVT_RADIOBUTTON, self.radioAmbos )
        self.m_button24.Bind( wx.EVT_BUTTON, self.onButtonSave )
        self.m_button25.Bind( wx.EVT_BUTTON, self.onButtonStart )
        self.btn_status_calibrarUltrassonico.Bind( wx.EVT_BUTTON,
self.onButtonCalibrarUltrassonico )
        self.btn_status_calibrarCelulaCarga.Bind( wx.EVT_BUTTON,
self.onButtonCalibrarCelulaCarga )
        self.m_button23.Bind( wx.EVT_BUTTON, self.onButtonDensidade )
        self.m_slider1.Bind( wx.EVT_SCROLL, self.onScroll )
        self.m_slider1.Bind( wx.EVT_SCROLL_CHANGED, self.onScrollChanged )
        self.m_button14.Bind( wx.EVT_BUTTON, self.onButtonSetPump )
        self.m_button15.Bind( wx.EVT_BUTTON, self.onButtonStopPump )
        self.m_button16.Bind( wx.EVT_BUTTON, self.onButtonStartPump )
        self.m_checkBox4.Bind( wx.EVT_CHECKBOX, self.onBoxDataUltra )
        self.m_checkBox5.Bind( wx.EVT_CHECKBOX, self.onBoxDataBalance )
        self.m_checkBox6.Bind( wx.EVT_CHECKBOX, self.onBoxBoth )
        self.m_button13.Bind( wx.EVT_BUTTON, self.onButtonLimparDados )

    def __del__( self ):
        pass
    # Virtual event handlers, overide them in your derived class
    def onChoicePortCOM( self, event ):
        event.Skip()

    def onButtonRefreshCOM( self, event ):
        event.Skip()

    def onButtonConnectCOM( self, event ):
        event.Skip()

    def onButtonCancelCOM( self, event ):
        event.Skip()

    def radioPorcentagem( self, event ):
        event.Skip()

    def radioLitros( self, event ):
        event.Skip()

    def radioUltrassonico( self, event ):
        event.Skip()

    def radioCelulaCarga( self, event ):
        event.Skip()
```

```
    def radioAmbos( self, event ):
        event.Skip()

    def onButtonSave( self, event ):
        event.Skip()

    def onButtonStart( self, event ):
        event.Skip()

    def onButtonCalibrarUltrassonico( self, event ):
        event.Skip()

    def onButtonCalibrarCelulaCarga( self, event ):
        event.Skip()

    def onButtonDensidade( self, event ):
        event.Skip()

    def onScroll( self, event ):
        event.Skip()

    def onScrollChanged( self, event ):
        event.Skip()

    def onButtonSetPump( self, event ):
        event.Skip()

    def onButtonStopPump( self, event ):
        event.Skip()

    def onButtonStartPump( self, event ):
        event.Skip()

    def onBoxDataUltra( self, event ):
        event.Skip()

    def onBoxDataBalance( self, event ):
        event.Skip()

    def onBoxBoth( self, event ):
        event.Skip()

    def onButtonLimparDados( self, event ):
        event.Skip()
```

requeriments.txt

```
numpy>=1.19.5
wxPython>=4.1.1
serial>=0.0.97
matplotlib>=3.3.1
matplot>=0.1.9
```