Pedro Henrique Oliveira Toscano Ximenes

Desenvolvimento de uma plataforma web para projetos de sistemas de controle lineares

Campina Grande, Brasil

28 de setembro de 2021

Pedro Henrique Oliveira Toscano Ximenes

Desenvolvimento de uma plataforma web para projetos de sistemas de controle lineares

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Orientador: Saulo Oliveira Dornellas Luiz, D. Sc.

Campina Grande, Brasil

28 de setembro de 2021

Pedro Henrique Oliveira Toscano Ximenes

Desenvolvimento de uma plataforma web para projetos de sistemas de controle lineares

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Trabalho aprovado em: 17/05/2021

Saulo Oliveira Dornellas Luiz, D. Sc.
Orientador

João Batista Morais dos Santos, D. Sc.
Convidado

Campina Grande, Brasil
28 de setembro de 2021

Dedico este trabalho aos meus pais, Rovênia Maria de Oliveira Toscano Ximenes e Carlos Alberto Toscano Ximenes.

Agradecimentos

Primeiramente gostaria de agradecer à minha família. Aos meus pais, minhas irmãs e à Tia por todo amor e por tudo que sempre fizeram por mim e aos meus cunhados por serem tão importantes na família e na minha caminhada até aqui. Sou grato por fazer parte dessa família.

Foram muitos anos difíceis percorridos até o fim desse trabalho. Contudo, teria sido muito mais difícil se não fossem algumas pessoas maravilhosas que tive o prazer de encontrar no caminho. À Caio, Camila e Ellen, obrigado por todos os momentos que vivemos e ajuda que sempre me deram, vocês são pessoas incríveis.

À André, Luan, Breno, Rodrigo, Rômulo, Zé, Arthur, Matheus, Saulo e vários outros grandes amigos que fiz, agradeço demais pela oportunidade de ter traçado essa jornada com vocês.

À Thiago, por ter passado cada dia dessa jornada ao meu lado, desde o dia 1 até o final. Por todas as noites de jogos, caronas, jantares, estudos e incontáveis outras coisas que vivemos. Você está em boa parte das minhas melhores memórias dessa jornada. Eu não teria conseguido sem você.

À Ana Paula, minha namorada e companheira, obrigado por me ajudar a superar os momentos difíceis e por fazer os momentos felizes ainda mais felizes. Sou grato por ter te conhecido e poder trilhar esse caminho ao seu lado.

À Àgui, Túlio, Marcelo e vários outros amigos que, mesmo não estando presentes fisicamente a todo momento, foram e são essenciais por incontáveis motivos, obrigado por tudo.

Por fim, ao Professor Saulo, que desde o meu segundo semestre até o último momento foi um orientador incrível, muito obrigado pelos ensinamentos.

Resumo

As ferramentas computacionais são grandes aliadas dos profissionais da engenharia, facilitando trabalhos e tornando possível a construção de projetos que, de outra forma, não seriam implementados. Muitas dessas ferramentas, contudo, não são acessíveis para todos, seja pelo fato de serem custosas financeira ou computacionalmente ou por serem de difícil utilização. O trabalho proposto tem como objetivo apresentar uma plataforma *web* de fácil utilização e gratuita no contexto de projetos de sistemas de controle. O desenvolvimento foi realizado em três etapas: 1) *back end*, utilizando Python e Flask; 2) *front end*, com Javascript e React e 3) infraestrutura, construída com Docker e recursos do Google Cloud Platform. A ferramenta foi construída de forma que possa ser acessada por qualquer navegador com acesso à internet e é capaz de fornecer ao usuário quatro informações úteis para projetos de sistemas de controle: Resposta ao Degrau do processo e do sistema em malha fechada; Diagramas de Bode e Lugar das Raízes. Foi apresentada a fundamentação teórica das ferramentas utilizadas e foi descrito todo o processo de desenvolvimento da plataforma e, por fim, foram apresentadas discussões sobre o resultado do projeto.

Palavras-chave: ferramentas computacionais, plataforma *web*, sistemas de controle, desenvolvimento *web*

Abstract

Computational tools are big allies of engineers, they facilitate work and make possible the development of projects that otherwise would not be built. A lot of these tools, however, are not available for the general public; some are expensive, hardware consuming or simply difficult to use. The work presented has the primary objective of introducing a free and easy to use web platform in the Control Systems context. It was developed in three steps: 1) back end, using Python and Flask; 2) front end with JavaScript and React and 3) infrastructure, built with Docker and Google Cloud Platform resources. The platform was built in a way that everyone with internet access can use it from any browser and it is capable of returning to the user four important information for control projects: Open-Loop Process and Closed-Loop System Step Responses, Bode Diagrams and Root Locus. The theoretical fundation was presented alongside with the description of the development process and, finally, discussions about the results.

Key-words: computational tools, web platform, control system, web development

Lista de ilustrações

Lista de abreviaturas e siglas

GCP	*Google Cloud Platform*
GCE	*Google Compute Engine*
GKE	*Google Kubernetes Engine*
IP	*Internet Protocol*
API	*Application Programming Interface*
HTTP	*Hypertext Transfer Protocol*
DNS	*Domain Name System*
URL	*Uniform Resource Locator*
JSON	*JavaScript Object Notation*
UI	*User Interface*
k8s	*Kubernetes*
HTML	*HyperText Markup Language*
CSS	*Cascading Style Sheets*

Sumário

1 Introdução

O auxílio de ferramentas computacionais é algo cada vez mais necessário na engenharia como um todo. Cálculos e simulações que podem demorar horas se feitos manualmente podem ser realizados em minutos utilizando as ferramentas corretas e, por isso, a computação é um dos maiores aliados do profissional da engenharia. Ferramentas extremamente poderosas podem ser encontradas nas mais diversas plataformas, para as mais diversas finalidades, tendo um constante crescimento em suas respectivas áreas. É impossível imaginar a realização de projetos de engenharia nos tempos modernos sem a existência desse tipo de instrumento.

Infelizmente, nem todas as ferramentas são acessíveis a todos. Muitas são caras, computacionalmente custosas ou simplesmente de difícil utilização. No caso de projetos de controle analógico, uma ferramenta bastante utilizada é o Control System Designer [1], que pode ser encontrada como *Toolbox* no *Matlab*. Nesse caso, trata-se de uma ferramenta bastante interessante, na qual diversas informações relevantes são disponibilizadas; por exemplo, é possível obter a função de transferência de um controlador para determinado sistema ao serem definidos alguns parâmetros pelo usuário, o que se mostra bastante útil no projeto de um sistema de controle. É possível, contudo, encaixar esse recurso nos três pontos de inacessibilidade descritos: o Matlab é um *software* financeiramente caro, além de exigir bastante poder computacional e pode ser de difícil utilização para muitas pessoas. Outro recurso com finalidade semelhante é o Sisotool [2], presente no pacote *control* disponível para a linguagem de programação Python. Apesar de gratuito, o Sisotool tem utilização complexa e requer um certo nível de conhecimento de programação para utilizá-lo.

A proposta deste trabalho é de minimizar os problemas encontrados nos recursos já existentes e desenvolver uma ferramenta gratuita, de fácil utilização e capaz de atender satisfatoriamente a necessidade dos usuários. O sistema a ser desenvolvido se propõe a entregar uma aplicação *web* na qual os usuários possam acessar a qualquer momento desde que haja conexão à internet, não sendo necessário qualquer tipo de instalação ou *download*. Ao acessar, o usuário deve conseguir facilmente navegar pelo *site* e fornecer as informações necessárias para que receba uma resposta adequada.

O principal objetivo é, portanto, apresentar um desenvolvimento consolidado de uma aplicação capaz de ajudar usuários em suas pesquisas e que possa ser facilmente aprimorado no futuro, de modo que esteja em constante aprimoramento, atendendo necessidades dos usuários e apresentando novas funcionalidades. O projeto foi nomeado temporariamente como *Project Achilles*.

2 Fundamentação Teórica

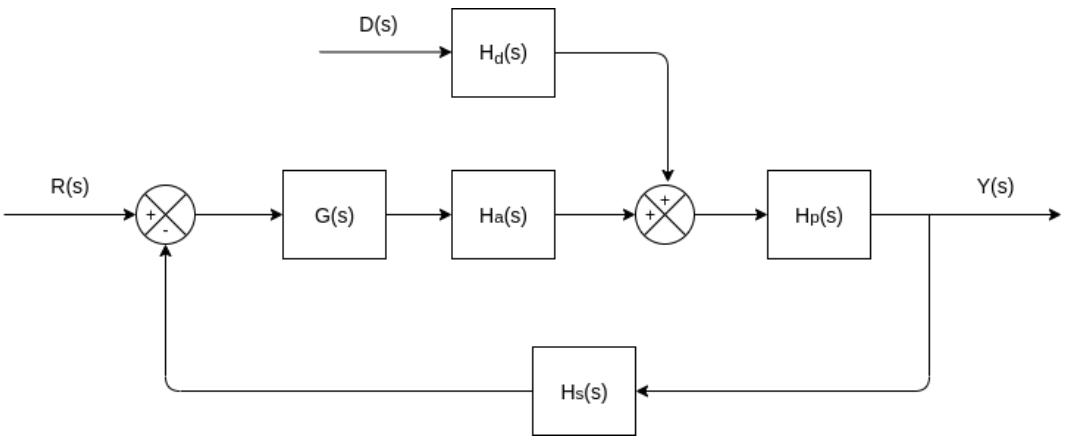
Neste capítulo são apresentados os princípios da Teoria de Controle necessários para o desenvolvimento do projeto em questão.

2.1 Sistemas de Controle

O primeiro passo para a análise de um sistema de controle é a obtenção de um modelo matemático para o sistema. Uma vez obtido esse modelo, é possível analisar o desempenho do sistema a partir dos vários métodos disponíveis (OGATA, 2010). Um sistema de controle pode apresentar diversas subsistemas, cada um com seu próprio modelo matemático, e pode ser representado graficamente por meio de um diagrama de blocos, como pode ser observado em um modelo genérico apresentado na Figura 1, no qual são representadas as transformadas de Laplace dos sinais de entrada $R(s)$, de perturbação $D(s)$ e de saída $Y(s)$, e as funções de transferência dos componentes:

- $G(s)$: controlador
- $H_a(s)$: atuador
- $H_p(s)$: planta a ser controlada
- $H_s(s)$: sensor
- $H_d(s)$: perturbação

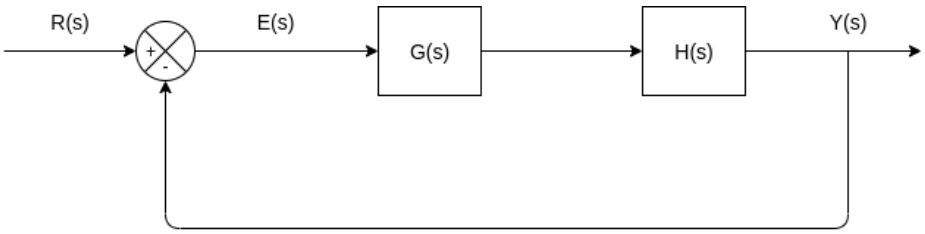
Figura 1 – Diagrama de blocos de um sistema de controle em malha fechada.



Fonte: autoria própria, 2021

Contudo, o trabalho em questão não leva em consideração a influência de perturbações externas, trabalhando apenas com sistemas com a entrada de referência $R(s)$ e, por isso, é possível desconsiderar o ramo de perturbação do diagrama de blocos. Além disso, também é possível simplificar ainda mais o digrama, sem perda de generalidade, considerando que a associação em série entre planta e atuador é reduzida a um único processo ($H_p(s)H_a(s) = H(s)$) e que o sensor não apresenta influência no sinal de realimentação, ou seja, considera-se que $H_s(s) = 1$. O diagrama resultante é apresentado na Figura 2, em que $E(s)$ representa o sinal de erro.

Figura 2 – Diagrama de blocos simplificado.



Fonte: autoria própria, 2021

Utilizando esse último diagrama, é possível montar uma função de transferência de malha fechada, ou seja, a relação entre a transformada de Laplace da variável de saída e a transformada de Laplace da variável de entrada, com todas as condições iniciais supostas iguais a zero (BISHOP, DORF, 2009). Dessa forma, é possível desenvolver a equação a partir da relação entre os três sinais apresentados:

$$E(s) = R(s) - Y(s) \tag{2.1}$$

$$Y(s) = E(s)G(s)H(s)$$

$$E(s) = \frac{Y(s)}{G(s)H(s)} \tag{2.2}$$

Substituindo (2.2) em (2.1):

$$R(s) - Y(s) = \frac{Y(s)}{G(s)H(s)}$$

$$\frac{Y(s)}{R(s)} = \frac{G(s)H(s)}{1 + G(s)H(s)} \tag{2.3}$$

Portanto, a função de transferência de malha fechada do diagrama genérico simplificado apresentado pode ser representada como mostrado em (2.3).

2.1.1 Lugar das Raízes

Uma das análises realizadas no trabalho se trata do lugar das raízes, técnica específica que mostra como mudanças em um dos parâmetros do sistema irá modificar as raízes da equação característica. O lugar das raízes é usado normalmente para o estudo do efeito da variação de um ganho de malha (FRANKLIN, et al., 2013).

Utilizando (2.3), pode-se definir sua equação característica como:

$$1 + G(s)H(s) = 0 \tag{2.4}$$

Segundo Franklin, assume-se que a equação característica pode ser definida por meio de componentes polinomiais $a(s)$ e $b(s)$ e, assim, defini-se a função de transferência $L(s) = \frac{b(s)}{a(s)}$. Desse modo (2.4) pode ser reescrita como (2.5).

$$1 + KL(s) = 0 \tag{2.5}$$

O lugar das raízes é, portanto, o gráfico de todas as possíveis raízes da equação (2.5) relativo ao parâmetro K (FRANKLIN, et al., 2013).

2.1.2 Diagramas de Bode

Outra forma de análise de sistemas, a resposta em frequência, significa a resposta em regime permanente de um sistema para uma entrada senoidal. Nos métodos de resposta em frequência, variamos a frequência do sinal de entrada dentro de certo intervalo e estudamos a resposta resultante. No projeto de um sistema de malha fechada, ajustamos as características da resposta em frequência em função da função de transferência de malha aberta para obter características aceitáveis da resposta transitória do sistema em malha fechada (OGATA, 2010).

Assim, é possível construir os diagramas de Bode que, segundo Ogata, são constituídos por dois gráficos: o primeiro é um gráfico logarítmico do módulo de uma função de transferência senoidal e o segundo é o gráfico do ângulo de fase. Ambos são traçados em escala linear em relação à frequência em escala logarítmica.

2.2 Tecnologias Utilizadas

O desenvolvimento do projeto pode ser identificado em três etapas. Primeiro, existe um servidor que é responsável pela lógica envolvida no processo de receber a entrada do usuário e transformar essa informação em dados para o projeto do sistema de controle: respostas ao degrau, diagramas de Bode e lugar das raízes. Essa etapa do desenvolvimento é chamada de *back end* e funciona como o servidor da aplicação.

Segundo, a etapa chamada de *front end* funciona como uma interface *web* entre o usuário e o servidor, sendo uma aplicação *client-side*, ou seja, do lado do cliente e não do servidor. Trata-se de uma forma amigável para que o usuário consiga enviar informações para o servidor sem precisar de comandos específicos ou outra forma mais complicada de comunicação. Essa é a forma principal de comunicação do cliente. Na página principal da aplicação, o usuário deve informar as funções de transferência do processo a ser controlado e do controlador. Essa informação é então enviada para o servidor que retorna os vetores necessários para construção dos gráficos, que também é feito no lado do cliente.

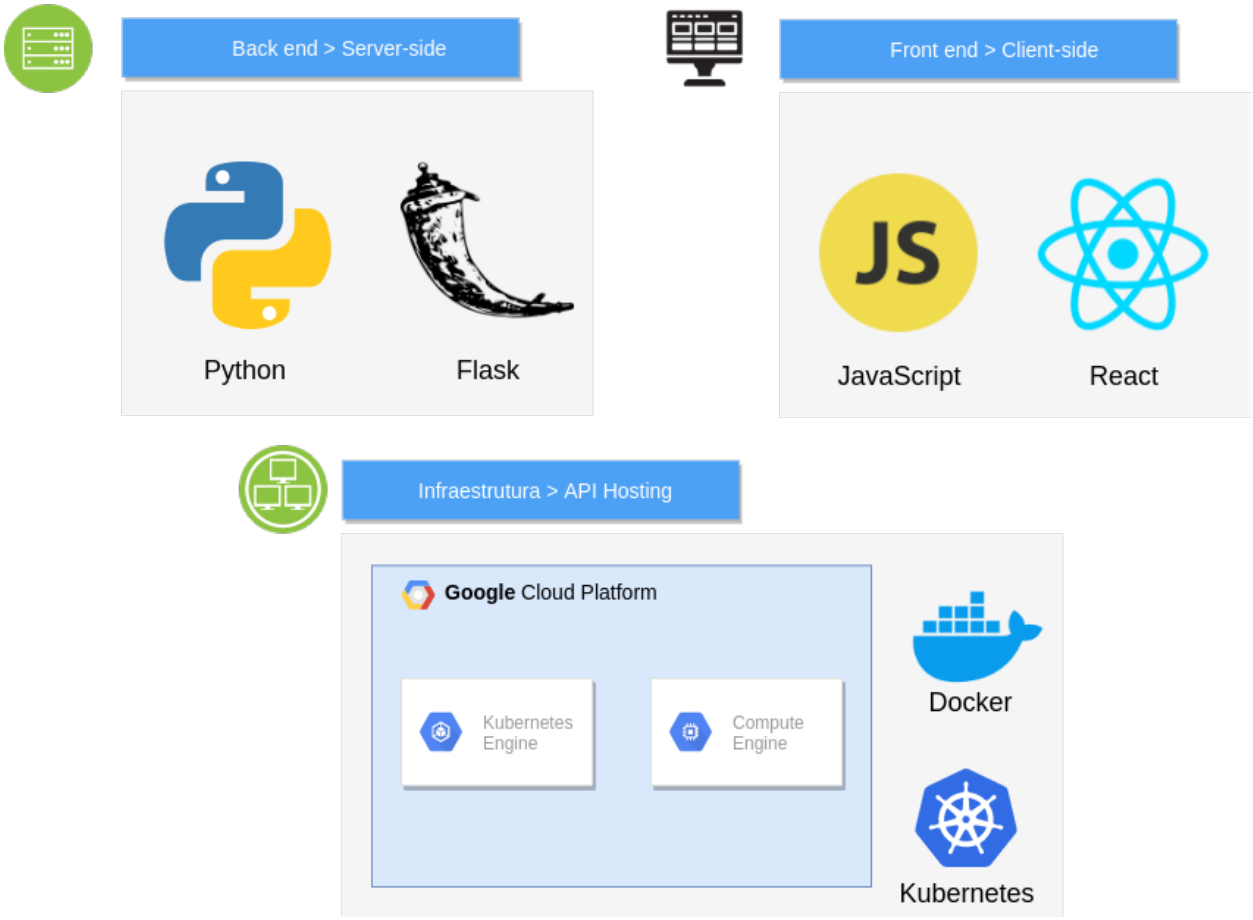
Por fim, as duas aplicações são empacotadas em contêineres e importadas em máquinas virtuais, nas quais são executadas ininterruptamente. Assim, o endereço IP (*Internet Protocol*) público da máquina virtual é exposto de forma que é possível acessa-lo de qualquer rede e de qualquer dispositivo, democratizando o acesso à plataforma. Para que o acesso não seja necessariamente realizado via IP, um servidor DNS (*Domain Name System*) é utilizado para que a URL de acesso apresente um nome comum, sendo mais amigável ao usuário.

O funcionamento de cada componente, entretanto, passa por uma série de etapas e utiliza diversas tecnologias. Os maiores detalhes de como cada componente funciona e como eles são interligados serão exploradas futuramente. Nessa seção, o foco será explicar cada tecnologia utilizada e o motivo de sua escolha no projeto, são elas:

- *Back end*
  - *Python*
  - *Flask*
- *Front end*
  - *JavaScript*
  - *React*
- Infraestrutura
  - *Docker*
  - *Kubernetes*
  - *Google Cloud Platform*
    - *Compute Engine*
    - *Kubernetes Engine*

Na Figura 3 são ilustradas as tecnologias acima citadas.

Figura 3 – Visão geral das tecnologias utilizadas.



Fonte: autoria própria, 2021

2.2.1 Linguagens de Programação

Para o desenvolvimento desse trabalho foram utilizadas duas linguagens de programação: *Python* e *JavaScript*. A primeira foi utilizada no desenvolvimento do *back end*, escolhida por possuir o pacote *control*, necessário para geração dos gráficos obtidos e a segunda no desenvolvimento do *front end*.

2.2.2 Flask

Flask é um pequeno *framework*, pequeno o suficiente para ser chamado de micro *framework* (GRINBERG, 2018), desenvolvido em *Python* para criação de aplicações *web* escritas também em *Python*. O objetivo principal dessa tecnologia é permitir de forma simples, rápida e flexível o desenvolvimento da aplicação. No projeto em questão, o *Flask* é utilizado para a criação do lado do servidor (*server-side* em inglês) e para a criação de rotas (ou *endpoints*), ou seja, especificações que reagem de maneira determinada.

Utilizando o exemplo de um *site* na *internet* bastante conhecido, o *Google*, é pos-

sível perceber que, ao acessar sua página principal, a URL (*Uniform Resource Locator*) em questão é *https://www.google.com/*. Ao clicar na opção "Estou com sorte", a URL é redirecionada para *https://www.google.com/doodles*. Ainda, se o próprio usuário digitar, por exemplo, *https://www.google.com/home*, é redirecionado para a página principal da loja da companhia. No exemplo, as extensões "*/*", "*/doodles*" e "*/home*" são rotas (*end-points)* e cada uma representa um objetivo específico. No contexto do projeto, as rotas do *back end* são criadas utilizando o framework *Flask*.

A comunicação com cada rota é feita utilizando requisições HTTP (*Hypertext Transfer Protocol*), que pode ser definido como um protocolo de comunicação entre máquinas que permite a transferência de hiper-texto, como o próprio nome sugere. Nesse protocolo, existem diversos tipos de requisições que podem ser feitas entre uma máquina e outra (cliente e servidor). Os principais tipos de requisição são:

- **GET**: operação simples de consulta. O cliente, ao realizar uma requisição do tipo *GET* simplesmente solicita ao servidor a informação contida naquela rota.
- **POST**: operação de envio de informações. Ao requisitar um *POST*, o cliente envia informações ao servidor, que tratará os dados da forma definida. Um exemplo de requisição *POST* é a simples pesquisa em um *site* de buscas. O usuário envia a informação e o servidor, por sua vez, utiliza essa informação para realizar a busca e devolver uma resposta ao cliente.
- **PUT**: operação de modificação. No caso do *PUT*, o cliente envia informações que devem ser modificadas no lado do servidor. Por exemplo, ao trocar a senha em uma rede social, o usuário realiza uma operação PUT, enviando a nova senha que substituirá a anterior no banco de dados.
- **DELETE**: operação de deletar algo do banco de dados do servidor. Por exemplo, ao excluir a conta de uma rede social, o usuário realizará uma requisição *DELETE*.

No projeto desenvolvido, são definidos os tipos de requisição *GET* e *POST*. No caso, o usuário deve informar as funções de transferência do processo a ser controlado e do controlador, o que acontece por meio de uma requisição do tipo *POST*. Os dados são tratados e utilizados nos cálculos necessários, por fim, o servidor retorna um objeto JSON (*JavaScript Object Notation*), que por sua vez será tratado no *front end* para a geração dos gráficos.

2.2.3 React

Sendo uma das principais ferramentas atuais para o desenvolvimento *web*, o *React* se trata de uma biblioteca da linguagem de programação *JavaScript* para criação de inter-

faces de usuário (comumente chamado de UI, *user interface*, em inglês) em projetos *web*. Desenvolvida e mantida por engenheiros do *Facebook*, a biblioteca se tornou *open-source* em 2013 e vem crescendo cada vez mais ao longo dos anos, sendo bastante utilizada em diversas aplicações, desde simples interfaces para pequenos projetos até interfaces complexas de grandes empresas multinacionais como o próprio *Facebook*, *Instagram*, *Netflix* e *Microsoft* (em muitos dos seus recursos).

Apesar de existirem diversas outras ferramentas de desenvolvimento *front end* como *Angular* e *Vue.js*, o *React* foi escolhido por sua versatilidade e por ser uma ferramenta muito poderosa. O desenvolvimento normalmente é feito a partir de pequenas e isoladas partes de código chamadas de *components*, permitindo que o uso do *React* seja feito em apenas algum detalhe da interface ou até mesmo construir UIs bastante complexas utilizando apenas a biblioteca.

A comunicação com o servidor *back end* pode ser feita pelo usuário enviando diretamente comandos de requisição sem precisar de uma interface. Porém, essa forma de comunicação é impraticável para qualquer aplicação *web* que seja voltada para clientes externos. O desenvolvimento *front end* realizado no projeto tem a finalidade de criar uma camada de comunicação entre o servidor e o usuário, que funciona como uma aplicação no lado do cliente, como já explicado.

Assim, existe um novo servidor cuja finalidade é realizar requisições para o servidor *back end* em nome do cliente, assim como receber as respostas do servidor e montar uma nova resposta para o usuário. No caso, o *React* é utizado para a criação da tela interativa e criação das rotas do *front end*. Além disso, quando recebe a resposta da requisição enviada, é utilizado o *Victory*, um conjunto de módulos e componentes para *React*, para a criação dos gráficos apresentados.

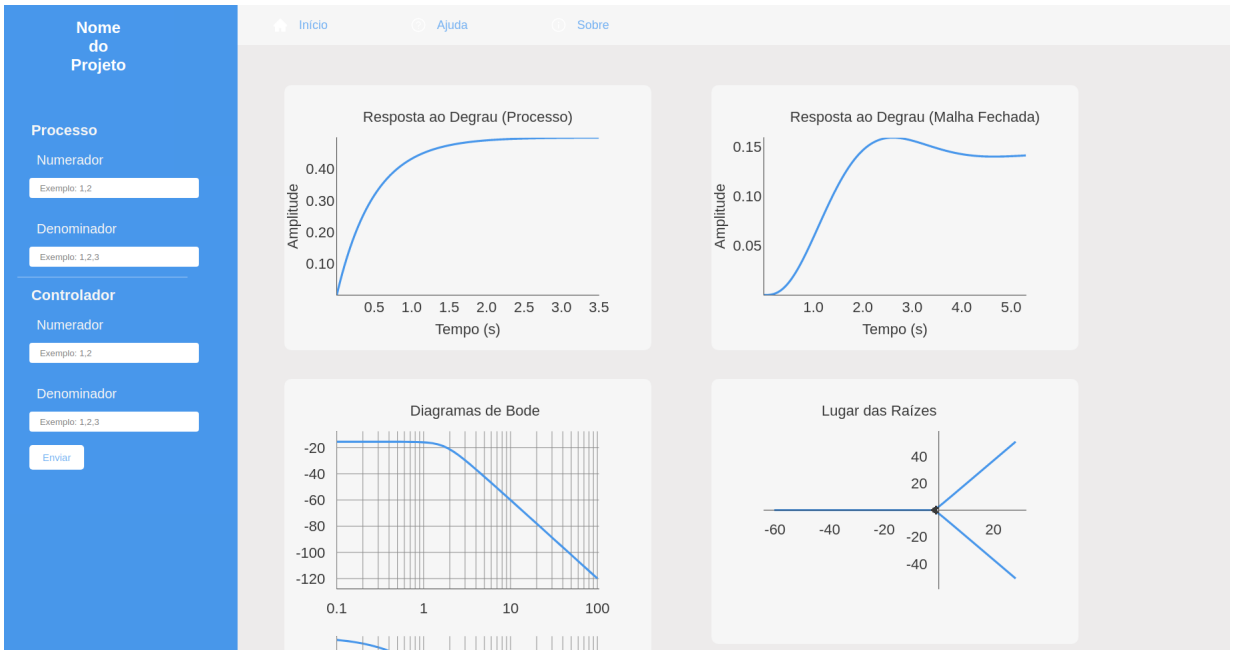
Na Figura 4, é apresentada a tela principal após o recebimento da resposta do servidor e geração dos gráficos.

2.2.4 Docker

A conteinerização de aplicações, também chamada de contentorização, é o processo de empacotamento e distribuição de *software* que serve de base para a integração de desenvolvimento de *software*. Como ocorre sem virtualização de *hardware*, os contêineres são pacotes portáteis e de alto desempenho que conservam as dependências e as bibliotecas com suas versões. Com essa utilização, o código empacotado pode ser executado consistentemente sem dependência do sistema operacional, facilitando assim a gestão do projeto em diversos ambientes.

Tendo em vista que se trata de uma tecnologia recente e em plena ascensão, novas ferramentas vão sendo criadas para oportunizar a conteinerização de forma cada vez mais

Figura 4 – Captura de tela da interface do projeto após a criação dos gráficos.



Fonte: autoria própria, 2021

acessível para os usuários. No presente trabalho, é apresentado o funcionamento de uma dessas ferramentas: o Docker.

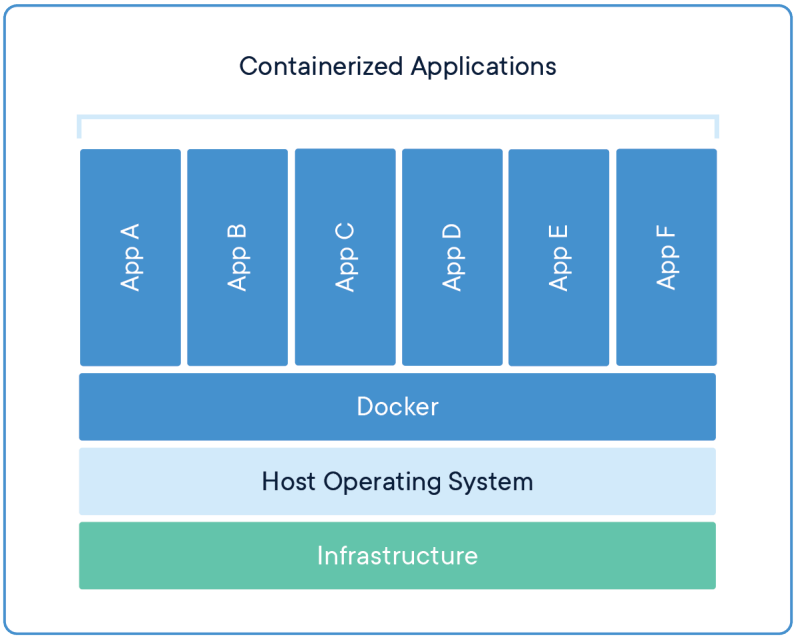
O Docker, *software* desenvolvido inicialmente em 2013, é uma plataforma *open-source* que simplifica a utilização da conteinerização de aplicações. Através do modelo cliente-servidor, o servidor é chamado de Docker *daemon*, responsável pela gestão dos contêineres, enquanto o cliente é uma interface da linha de comando (ILC) que interage com o servidor através de pedidos à API com o *kernel* do hospedeiro compartilhado (REIS, 2017). Na Figura 5 é esboçada a estrutura de aplicações conteinerizadas.

No projeto em questão, o Docker foi utilizado para a conteinerização das duas aplicações envolvidas (*back end* e *front end*), sendo criadas duas imagens *docker* que são empurradas para repositórios públicos, de forma que qualquer pessoa com o *Docker* instalado em seu computador pode baixar e utilizar as imagens.

2.2.5 Kubernetes

Também conhecido como k8s, o Kubernetes é um sistema de orquestração de aplicações conteinerizadas para automatização de *deployments* (subir uma nova versão da aplicação para o ambiente de produção, ou seja, ambiente de utilização dos clientes), escalonamento de infraestrutura e gerenciamento de aplicações. Trata-se de uma das tecnologias mais em alta no mercado e, ao contrário do que muitas vezes é pensado, não é um concorrente do *Docker*, mas na verdade são tecnologias que andam lado a lado.

Figura 5 – Estrutura de aplicações conteinerizadas.



Fonte: Docker. Disponível em: https://www.docker.com/resources/what-container. Acesso em: 13 mai. 2021.

O k8s funciona com a criação de *clusters* podendo conter vários nós com aplicações. Além disso, a ferramenta apresenta diversas outras funcionalidades como a "auto cura", que reinicia contêineres que falham ou cancelam aqueles que podem prejudicar a saúde do nó, e a alocação de endereços *IPv4* e *IPv6* para os serviços.

2.2.6 Google Cloud Platform

O *Google Cloud Platform* (GCP) é um serviço de computação em nuvem oferecido pelo *Google* que provisiona diversos serviços para o usuário, tais como Infraestrutura como Serviço (*Infrastructure as a Service, IaaS*), Plataforma como Serviço (*Platform as a Service*, PaaS), Função como Serviço (*Function as a Service*, FaaS), além de diversos outros recursos. Alguns dos principais e mais famosos recursos do GCP são os seguintes:

- **BigQuery**: o *Google Cloud Platform* oferece a infraestrutura necessária para armazenamento e consulta de massivos blocos de dados, deixando o usuário livre de qualquer preocupação com infraestrutura, assim como boa parte dos recursos no GCP. É, portanto, o ambiente em que os dados são armazenados em tabelas relacionais e consultados via SQL.
- **Google Cloud Storage**: assim como o *BigQuery*, o *Storage* é um recurso voltado para armazenamento. Porém, diferente do anterior que armazena dados em forma de tabelas, o *Storage* armazena arquivos.

- **Dataflow**: recurso que oferece uma solução unificada para processamento de dados em lote (*batch*) e *streaming*.
- **Cloud Functions**: esse recurso se enquadra na categoria FaaS e oferece toda a infraestrutura operacional necessária para executar códigos escritos em diferentes ambientes, como *Python* 3.8, *Java* 11, *Go 1.13* e *Node.js* 12 (no momento da escrita deste trabalho). Além disso, as *functions* podem ser gatilhadas por algum evento específico. Por exemplo, uma das possibilidades de gatilho é o *Cloud Storage*, de modo a executar o código contido na *Cloud Function* sempre que algum arquivo é inserido em determinada pasta.

No projeto desenvolvido nesse trabalho, contudo, foram utilizados dois recursos da plataforma, são eles: o *Google Compute Engine* (GCE) e o *Google Kubernetes Engine* (GKE). Assim como todos os recursos oferecidos no GCP, a utilização do GCE e do GKE tem um custo financeiro e, por isso, foi utilizado o período de licença grátis oferecido pela companhia.

2.2.6.1 Google Compute Engine

De forma geral, o Google Compute Engine é um recurso de IaaS e serve, como sugerido pela categoria em que se encaixa, como um serviço que entrega infraestrutura para o usuário, no caso do GCE, máquinas virtuais. São oferecidos diversos modelos de máquina, de modo a atender clientes sem a necessidade de um *hardware* muito poderoso, até mesmo usuários que necessitam de máquinas com altíssimo poder computacional, seja qual for seu propósito.

O Google Compute Engine é utilizado nesse trabalho como o serviço que oferece as máquinas responsáveis por executar as duas aplicações desenvolvidas.

2.2.6.2 Google Kubernetes Engine (GKE)

O Recurso de Kubernetes como Serviço (*Kubernetes as a Service*, KaaS) da plataforma se trata de um recurso integrado com o Google Compute Engine no qual é possível criar nós de aplicações, gerenciar em que máquina essas aplicações irão rodar, expor o IP para uma rede externa, gerenciar aplicações conteinerizadas e diversas outras funcionalidades.

Além disso, o GKE garante a segurança da rede, oferece um serviço de *load balancer*, ou seja, distribui automaticamente as requisições que chegam na aplicação nos recursos disponíveis. Em outras palavras, se a infraestrutura atual não suportar a quantidade de requisições que chegam no momento, o *load balancer* deve, por exemplo, criar novas instâncias de máquinas virtuais e distribuir a carga total entre elas. Também existe

a funcionalidade "piloto automático" em que todas essas configurações podem ser pré-definidas e o GKE se responsabiliza por comandá-las.

No trabalho em questão, o GKE é utilizado para criar e buscar as imagens *Docker* no repositório do *Dockerhub* e criar um nó para cada aplicação, alocando uma máquina virtual e um *load balancer*. Além disso, o IP público do nó da aplicação *front end* é exposto à rede externa, de modo que seja possível acessar o *site* de qualquer rede, em qualquer dispositivo.

3 Desenvolvimento

Nesse capítulo, é descrito o processo de desenvolvimento do projeto. Serão abordados detalhes sobre cada etapa (*back end*, *front end* e *infraestrutura*), explicando as principais partes dos códigos e como os componentes se conectam para formar a solução proposta nesse trabalho. O código fonte do projeto está disponível em um repositório público no *Github* e pode ser encontrado no endereço eletrônico *https://github.com/PedroXimenes/project-achilles*.

3.1 Servidor

Como já explicado no capítulo anterior, o desenvolvimento do *back end* foi realizado por meio da linguagem de programação Python e com auxílio do micro *framework* Flask. A aplicação aqui criada tem como objetivo determinar os resultados que serão mostrados ao usuário na interface gráfica. Para isso, são utilizados, principalmente, os seguintes pacotes do Python:

- control 0.8.4
- numpy 1.19.5
- scipy 1.5.4

Nesse componente do projeto existem dois arquivos principais: o *app.py* e o *controller.py*. O primeiro é responsável por definir as rotas e o que cada rota retorna de acordo com a requisição recebida. O segundo tem por objetivo calcular e retornar as informações via construção de um objeto *json*.

O trecho de código apresentado a seguir se refere à parte principal do arquivo *app.py*. De forma simples, as rotas são definidas utilizando o *@app.route*, além de definir o tipo de requisição que é suportado pelo *endpoint* (a requisição padrão é o *GET*). A rota principal, que é chamada pelo cliente, se chama "*/analysis*", na qual os métodos *GET* e *POST* são definidos. Ao receber uma requisição do tipo *POST*, o objeto *json* enviado é carregado e tratado para que a informação enviada pelo cliente possa ser compreendida. Cada parte da requisição (numerador e denominador das funções de transferência do processo e do controlador) é transformada em uma lista e passada para a função *calculate* que é definida no arquivo *controller.py*.

app.py

```python
from flask import Flask, request
import json
from controller import calculate
from flask_cors import CORS

# Initialize Flask
app = Flask(__name__)
CORS(app)

@app.route('/')
@app.route('/home')
def home():
    return {"result": "Project Achilles"}

@app.route('/health')
def health():
    return {"alive": True}

@app.route('/analysis', methods=["GET", "POST"])
def analysis():
    if request.method == "POST":
        data = json.loads(request.data)

        hnum = list(map(float,data['hnum'].split(',')))
        hden = list(map(float,data['hden'].split(',')))
        gnum = list(map(float,data['gnum'].split(',')))
        gden = list(map(float,data['gden'].split(',')))

        resp = analysis1(hnum, hden, gnum, gden)

        return resp

    return "Analysis"
```

Já o arquivo *controller.py*, o outro arquivo principal do servidor, apresenta quatro funções: *calculate*, *stepResponse*, *bodeDiagram* e *rootLocus*. A função *calculate* (apresentada a seguir) é responsável por chamar as outras três funções e organizar as informações resultantes em um objeto *json* e retorná-las para que a rota devolva uma resposta à requisição.

calculate - controller.py

```
def calculate(hnum=None, hden=None, gnum=None, gden=None):
    h_t, h_y, g_t, g_y, series = stepResponse(hnum, hden,
                                    gnum, gden)
    mag, phase, omega = bodeDiagram(series)
    real, imag, klist = rootLocus(series)
    poles = control.pole(series)
    zeros = control.zero(series)
    num_poles = len(poles)

    fb_data = {
                "x_axis_ol": h_t.tolist(),
                "y_axis_ol": h_y.tolist(),
                "hnum": hnum,
                "hden": hden,
                "x_axis_cl":    g_t.tolist(),
                "y_axis_cl":   g_y.tolist(),
                "omega":          omega.tolist(),
                "magnitude":      mag.tolist(),
                "phase":          phase.tolist(),
                "root_real": real.tolist(),
                "root_imag": imag.tolist(),
                "root_gain": klist.tolist(),
                "num_poles": num_poles,
                "zeros": zeros.tolist()
                }
    data = json.dumps(fb_data)

    return data
```

A função *stepResponse* tem por finalidade calcular a resposta ao degrau tanto do processo em malha aberta quanto do sistema de controle em malha fechada. Para isso, utiliza os pacotes *scipy* e *control*.

stepResponse - controller.py

```
1  def stepResponse(hnum=None, hden=None, gnum=None, gden=None):
2      #Process open loop step response
3      sys = signal.TransferFunction(hnum, hden)
4      t, y = signal.step(sys)
5
6      # Calculate Step Response for Closed Loop System
7      system = control.tf(hnum, hden)
8      controller = control.tf(gnum, gden)
9      series = control.series(system, controller)
10     fb = control.feedback(series, 1, -1)
11
12     T, Y = control.step_response(fb)
13
14     return t, y, T, Y, series
```

Em sequência, a função *bodeDiagram* calcula os vetores de magnitude, fase e frequência dos diagramas de Bode e aplica a transformação da magnitude para decibéis e da fase para graus.

bodeDiagram - controller.py

```
1  def bodeDiagram(series=None):
2      magnitude, phase, omega = control.bode(series, dB=True)
3      mag_db = control.mag2db(magnitude)
4      phase_deg = np.rad2deg(phase)
5
6      return mag_db, phase_deg, omega
```

Por fim, a função *rootLocus* calcula os vetores contendo as partes real e imaginária dos pontos do Lugar das Raízes, além do ganho correspondente. Ainda, é necessário aplicar uma transformação nos dados para que sejam utilizados no *front end* de forma mais eficiente e, por isso, dois novos vetores são formados antes de serem retornados.

rootLocus - controller.py

```
def rootLocus(series=None):
    rlist, klist = control.root_locus(series)

    total, shape = rlist.shape
    index = 0
    j = 0
    real = np.zeros((total, shape))
    imag = np.zeros((total,shape))

    for x in rlist:
        for i in x:
            if j == shape:
                j = 0
            real[index][j] = i.real
            imag[index][j] = i.imag

            j = j + 1
        index = index + 1

    return real, imag, klist
```

3.2 Cliente

Inicialmente, o *front end* da aplicação seria construído de forma simples, utilizando apenas HTML (*HyperText Markup Language*) e CSS (Cascading Style Sheets) que seriam aplicados diretamente no *back end*, ou seja, as páginas criadas dessa forma seriam diretamente assimiladas às rotas criadas pelo *Flask*. Contudo, essa é uma abordagem bastante simplória e poderia não se mostrar suficiente à medida que o projeto fosse expandido, já que possui grandes limitações. Apesar de teoricamente ser possível construir grandes projetos usufruindo apenas dessas duas ferramentas, o desenvolvimento pode se tornar bastante complexo conforme a complexidade do projeto também aumenta. Assim, optou-se pelo desenvolvimento de uma aplicação cliente utilizando *JavaScript* como linguagem de programação e o *React*, que proporcionam muito mais controle e poder de construção de interfaces de forma mais simples.

No diretório principal, é possível encontrar a pasta *front*, em que tudo relacionado à essa etapa do desenvolvimento se encontra, tendo sua estrutura de diretórios descrita a seguir:

- project-achilles
  - front
    - build
    - nginx
    - node_modules
    - public
    - src
      - components

A primeira etapa do desenvolvimento trata-se de criar o projeto *React* padrão. Com isso, grande parte dos arquivos e dos diretórios encontrados no projeto são importados nessa etapa. Para o desenvolvimento em si, os principais arquivos são o *src/App.js*, *index.css* e todos os arquivos encontrados na pasta *src/components*, descritos a seguir:

- src
  - App.js
  - index.css
  - components
    - BodeDiagram.js
    - Loading.js
    - NavbarData.js
    - RootLocus.js
    - Sidebar.js
    - StepResponse.js

Cada um desses arquivos presentes no diretório *components* representam os componentes da aplicação, ou seja, são partes isoladas de código que podem ser utilizadas também isoladamente ou juntas para criação de uma interface. O arquivo *App.js* serve de maneira semelhante ao arquivo *app.py* presente na aplicação *back end*, servindo como principal organizador do projeto, criando rotas e contendo a lógica principal.

App.js

Nessa seção será explicado o desenvolvimento do arquivo *App.js*, cujo código pode ser encontrado no Apêndice A. Nesta seção, tal arquivo será apresentado por meio do Algoritmo 1. Após as importações necessárias, a função *App()* é definida. Nela, primeiramente são definidas as variáveis: *systemAnalysis*, que é responsável por guardar as informações

recebidas pela resposta do servidor; *showChartAnalysis*, variável booleana que habilita ou desabilita a presença dos gráficos; e *showLoading*, que habilita o símbolo de carregamento. O retorno dessa função está descrito em *React JSX*, que é uma extensão do *JavaScript* que nos permite definir elementos do React com uma sintaxe de *template* diretamente no código JavaScript (BANKS, PORCELLO, 2020). Dessa forma, são definidas as rotas (apenas a rota principal "/"é de fato implementada) e o que deve ser renderizado em cada uma. Na rota principal, o componente "*Sidebar*" sempre é chamado, enquanto os componentes *Loading*, *StepResponse*, *BodeDiagram* e *RootLocus* são renderizados caso as variáveis *showLoading* ou *showChartAnalysis* estejam habilitadas.

Algorithm 1: App.js

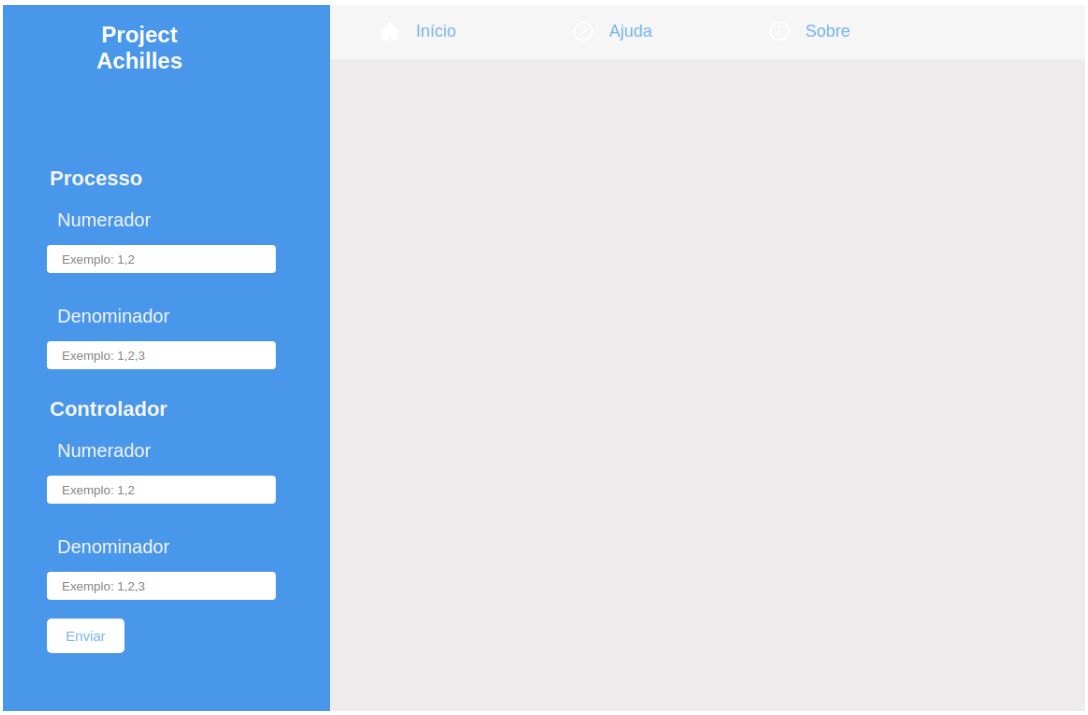
```
importações;
Function App()
    set systemAnalysis, showChartAnalysis, showLoading;
    if load is true then
        showChartAnalysis = true;
        showLoading = true;
    end
    wait resposta do servidor then;
    systemAnalysis = server data;
    showLoading = false;
    showChartAnalysis = true;
    route '/' render:
    show component Sidebar;
    if showLoading is true then
        show component Loading ;
    end
    if showChartAnalysis is true then
        show component StepResponse;
        show component BodeDiagram;
        show component RootLocus;
    end
end
```

No caso, *load* será verdadeiro quando o usuário enviar uma informação para o servidor, o que é definido no componente *Sidebar* e permanecerá com esse estado até que uma resposta seja recebida, quando *showChartAnalysis* se torna verdadeiro.

Sidebar.js

O componente *Sidebar*, definido no arquivo *Sidebar.js*, representa a construção da barra lateral e da barra superior da página principal (cujos dados são buscados no componente *NavbarData*, encontrado no Apêndice F), sendo as únicas informações visíveis até que o usuário informe o sistema a ser analisado. Um recorte dessa visão é mostrado na Figura 6.

Figura 6 – Recorte da interface do projeto antes da criação dos gráficos.



Fonte: autoria própria, 2021

O código completo desse componente pode ser encontrado no Apêndice B. Nesse componente, também, são realizadas duas verificações. Caso o usuário não digite algum dos quatro campos obrigatórios, será retornada uma mensagem de erro. O mesmo acontece caso seja incluído algo diferente de números em algum dos campos: vírgulas, pontos ou sinais. Além disso, no momento em que o usuário clica no botão "enviar", as informações por ele digitadas são coletadas e enviadas via requisição para o servidor.

Loading.js

Possivelmente o componente mais simples da aplicação, o *Loading*, presente no arquivo *Loading.js*, é responsável apenas por criar o símbolo dinâmico de carregamento que aparecerá no intervalo entre o envio da requisição e obtenção da resposta. Por ser bastante simples e curto, o código é inteiramente mostrado a seguir.

Loading.js

```
import React from 'react'
import Loader from 'react-loader-spinner';
import "react-loader-spinner/dist/loader/css/react-spinner-loader.css";
const Loading = () => {
    return (
        <div className="loading">
            <Loader  type="TailSpin" color="#549ff5e1" height={100} width={100} />
        </div>
    )
}

export default Loading
```

StepResponse.js, BodeDiagram.js e RootLocus.js

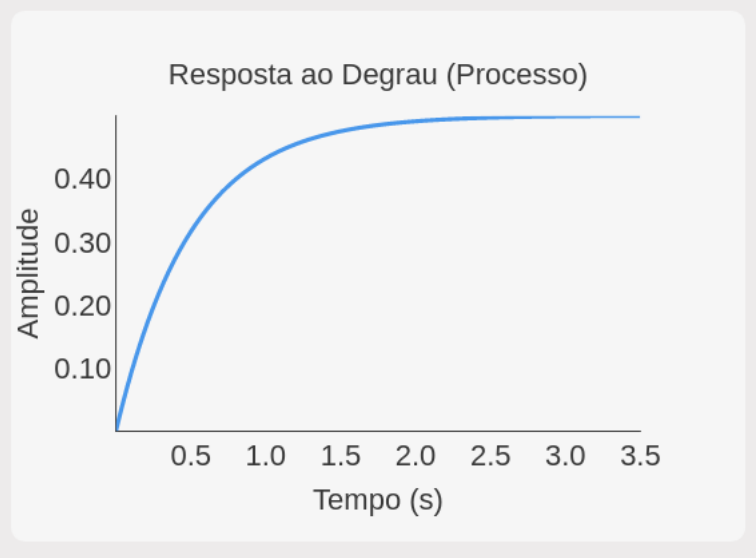
Três dos componentes principais do projeto em questão apresentam a mesma proposta, porém são construídos de maneiras diferentes, são eles: *StepResponse*, *BodeDiagram* e *RootLocus.js*, definidos respectivamente nos arquivos *StepResponse.js* (Apêndice C), *BodeDiagram.js* (Apêndice D) e *RootLocus.js* (Apêndice E).

As informações retornadas pelo servidor são utilizadas, principalmente nesses três componentes. Todos utilizam a biblioteca *Victory* para a criação dos gráficos. Com ela, são definidos os dados, o formato, eixos e diversos outros aspectos de sua criação. Utilizando o elemento *VictoryVoronoiContainer* do *Victory*, foi possível, também, adicionar o reconhecimento de eventos do *mouse*, exibindo as informações do gráfico em cada ponto que o usuário passar o cursor. Além disso, o componente *RootLocus* também faz uso do *VictoryZoomContainer*, que permite o zoom em qualquer ponto do gráfico utilizando a roleta do *mouse*.

O componente *StepResponse* utiliza as informações de eixos x e y do processo em malha aberta e sistema de controle em malha fechada para construir os gráficos "Resposta ao Degrau (Processo)"e "Resposta ao Degrau (Malha Fechada)", como pode ser visto nas Figuras 7 e 8 respectivamente.

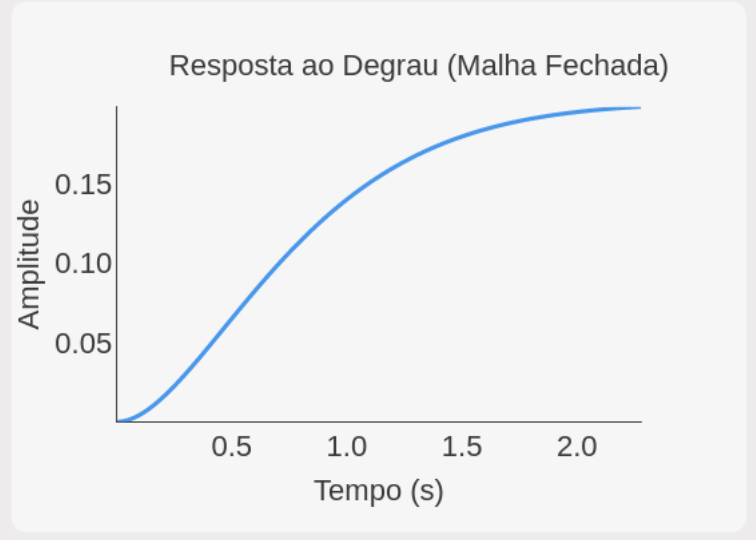
Já o componente *BodeDiagram* utiliza as informações de magnitude, fase e frequência retornadas pelo servidor para criação dos dois gráficos semi-logarítmicos que compõem o diagrama e que podem ser observados na Figura 9.

Figura 7 – Resposta ao Degrau do Processo gerado pela plataforma.



Fonte: autoria própria, 2021

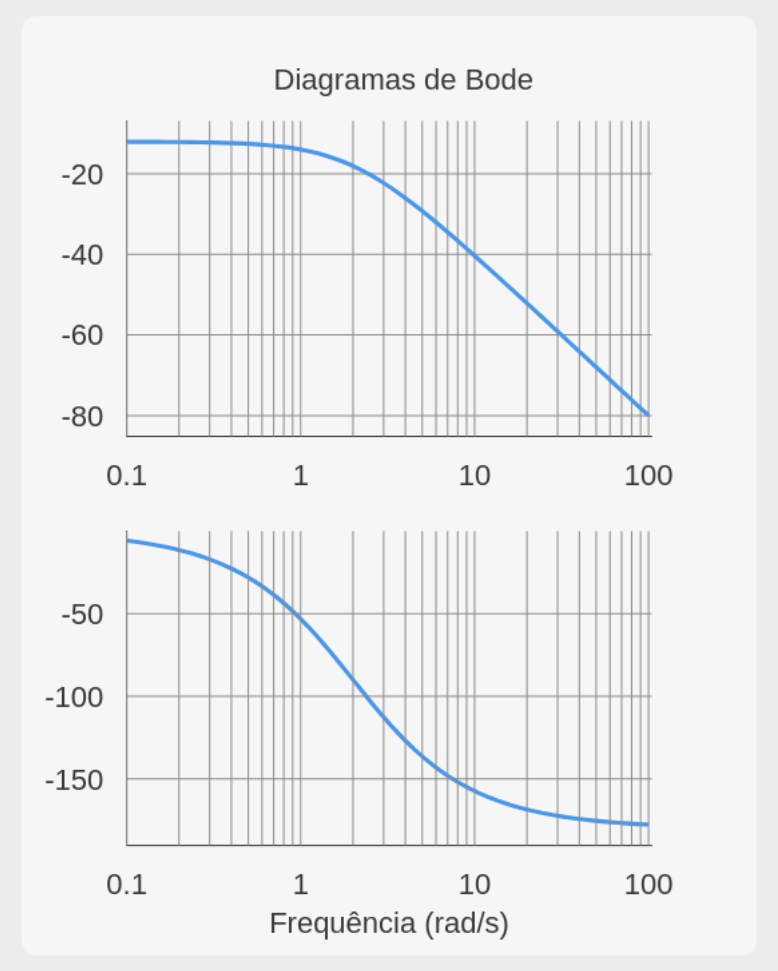
Figura 8 – Resposta ao Degrau do Sistema em Malha Fechada gerado pela plataforma.



Fonte: autoria própria, 2021

Por fim, o componente *RootLocus* utiliza as informações de número de polos, vetor do eixo real, vetor do eixo imaginário e ganho para construção do gráfico. Os polos são calculados com as informações passadas e indicados no gráfico com um sinal de "+", o que deve ser alterado futuramente para um "x". A informação com os zeros do sistema também é passada pelo servidor, porém não é implementada, o que também deve ser incluído como trabalho futuro.

Figura 9 – Diagramas de Bode gerados pela plataforma.



Fonte: autoria própria, 2021

index.css

O index.css é o principal arquivo CSS do projeto. Nele, são definidas as "classes", que podem ser utilizadas tanto em arquivos HTML ou por outras fontes, como no caso do trabalho em questão, em que é utilizado no JSX do React. Com o CSS é possível definir estilos (como cor, fundo, tempo de transição e diversos outros aspectos) ao documento web. O trecho de código a seguir representa a definição de estilo loading:

```
1 .loading{
2   position: fixed;
3   top: 50%;
4   left: 50%;
5 }
```

De forma simples, o trecho define que a classe *loading* tem uma posição fixa na tela e está posicionada no centro, a uma distância de 50% em relação à margem superior e 50% em relação à margem esquerda. Já no componente *Loading*, cujo código foi mostrado

nesse capítulo, pode-se perceber que, ao iniciar o elemento "*div*", é explicitado o className "*loading*", ou seja, esse "*div*"cujo símbolo de carregamento é definido herda as caraterísticas definidas na classe *loading* presente no index.css. O mesmo princípio é válido para todos os outros componentes.

3.3 Infraestrutura

A primeira etapa da infraestrutura é a de conteinerização das aplicações utilizando a ferramenta *Docker*. Para isso, é necessário criar um arquivo chamado *Dockerfile* correspondente a cada uma delas. O *Dockerfile* é o arquivo em que são definidas as condições para a criação da imagem *docker* do contêiner, ou seja, o arquivo que será utilizado para a criação do contêiner.

Dockerfile - servidor

```
1  FROM python:3
2
3  COPY ./requirements.txt .
4
5  ADD ./controller.py .
6  ADD ./app.py .
7
8  RUN pip3 install -r requirements.txt
9
10 EXPOSE 5000
11
12 CMD ["python3", "app.py"]
```

O código acima se refere ao *Dockerfile* do servidor. Primeiro, a imagem base do Python é carregada, logo em seguida são carregados os arquivos principais e o comando de instalar as dependências presentes no arquivo *requirements.txt* é executado. Em seguida, a porta 5000 (porta padrão do Flask) é exposta pelo contêiner e é definido um comando a ser executado quando o próprio contêiner for executado, no caso, o comando "*python3 app.py*" que inicializará o servidor. De forma semelhante, é criado um *Dockerfile* para a aplicação cliente, que pode ser encontrado no apêndice G.

Em seguida, é feito o *build* de cada arquivo para a criação das imagens *docker*, que são incluídas em seus respectivos repositórios remotos, sendo eles os seguintes:

pedroximenes/nome-do-projeto-back para a aplicação *back end* e *pedroximenes/nome-do-projeto-front*, ambos encontrados no *Dockerhub* (*https://hub.docker.com/*).

A partir do momento em que as imagens são incluídas no *Dockerhub*, é possível iniciar a próxima etapa da criação da infraestrutura que consiste basicamente em criar os recursos necessários no *Google Kubernetes Engine* que está presente no *Google Cloud Platform*. Para isso, foi utilizado o *Cloud Shell*, ferramenta de linha de comando do GCP disponível diretamente no console da plataforma. Primeiramente, é necessário criar o *cluster*, no qual é organizada a estrutura das máquinas *workers* que serão posteriormente criadas. A criação do *cluster* é feita com o seguinte comando:

```
gcloud container clusters create project-achilles
```

Com o *cluster* criado, o próximo passo é fazer o *deploy* da aplicação conteinerizada. O comando a seguir foi utilizado para criação da aplicação do servidor *back end* e um processo semelhante é feito para o *front end*, apenas ajustando a nomenclatura. As imagens são carregadas diretamente do repositório do *Dockerhub*.

```
kubectl create deployment project-achilles-back –image=pedroximenes/project-achilles-back:latest
```

Assim, uma máquina virtal no *Google Compute Engine* é criada e ela executará o contêiner cuja imagem foi relatada no comando de criação. Em seguida, é necessário criar um *Kubernetes Service*, que é um recurso do *Kubernetes* que permite a exposição da aplicação para o tráfego da rede externa, o que é feito por meio do seguinte comando:

```
kubectl expose deployment project-achilles-front –type=LoadBalancer –port 3000
```

Apenas a aplicação cliente, *front end*, é exposta à rede externa. Dessa forma, é gerado um IP externo que pode ser acessado por qualquer usuário.

4 Resultados e Discussões

Esse capítulo é dedicado à apresentação dos resultados finais obtidos por meio do desenvolvimento do projeto, analisando-os criticamente e expondo possíveis pontos de melhoria. Para fins de validação, será feito uma comparação entre os resultados obtido por este trabalho (*Project Achilles*) e pelo *software Matlab*. São analisados os seguintes gráficos: resposta ao degrau do processo em malha aberta e do sistema de controle em malha fechada, diagramas de Bode e lugar das raízes.

4.1 Exemplo 1

O primeiro exemplo foi obtido da documentação de uma *toolbox* do *Matlab* chamada *Control System Designer*, no qual o projeto do controlador é disponibilizado. Os dados inseridos são os seguintes.

- Processo: $\frac{1}{s+1}$
- Controlador: $\frac{s+2.02}{s}$

Resposta ao Degrau - Processo

Os gráficos gerados nas duas plataformas são mostrados nas Figuras 10 e 11. É possível perceber a clara semelhança entre os gráficos, mostrando que ambos disponibilizam a mesma informação.

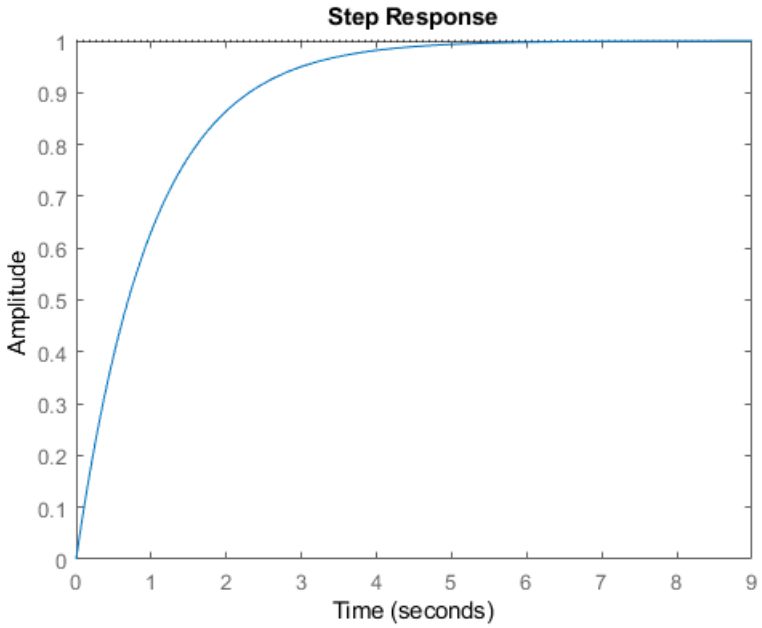
Resposta ao Degrau - Sistema de controle em Malha Fechada

A resposta ao degrau do sistema de controle em malha fechada é apresentada nas Figuras 12 e 13. Apesar de ser perceptível que o gráfico gerado pelo *Project Achilles* apresenta uma amostra menor de dados no eixo temporal, percebe-se que os dois gráficos, mais uma vez, são muito semelhantes e os valores são condizentes.

Diagramas de Bode

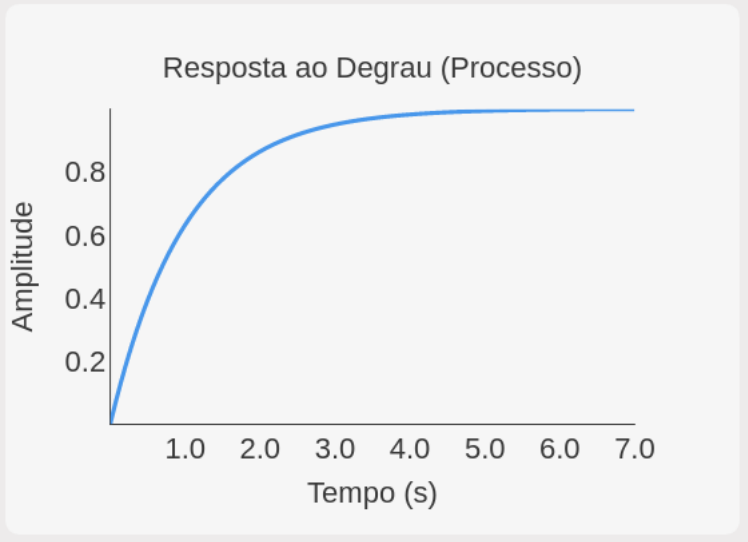
Nas Figuras 14 e 15 são representados os Diagramas de Bode obtidos pelo Matlab e pelo *Project Achilles* respectivamente.

Figura 10 – Resposta ao Degrau do Processo em malha aberta gerado pelo Matlab.



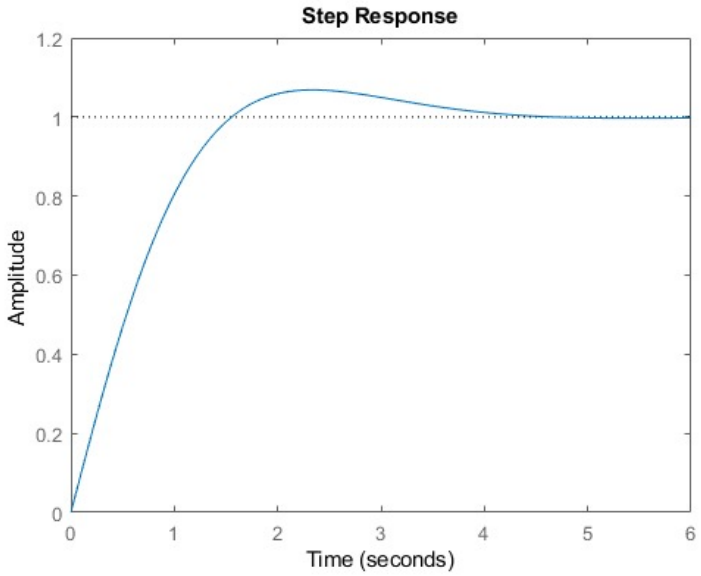
Fonte: autoria própria, 2021

Figura 11 – Resposta ao Degrau do Processo em malha aberta gerado pelo *Project Achilles*.



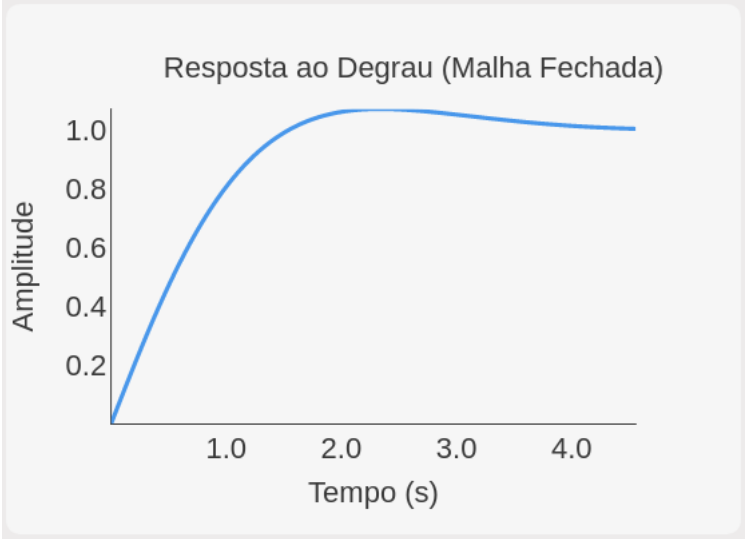
Fonte: autoria própria, 2021

Figura 12 – Resposta ao Degrau do sistema de controle em malha fechada gerado pelo Matlab.



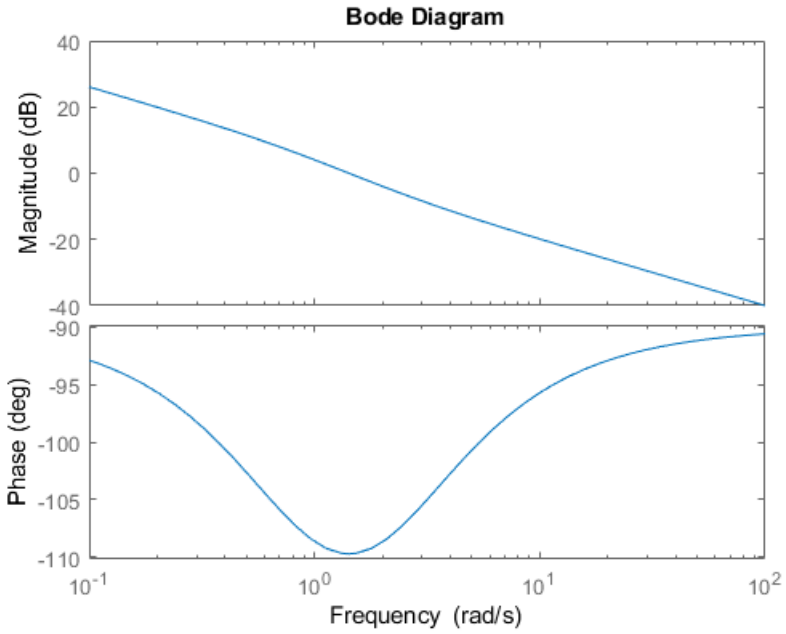
Fonte: autoria própria, 2021

Figura 13 – Resposta ao Degrau do sistema de controle em malha fechada gerado pelo *Project Achilles*.



Fonte: autoria própria, 2021

Figura 14 – Diagrama de Bode gerado pelo Matlab.

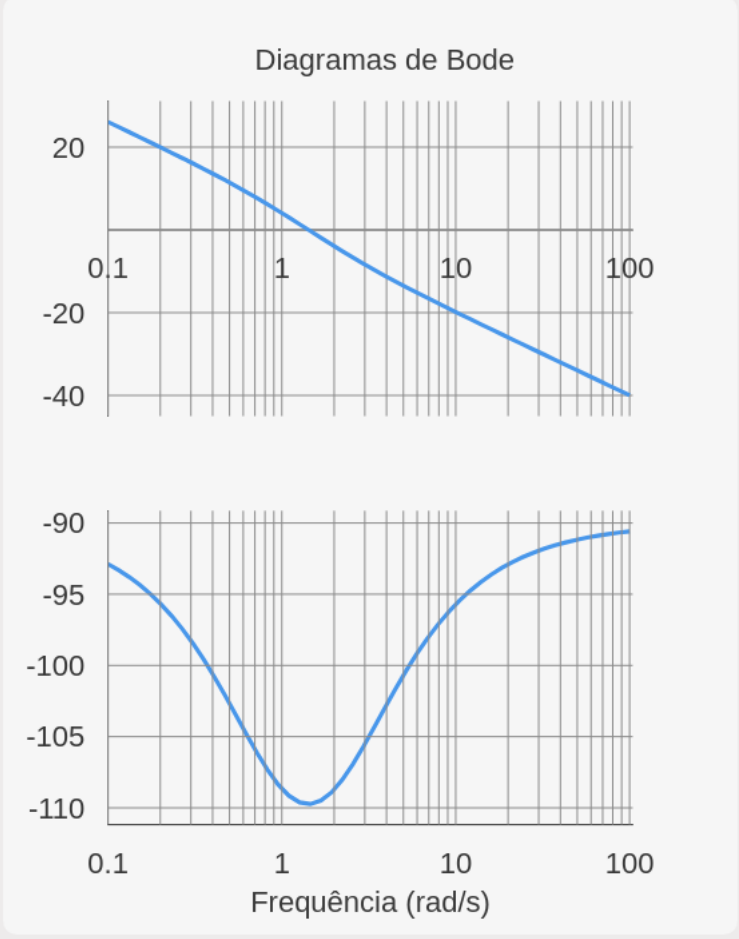


Fonte: autoria própria, 2021

Lugar das Raízes

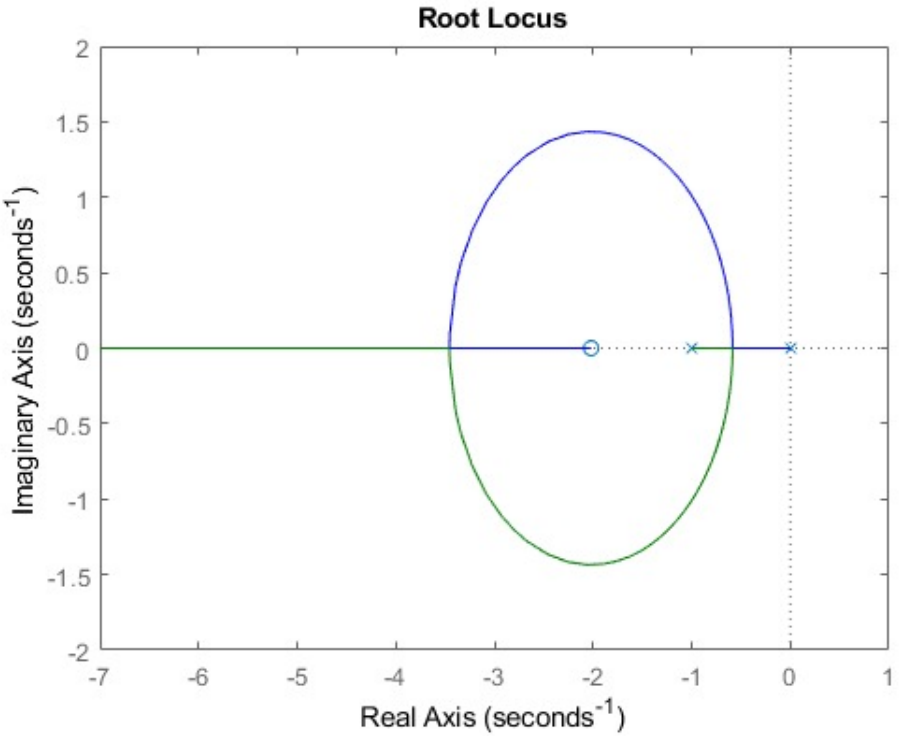
O lugar das raízes, mostrado nas Figuras 16 e 17, é o gráfico onde a diferença entre as duas plataformas é perceptível. Notavelmente, o gráfico gerado pelo *Project Achilles* apresenta falhas na construção do gráfico ao não gerar um círculo perfeito, dispondo de vários pontos de descontinuidade. Além disso, não possui indicação dos zeros. A indicação dos polos é marcada com um sinal de "+"ao invés de um "x"e o eixo imaginário não apresenta rótulo.

Figura 15 – Diagrama de Bode gerado pelo *Project Achilles*.



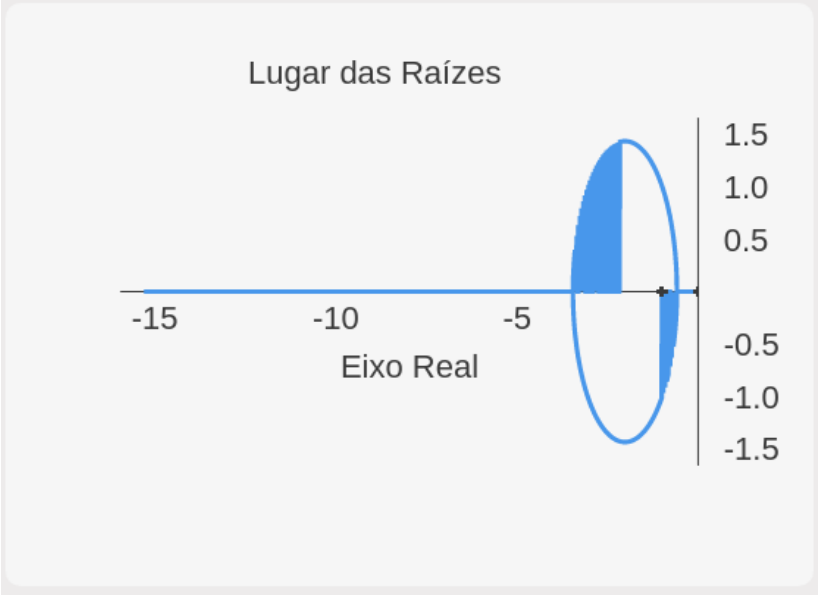
Fonte: autoria própria, 2021

Figura 16 – Lugar das Raízes gerado pelo Matlab.



Fonte: autoria própria, 2021

Figura 17 – Lugar das Raízes gerado pelo Project Achilles.



Fonte: autoria própria, 2021

4.2 Exemplo 2

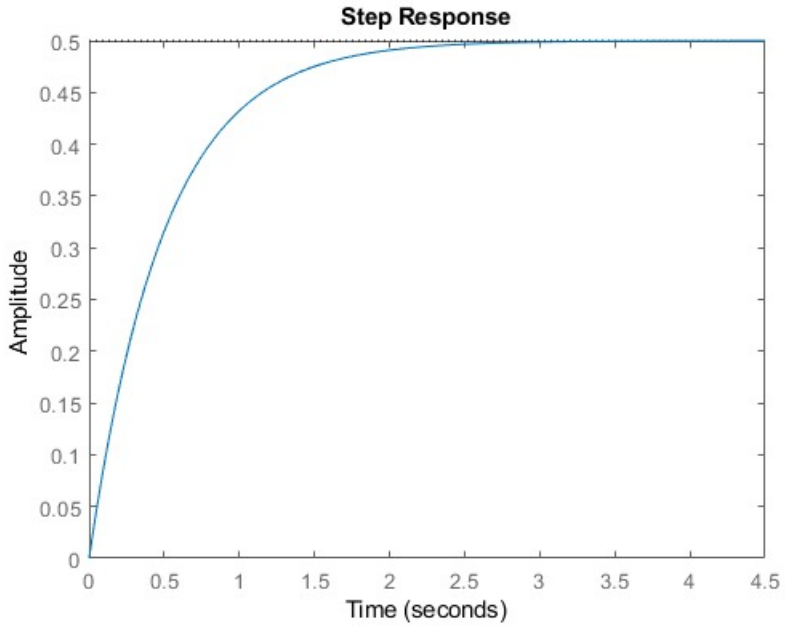
Para fins de validação do primeiro exemplo, foi realizado uma nova comparação. Os sistemas envolvidos foram os seguintes:

- Processo: $\frac{1}{s+2}$
- Controlador: $\frac{1}{s^2+2s+3}$

Resposta ao Degrau - Processo

Os gráficos gerados nas duas plataformas são mostrados nas Figuras 18 e 19.

Figura 18 – Resposta ao Degrau do Processo em malha aberta gerado pelo Matlab.



Fonte: autoria própria, 2021

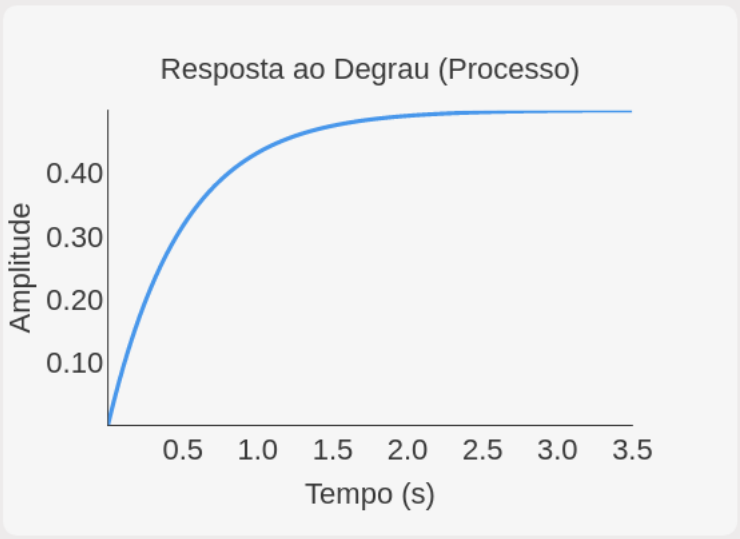
Resposta ao Degrau - Sistema em Malha Fechada

A resposta ao degrau do sistema em malha fechada é mostrada nas Figuras 20 e 21.

Diagramas de Bode

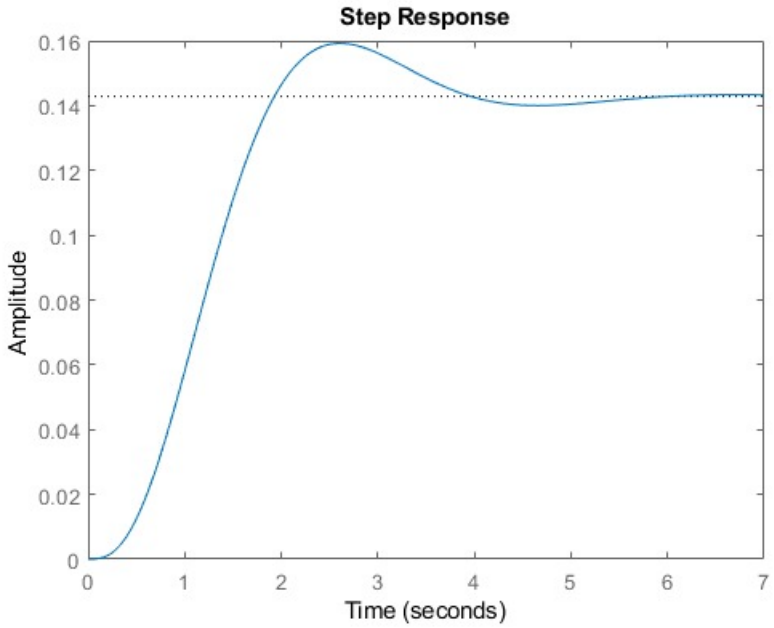
Nas Figuras 22 e 23 são representados os Diagramas de Bode obtidos pelo Matlab e pelo *Project Achilles* respectivamente.

Figura 19 – Resposta ao Degrau do Processo em malha aberta gerado pelo *Project Achilles*.



Fonte: autoria própria, 2021

Figura 20 – Resposta ao Degrau do sistema em malha fechada gerado pelo Matlab.

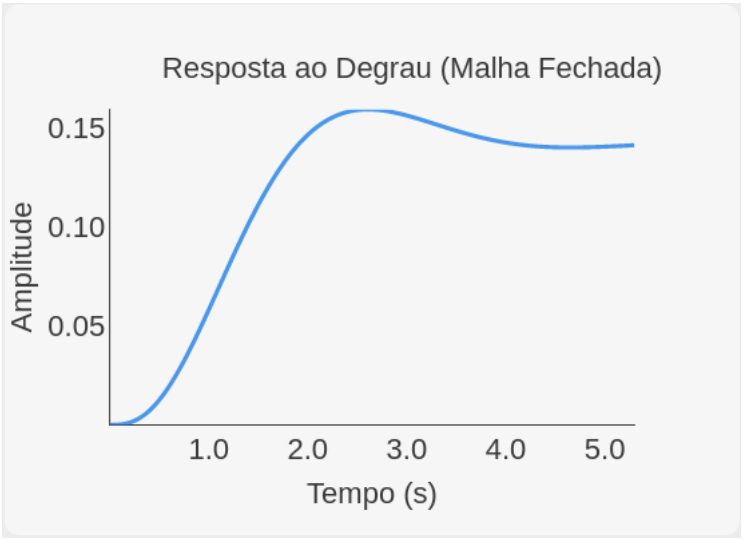


Fonte: autoria própria, 2021

Lugar das Raízes

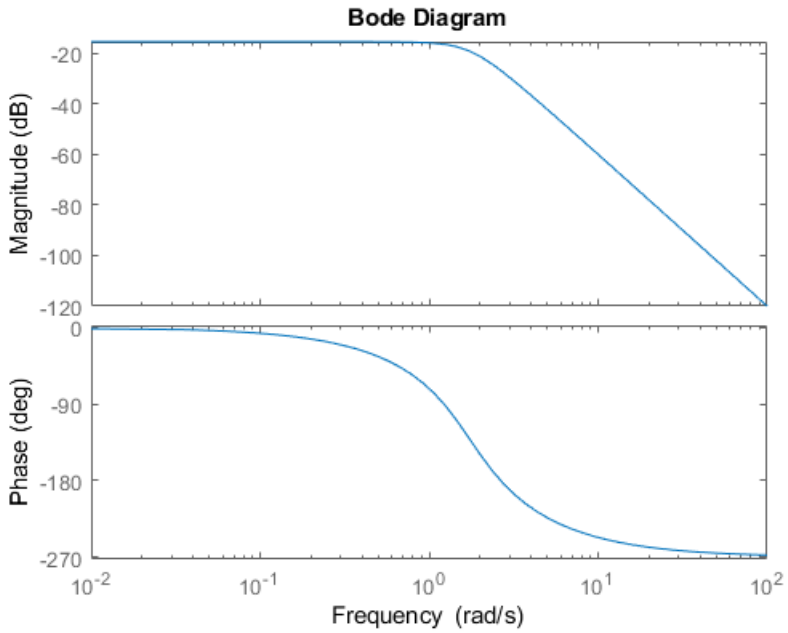
O lugar das raízes, mostrado nas Figuras 24 e 25, é o gráfico onde a diferença entre os dois é perceptível. Notavelmente, o gráfico gerado pelo *Project Achilles* apresenta

Figura 21 – Resposta ao Degrau do sistema em malha fechada gerado pelo Project Achilles.



Fonte: autoria própria, 2021

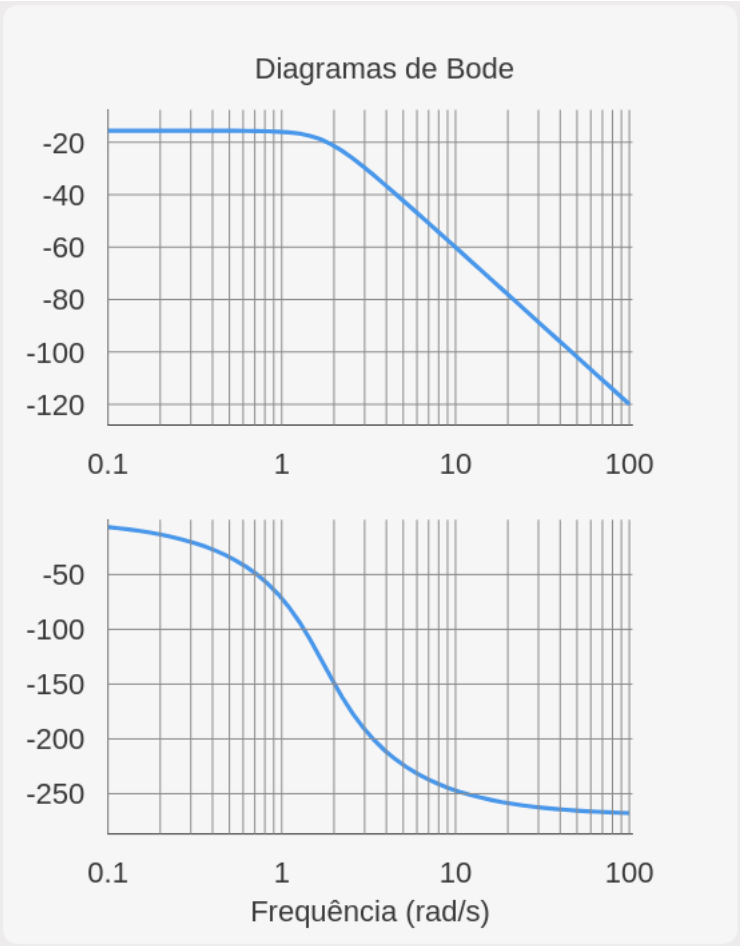
Figura 22 – Diagrama de Bode gerado pelo Matlab.



Fonte: autoria própria, 2021

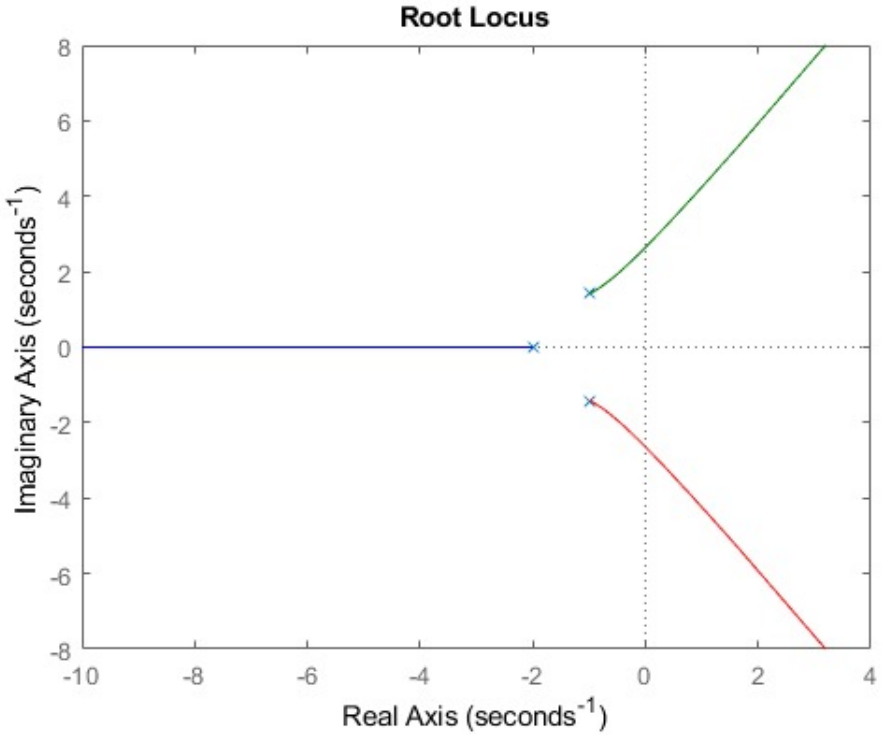
falhas na construção do gráfico ao não gerar um círculo perfeito, dispondo de vários pontos de descontinuidade. Além disso, não possui indicação dos zeros. A indicação dos polos é marcada com um sinal de "+"ao invés de um "x"e o eixo imaginário não apresenta rótulo.

Figura 23 – Diagrama de Bode gerado pelo *Project Achilles*.



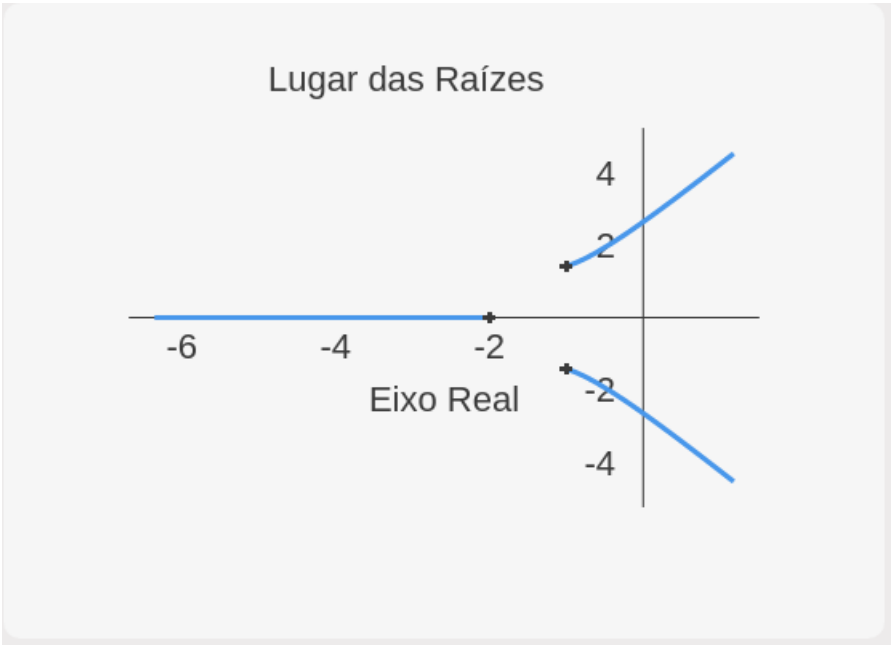
Fonte: autoria própria, 2021

Figura 24 – Lugar das Raízes gerado pelo Matlab.



Fonte: autoria própria, 2021

Figura 25 – Lugar das Raízes gerado pelo *Project Achilles*.



Fonte: autoria própria, 2021

4.3 Resultados e Melhorias Futuras

É possível afirmar, com base nas comparações, que a plataforma alcançou seu objetivo inicial. Pode-se perceber que os dados mostrados estão condizentes com os da plataforma *Matlab*, um *software* bastante confiável, o que confirma a validação gráfica da ferramenta, faltando apenas alguns ajustes. Fica claro, contudo, a necessidade de algumas melhorias diretas para uma melhor experiência do usuário com a utilização do *software* e, por isso, são listadas a seguir algumas dessas possíveis melhorias:

- Correção do gráfico do Lugar das Raízes:
  - Incluir apresentação dos zeros.
  - Corrigir marcação dos polos (de "+"para "x").
  - Incluir melhor identificação nos eixos.
  - Capturar mais informações nos gráficos (caso não seja possível com o pacote *control* do Python, utilizar um método numérico.
- Melhorar tratamento de erros:
  - Tratar erros de comunicação com o servidor e outros tipos de erro que não são tratados atualmente.
- Adição de testes unitários.
- Inclusão de outras informações como o Diagrama de Nyquist.
- Inclusão de outras páginas como "Sobre"e "Ajuda".
- Inclusão de análise para sistemas discretos (domínio z).
- Inclusão de análise por variáveis de estados.
- Adição da possibilidade de incluir atraso.
- Adição de mais informações nos gráficos como indicação gráfica do *sobre-sinal* e frequência de corte.
- Verificar possibilidade de cálculo em tempo real, permitindo alterações mais rápidas nos gráficos.

5 Considerações Finais

Este trabalho teve como objetivo detalhar a construção e desenvolvimento do projeto de conclusão de curso intitulado *Project Achilles*, que se trata de uma plataforma *web* gratuita e de fácil utilização para projetos de sistemas de controle lineares. Primeiramente foi apresentada uma introdução teórica sobre as principais tecnologias utilizadas no desenvolvimento e uma visão geral sobre as representações de sistemas de controle. Todo o desenvolvimento, que se deu em três etapas, foi descrito, contando com inclusões de códigos, esquemas e pseudo-códigos para explicar o funcionamento dos componentes.

Foram gerados gráficos de resposta ao degrau, diagramas de Bode e lugar das raízes por meio do *Matlab* e do *Project Achilles* para fins de comparação e validação. Verificando que o *Project Achilles* de fato apresenta resultados satisfatórios. Contudo, ainda existe muito a ser construído para que a plataforma alcance um nível de excelência. É perceptível que existem alguns pontos falhos, principalmente no gráfico gerado do lugar das raízes. Porém, é importante frisar que esse foi apenas o início do trabalho. Plataformas como essa são normalmente construídas por equipes de várias pessoas, com tempo e tarefas dedicadas ao desenvolvimento. Pela própria natureza do trabalho em questão, não é possível dedicar os recursos necessários que outras plataformas possuíram para seu completo desenvolvimento.

Várias dificuldades foram encontradas durante a execução do trabalho, principalmente na construção da aplicação cliente, ou *front end*, já que não existia qualquer experiência com o *React* e pouca experiência com *JavaScript* de um modo geral, além de ser necessário desenvolver uma visão do sistema como um todo, integrando as três etapas de desenvolvimento (*back end*, *front end* e infraestrutura). Porém essas dificuldades se mostraram bastante recompensadoras justamente pela oportunidade de trabalhar com essas tecnologias com as quais não existia um conhecimento profundo. Foi possível aprimorar o conhecimento em todas essas ferramentas, mostrando que o desenvolvimento do projeto também forneceu um grande crescimento pessoal.

Por fim, espera-se que o desenvolvimento do trabalho possa ser continuado no futuro, trazendo as melhorias propostas para que a plataforma possa ser uma ferramenta cada vez mais útil aos interessados.

Referências

BANKS, A., PORCELLO, E. (2020). Learning React: Modern Patterns for Developing React Apps. United States: O'Reilly Media, p. 71.

OGATA, K. (2011). Engenharia de controle moderno. Brasil: PRENTICE HALL BRASIL.

BISHOP, R. H., DORF, R. C. (2009). Sistemas de controle modernos. Brasil: LTC.

GRINBERG, M. (2018). Flask Web Development: Developing Web Applications with Python. United States: O'Reilly Media.

REIS, D. C. S (2017). Execução e Gestão de Aplicações Conteinerizadas. 169 f. Dissertação (Mestrado) - Curso de Engenharia de Redes e Sistemas Informáticos, Departamento de Ciências dos Computadores, Faculdade de Ciências da Universidade do Porto, Porto, 2017. Disponível em: https://repositorio-aberto.up.pt/bitstream/10216/109650/2/236222.pdf. Acesso em: 18 abr. 2021.

FRANKLIN, G. F., et al. (2013). Sistemas de Controle para Engenharia - 6ed. Brasil: Bookman Editora. p. 189-191.

Apêndices

APÊNDICE A – App.js

```
1  import React, { useState, useEffect } from 'react';
2  import {BrowserRouter as Router, Route, Switch} from 'react-router-dom'
3
4  import BodeDiagram from './components/BodeDiagram'
5  import RootLocus from './components/RootLocus'
6  import StepResponse from './components/StepResponse'
7  import Sidebar from './components/Sidebar'
8
9  import './App.css';
10 import Loading from './components/Loading';
11
12 function App() {
13   const [systemAnalysis, setSystemAnalysis] = useState([]);
14   const [showChartAnalysis, setShowChartAnalysis] = useState(false);
15   const [showLoading, setShowLoading] = useState(false);
16
17   useEffect(() => {
18     fetch('http://35.231.19.210:5000/home').then(res => res.json()).then
         (data => {});
19   }, []);
20
21   const sendInfoOL = async (system) => {
22       console.log(system)
23       if(system.load === true){
24         setShowChartAnalysis(false)
25         setShowLoading(true)
26       }
27
28       const res = await fetch('http://35.231.19.210:5000/analysis',{
29         method: 'POST',
30         headers: {
31           'Content-type': 'application/json'
32         },
33         body: JSON.stringify(system)
34     })
35
36     const data = await res.json()
37     console.log(data)
38     setShowLoading(false)
39
40     var input_data = []
41     data.x_axis_ol.forEach((element, index) => {
42       input_data[index] = {"x": element, "y": data.y_axis_ol[index]}
```

```
43      });
44
45      setSystemAnalysis(data)
46      setShowChartAnalysis(true)
47    }
48
49    return (
50       <Router >
51       <div className="page">
52         <Switch>
53            <Route path='/' exact render={(props) => (
54              <>
55              <Sidebar onSend={sendInfoOL}/>
56              { showLoading === true && <Loading />}
57              { showChartAnalysis === true && <StepResponse input_data={
                    systemAnalysis} className="test"/>}
58              { showChartAnalysis === true && <BodeDiagram input_data={
                    systemAnalysis} className="test"/>}
59              { showChartAnalysis === true && <RootLocus input_data={
                    systemAnalysis} className="test"/>}
60              </>
61            )}/>
62            <Route path='/help' exact render={(props) => (
63              <>
64                <p>Pagina indisponivel no momento</p>
65              </>
66            )}/>
67            <Route path='/about' exact render={(props) => (
68              <>
69                <p>Pagina indisponivel no momento</p>
70              </>
71            )}/>
72         </Switch>
73         </div>
74     </Router>
75
76    );
77  }
78
79  export default App;
```

APÊNDICE B – Sidebar.js

```
import { Link } from 'react-router-dom'
import React, {useState} from 'react'
import {NavbarData} from './NavbarData'
import '../index.css'
import {IconContext} from 'react-icons'

export const Sidebar = ({ onSend }) => {
    const [hnum, setHnum] = useState('')
    const [hden, setHden] = useState('')
    const [gnum, setGnum] = useState('')
    const [gden, setGden] = useState('')
    const [load, setLoad] = useState(false)

    const onSubmit = (e) => {
        e.preventDefault()

        if(!hnum) {
            alert('Por favor, digite um numerador para o processo')
            return
        }
        if(!hden) {
            alert('Por favor, digite um denominador para o processo')
            return
        }
        if(!gnum) {
            alert('Por favor, digite um numerador para o controlador')
            return
        }
        if(!gden) {
            alert('Por favor, digite um denominador para o controlador')
            return
        }

        if (!hnum.match(/[0-9]+$/)){
            alert('Por favor, digite apenas numeros.')
            return
        }
        if (!hden.match(/[0-9]+$/)){
            alert('Por favor, digite apenas numeros.')
            return
        }
        if (!gnum.match(/[0-9]+$/)){
            alert('Por favor, digite apenas numeros.')
```

```
44              return
45          }
46          if (!gden.match(/[0-9]+$/)){
47              alert('Por favor, digite apenas numeros.')
48              return
49          }
50
51          onSend({ hnum, hden, gnum, gden, load })
52
53          setHnum('')
54          setHden('')
55          setGnum('')
56          setGden('')
57      }
58      return (
59          <>
60          <IconContext.Provider value={{color: '#fff'}}>
61                  <div className="navbar">
62                          {NavbarData.map((item, index) => {
63                              return (
64                                  <li key={index} className={item.
                                        className}>
65                                      <Link to={item.path}>
66                                          {item.icon}
67                              <span>{item.title}</span>
68                                      </Link>
69                                  </li>
70                              )
71                          })}
72                  </div>
73                  <nav className='nav-menu-active'>
74                      <h1 classNAme="title-text">Project Achilles</h1>
75                          <ul className='nav-menu-items'>
76
77                          <form className="form" onSubmit={onSubmit}>
78                              <h2 className='form-text2'>Processo</h2>
79                              <div>
80                                  <li className='form-text'>Numerador</li>
81                                  <input className='input-bar' type="text"
                                        placeholder="Exemplo: 1,2" value={
                                        hnum} onChange={(e) => setHnum(e.
                                        target.value)}/>
82                              </div>
83
84
85                              <div>
86                                  <li className='form-text'>Denominador</
```

```
                                    li>
87                                  <input className='input-bar' type="text"
                                        placeholder="Exemplo: 1,2,3" value={
                                    hden} onChange={(e) => setHden(e.
                                    target.value)}/>
88                              </div>
89
90                              <div className="line"/>
91                              <h2 className='form-text2'>Controlador</h2>
92                              <div>
93                                  <li className='form-text'>Numerador</li>
94                                  <input className='input-bar' type="text"
                                        value={gnum} placeholder="Exemplo:
                                    1,2" onChange={(e) => setGnum(e.
                                    target.value)}/>
95                              </div>
96
97
98                              <div>
99                                  <li className='form-text'>Denominador</
                                    li>
100                                 <input className='input-bar' type="text"
                                        value={gden} placeholder="Exemplo:
                                    1,2,3" onChange={(e) => setGden(e.
                                    target.value)}/>
101                             </div>
102                             <input type='submit' value='Enviar'
                                    className='btn' onClick={(e) => setLoad(
                                    true)}/>
103                         </form>
104
105                     </ul>
106                 </nav>
107             </IconContext.Provider>
108         </>
109
110     )
111 }
112
113 export default Sidebar
```

APÊNDICE C – StepResponse.js

```
1  import React from 'react'
2  import { VictoryLine, VictoryChart, VictoryAxis, VictoryLabel,
       VictoryVoronoiContainer } from 'victory'
3
4  const StepResponse = ({ input_data }) => {
5
6      let data = [];
7      input_data.x_axis_ol.forEach((element, index) => {
8      data[index] = {"x": element, "y": input_data.y_axis_ol[index]};
9      });
10
11     let data_cl = [];
12     input_data.x_axis_cl.forEach((element, index) => {
13     data_cl[index] = {"x": element, "y": input_data.y_axis_cl[index]};
14     });
15
16     return (
17         <div>
18             <div className="step-response">
19                 <VictoryChart height={250} width={350}
20                 containerComponent={
21                 <VictoryVoronoiContainer
22                     labels={({ datum }) => `Amplitude: ${datum.y.
                           toPrecision(2)}\n Tempo: ${datum.x.toPrecision(2)
                           }s`  }/>}
23                 >
24                     <VictoryLabel x={75} y={30} text="Resposta ao Degrau
                           (Processo)" />
25                     <VictoryAxis label="Tempo (s)" tickLabelComponent={<
                           VictoryLabel dy={-7}/>}/>
26                     <VictoryAxis dependentAxis label="Amplitude"
27                         axisLabelComponent={<VictoryLabel dy={-10}/>}
28                         tickLabelComponent={<VictoryLabel dx={8.8}/>}
29                     />
30                     <VictoryLine
31                         data={data} style={{ data:{ stroke: "#378bec"}}}
32                     />
33                 </VictoryChart>
34             </div>
35
36             <div className="step-response2">
37                 <VictoryChart height={250} width={350}
38                 containerComponent={
```

```
39                      <VictoryVoronoiContainer
40                          labels={({ datum }) => `Amplitude: ${datum.y.
                               toPrecision(2)}\n Tempo: ${datum.x.
                               toPrecision(2)}s`  }/>}
41                      >
42                      <VictoryLabel x={75} y={30} text="Resposta ao Degrau
                            (Malha Fechada)" />
43                      <VictoryAxis label="Tempo (s)" tickLabelComponent={<
                           VictoryLabel dy={-7}/>} />
44                      <VictoryAxis dependentAxis label="Amplitude "
45                          axisLabelComponent={<VictoryLabel dy={-10}/>}
46                          tickLabelComponent={<VictoryLabel dx={8.8}/>}
47                          // tickFormat={(t) => `\${t.toPrecision(2)}`}
48                      />
49                      <VictoryLine
50                      data={data_cl} style={{ data:{ stroke: "#378bec"
                           }}}/>
51                  </VictoryChart>
52              </div>
53          </div>
54      )
55  }
56
57  export default StepResponse
```

APÊNDICE D – BodeDiagram.js

```
import React from 'react'
import { VictoryLine, VictoryChart, VictoryAxis, VictoryLabel,
    VictoryVoronoiContainer } from 'victory'

const BodeDiagram = ({input_data}) => {

    let data = [];
    let phase = [];
    input_data.omega.forEach((element, index) => {

    data[index] = {"x": element, "y": input_data.magnitude[index]};
    phase[index] = {"x": element, "y": input_data.phase[index]};
    });

    return (
        <div>
            <div className="bode-mag">
                <VictoryChart  domainPadding={10} scale={{x: "log", y:"
                    linear"}}
                    minDomain={{ x: input_data.omega[0] }} height={250}
                        width={350}
                    containerComponent={
                        <VictoryVoronoiContainer
                            labels={({ datum }) => `Amplitude: ${datum.y
                                .toPrecision(2)} dB \n Frequ ncia: ${
                                datum.x.toPrecision(2)} rad/s `  }/>}
                >

                    <VictoryLabel x={120} y={30} text="Diagramas de Bode
                        " />

                    <VictoryAxis dependentAxis label="Magnitude (dB)"
                        axisLabelComponent={<VictoryLabel dy={-30}/>}
                        style={{grid:{stroke: ({ tick }) => "grey"}}}
                    />

                    <VictoryAxis axisLabelComponent={<VictoryLabel dy
                        ={25}/>}
                        orientation="bottom"
                        style={{grid:{stroke: ({ tick }) => "grey"}}}
                        tickFormat={(t) => Number.isInteger(Math.log10(t
                            )) ? t: ''}
                     />
```

```
36
37                      <VictoryLine
38                      data={data} style={{ data:{ stroke: "#378bec"}}} />
39                  </VictoryChart>
40              </div>
41
42              <div className="bode-phase">
43                  <VictoryChart domainPadding={10} scale={{x: "log", y:"
                        linear"}}
44                      minDomain={{ x: input_data.omega[0] }} height={250}
                            width={350}
45                      containerComponent={
46                          <VictoryVoronoiContainer
47                              labels={({ datum }) => `Magnitude: ${datum.y
                                    .toPrecision(2)} dB \n Frequ ncia: ${
                                    datum.x.toPrecision(2)} rad/s `  }/>}
48                  >
49
50                      <VictoryLine
51                      data={phase} style={{ data:{ stroke: "#378bec"}}}
52                      />
53
54                      <VictoryAxis dependentAxis label="Phase (  )"
55                          axisLabelComponent={<VictoryLabel dy={-30}/>}
56                          style={{grid:{stroke: ({ tick }) => "grey"}}}
57                      />
58
59                      <VictoryAxis  label="Frequ ncia (rad/s)"
60                          axisLabelComponent={<VictoryLabel dy={5}/>}
                                orientation="bottom"
61                          style={{grid:{stroke: ({ tick }) => "grey"}}}
62                          tickFormat={(t) => Number.isInteger(Math.log10(t
                                )) ? t: ''}
63                      />
64                  </VictoryChart>
65
66              </div>
67          </div>
68
69
70      )
71
72  }
73  export default BodeDiagram
```

APÊNDICE E – RootLocus.js

```
import React from 'react'
import { VictoryLine, VictoryChart, VictoryScatter,
    VictoryVoronoiContainer, VictoryAxis, VictoryLabel, createContainer}
    from 'victory'

const RootLocus = ({input_data}) => {
    var data = [];
    const items = [];
    const VictoryZoomVoronoiContainer = createContainer("zoom", "voronoi
        ");

    for (var i=0; i < input_data.num_poles; i++){
        data[i] = [];
        items.push(<VictoryLine key={i} data={data[i]} eventKey="" style
            ={{ data:{ stroke: "#378bec"}}} />)
    }
    let poles = [];

    input_data.root_imag.forEach((element, index) => {
    for (var i=0; i < input_data.num_poles; i++){
        data[i][index] = {"x": input_data.root_real[index][i], "y":
            element[i]};
        poles.push({x:input_data.root_real[0][i], y:input_data.root_imag
            [0][i]})
    }
    });

    return (
        <div className="root-locus">
            <VictoryChart domainPadding={10} height={250} width={350}
                containerComponent={
                    <VictoryZoomVoronoiContainer labels={({ datum, data=
                        input_data }) => `Real: ${datum.x.toPrecision(2)
                        }\n Imag: ${datum.y.toPrecision(2)} \n Ganho: ${
                        data.root_gain[datum.eventKey]}\n` }/>
                }>
                <VictoryLabel x={105} y={30} text="Lugar das Ra zes" />
                {items}
                <VictoryScatter data={poles} symbol="plus" size={1.75}/>
                <VictoryAxis fixLabelOverlap={true} />
            </VictoryChart>
        </div>
    )
```

```
35
36 }
37 export default RootLocus
```

APÊNDICE F – NavbarData.js

```
1  import React from 'react'
2  import { AiFillHome, AiOutlineQuestionCircle, AiOutlineInfoCircle } from
       'react-icons/ai'
3
4  export const NavbarData = [
5      {
6          title: 'In cio',
7          path: '/',
8          icon: <AiFillHome className="nav-icon"/>,
9          className: 'nav-text'
10     },
11     {
12         title: 'Ajuda',
13         path: '/help',
14         icon: <AiOutlineQuestionCircle className="nav-icon"/>,
15         className: 'nav-text'
16     },
17     {
18         title: 'Sobre',
19         path: '/about',
20         icon: <AiOutlineInfoCircle className="nav-icon"/>,
21         className: 'nav-text'
22     },
23 ]
```

APÊNDICE G – Dockerfile - Front

```
FROM node:13.12.0-alpine as build

WORKDIR /front
ENV PATH /front/node_modules/.bin:$PATH
COPY package.json ./
COPY package-lock.json ./
RUN npm ci --silent
RUN npm install react-scripts@3.4.1 -g --silent
COPY . ./
RUN npm run build

FROM nginx:stable-alpine
COPY --from=build /front/build /usr/share/nginx/html
RUN rm /etc/nginx/conf.d/default.conf
COPY nginx/nginx.conf /etc/nginx/conf.d
EXPOSE 3000
CMD ["nginx", "-g", "daemon off;"]
```