UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ALESSANDRO CAPPELLINI PORTUGAL

EMPREGO DA BIBLIOTECA DRWATSON.JL EM SIMULAÇÕES DE CIÊNCIAS TÉRMICAS

GUARAPUAVA

2023

ALESSANDRO CAPPELLINI PORTUGAL

EMPREGO DA BIBLIOTECA DRWATSON.JL EM SIMULAÇÕES DE CIÊNCIAS TÉRMICAS

USE OF THE DRWATSON.JL PACKAGE IN THERMAL SCIENCE SIMULATIONS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia Mecânica do Curso de Bacharelado em Engenharia Mecânica da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Christian Naaktgeboren

GUARAPUAVA

2023

ALESSANDRO CAPPELLINI PORTUGAL

EMPREGO DA BIBLIOTECA DRWATSON.JL EM SIMULAÇÕES DE CIÊNCIAS TÉRMICAS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia Mecânica do Curso de Bacharelado em Engenharia Mecânica da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 07/julho/2023

Prof. Christian Naaktgeboren
Doutorado
Universidade Tecnológica Federal do Paraná

Prof. Sérgio Dalmas
Doutorado
Universidade Tecnológica Federal do Paraná

Prof. Sediane Carmem Lunardi Hernandes
Doutorado
Universidade Tecnológica Federal do Paraná

Prof. Aldo Przybysz
Doutorado
Universidade Tecnológica Federal do Paraná

GUARAPUAVA

2023

Dedico este trabalho aos meus pais que por toda a vida me incentivaram e apoiaram em meus estudos, além de me proporcionar melhor a educação que estava ao nosso alcance.

AGRADECIMENTOS

Gostaria de agradece a priori a Deus pelo milagre da vida e pelos caminhos que tem me conduzido. A posteriori, gostaria de agradecer aos meus pais pelo apoio em mais essa jornada, minha namorada Vitória pela paciência em minhas ausências, meu amigo Luiz Eduardo Caldas Kramer pelo suporte na área tecnológica, aos meus professores, em especial ao meu orientador, por apontar o caminho a ser trilhado e à UTFPR como um todo por toda a estrutura e conhecimentos oferecidos.

RESUMO

Este trabalho, classificado como qualitativo, se propôs a explorar e estudar o potencial das funcionalidades disponíveis na biblioteca `DrWatson.jl` que proporcionem a engenheiros e pesquisadores pouca preocupação em relação ao gerenciamento de suas simulações com variação de parâmetros de entrada, permitindo-lhes focar na ciência de fato. As funcionalidades do pacote são testadas em uma simulação simplificada do Ciclo Otto que se aproxime suficientemente de um ciclo real, de forma a extrair o máximo possível do que o pacote tem a oferecer, tal como a inicialização de projetos organizados, criação e ativação de um ambiente virtual, comandos que facilitem o armazenamento dos dados do projeto nos diretórios locais e seu posterior acesso, além de fornecer auxílio a outros que queiram fazer verificações independentes dos resultados das simulações.

Palavras-chave: julia lang; drwatson; ciência de dados; eficiência.

ABSTRACT

This work, classified as qualitative, proposes to explore and study the potential of the functionalities available in the `DrWatson.jl` library that provide engineers and researchers a little concern in relation to the management of their simulations with variation of input parameters, allowing them to focus in the science. The package's functionality is tested in a simplified Otto Cycle simulation that is close enough to a real cycle, to extract as much as possible what the package has to offer. Such as starting organized projects, creating and activating a virtual environment, commands that facilitate the storage of project data in local directories and their subsequent access, beside to providing assistance to others who want to make independent verifications of simulation results.

Keywords: julia lang; drwatson; data science; efficiency.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Abreviaturas

MB	Megabyte

Siglas

2T	Dois Tempos
4T	Quatro Tempos
CWI	Centrum Wiskunde & Informatica
FTHA	Finite-Time Heat Addition
MEF	Método dos Elementos Finitos
MIT	Massachusetts Institute of Technology
PMI	Ponto Morto Inferior
PMS	Ponto Morto Superior

LISTA DE SÍMBOLOS

LETRAS LATINAS

$\bar{f}$	Fração residual de gases	
c_p	Calor específico à pressão constante	$[\text{kJ/kg·K}]$
c_v	Calor específico à volume constante	$[\text{kJ/kg·K}]$
k	Razão dos calores específicos	
m	Massa	$[\text{kg}]$
N	Rotação do motor	$[\text{RPM}]$
n	Coeficiente politrópico	
Q	Calor	$[\text{kJ}]$
q	Calor específico	$[\text{kJ/kg}]$
r	Razão de compressão	
S	Curso do pistão	$[\text{m}]$
V	Volume	$[\text{m}^3]$
V_d	Volume deslocado ou cilindrada do motor	$[\text{m}^3]$
V_{du}	Volume deslocado unitário ou cilindrada unitária	$[\text{m}^3]$
W	Trabalho	$[\text{kJ}]$
w	Trabalho específico	$[\text{kJ/kg}]$

LETRAS GREGAS

η_t	Eficiência térmica

SUBSCRITOS

$_{ar}$	Propriedade relacionada ao ar
$_{comb}$	Propriedade relacionada ao combustível
$_{ent}$	Propriedade relacionada a um valor que está entrando no sistema
$_{liq}$	Propriedade relacionada a um valor líquido
$_{max}$	Propriedade relacionada a um valor máximo
$_{min}$	Propriedade relacionada a um valor mínimo
$_{PMI}$	Propriedade relacionada ao ponto morto inferior
$_{PMS}$	Propriedade relacionada ao ponto morto superior
$_{res}$	Propriedade relacionada a um valor residual de queima
$_{sai}$	Propriedade relacionada a um valor que está saindo no sistema
$_{tot}$	Propriedade relacionada a um valor total

SUMÁRIO

1 INTRODUÇÃO

Um problema recorrente dentro do mundo da ciência de dados e da computação numérica, embora omitido em muitas publicações científicas, é o gerenciamento de projetos e sua compatibilidade entre diferentes máquinas (Driven Data, 2018). Certamente, muitos dos que estão inseridos nesse meio já tiveram problemas voltados aos dados de simulações que se sobrescreveram por terem o mesmo nome ou que não foram salvos de maneira organizada e se perderam, projetos que não rodaram em outras máquinas por não terem os mesmos pacotes instalados, entre muitos outros problemas comuns nesse ambiente (DATSERIS *et al.*, 2020b).

Desta forma, com o passar dos anos e com a evolução tanto das máquinas quanto das linguagens de programação, houve o surgimento de soluções criadas pela comunidade e disponibilizadas na *web* para que todos pudessem usufruir e contribuir para melhorar suas funcionalidades. Deste modo, as soluções criadas para linguagens de programação já existentes, usadas no nicho de análise de dados e simulações, deram ao usuário a possibilidade de interagir com a máquina de maneira mais ágil e menos "burocrática", facilitando essa comunicação (PERKEL, 2015) como é o caso de `DrWatson.jl`.

Dentro deste campo, a proposta do pacote é eliminar preocupações e prejuízos que possam ocorrer devido a inconsistências e que causem impactos negativos no trabalho. O objetivo de `DrWatson.jl` é acelerar o *workflow* do usuário e garantir robustez ao projeto. Com ênfase no método científico, a biblioteca se propõe a criar um projeto estruturado e disponibilizar comandos que facilitem a rastreabilidade e execução das simulações, evitando prejuízos que levam ao retrabalho e, consequentemente, à perda de tempo. Seu lema é permitir que o cientista se concentre na ciência e deixe de lado problemas computacionais, aumentando assim sua produtividade (DATSERIS *et al.*, 2020a).

Sendo assim, este trabalho tem o objetivo de realizar um estudo de caso de caráter investigativo com relação à biblioteca `DrWatson.jl`, em que suas funcionalidades serão postas à prova para entender quais dos seus recursos disponíveis realmente podem auxiliar na criação, desenvolvimento, gerenciamento e compartilhamento de projetos científicos de maneira robusta e segura.

2 OBJETIVOS

No capítulo de objetivos, o objetivo geral assume um papel fundamental de definir o escopo da investigação. Ele estabelece uma visão abrangente do projeto como um todo. Já os tópicos dos objetivos específicos são responsáveis por abordar, de forma explícita, cada assunto estudado pelo trabalho e assegurar uma compreensão completa do projeto.

2.1 Objetivos Gerais

O presente trabalho pretende explorar ao máximo, de forma qualitativa e com caráter investigativo, as possibilidades inclusas no pacote `DrWatson.jl` e suas aplicações em estudos e simulações de ciências térmicas a fim de descobrir como a extensão é capaz de auxiliar no gerenciamento e automatização de simulações e obtenção de resultados.

2.2 Objetivos Específicos

1. Revisão da documentação do pacote `DrWatson.jl`
2. Identificar simulações para testes de funcionalidades utilizando a linguagem `Julia`.
3. Conduzir estudos de caso em cima do pacote aplicando-o às simulações do Ciclo Otto FTHA.
4. Documentação dos estudos.
5. Conclusão e apresentação.

3 JUSTIFICATIVA

Dentro da área de simulação computacional, existe um grande ponto a ser levantado e discutido com relação à criação de novos projetos, salvamento de bancos de dados referente a simulações repetitivas com diferentes parâmetros de entrada e o seu compartilhamento entre diferentes ambientes computacionais/pesquisadores e/ou adaptações de estudo de repetibilidade independentes e/ou aplicação do método científico. Esses fatores geram erros e falta de consistência nos estudos, o que acaba por desviar a atenção do pesquisador do trabalho para a ferramenta de trabalho.

Tendo isso em vista, o projeto possui relevância científica por levantar um estudo de caso a respeito de uma ferramenta que pode contribuir para o auxílio e gerenciamento de simulações tanto da indústria quanto acadêmicas, além de discutir um fato pouco abordado que são as dificuldades em gerar reprodutibilidade nas simulações. Por meio da biblioteca `DrWatson.jl`, pode-se gerenciar diretórios de maneira robusta sem que se percam referências de pastas, executar e salvar simulações distintas sem que acabem sendo sobrescritas ou perdidas, permitir a integração com nuvens e ferramentas de versionamento – como `Git` e `GitHub`, por exemplo – além de tornar o projeto compatível com qualquer máquina que tenha `Julia` instalada por meio do comando de ativação rápida de projetos.

Nesse contexto, o trabalho irá explorar as funcionalidades de `DrWatson.jl` e aplicá-las às simulações computacionais de ciências térmicas, objetivando encontrar formas para resolver essas inconsistências de projeto causadas pela falta de uma ferramenta que assista o pesquisador em tarefas organizacionais de maneira simples.

4 REFERENCIAL TEÓRICO

O ecossistema computacional é composto por uma vasta gama de linguagens de programação, *frameworks*, bibliotecas e ferramentas que fornecem suporte e funcionalidades para desenvolvedores em suas tarefas diárias. Nesse cenário, as linguagens `Python` e `Julia` destacam-se como duas das mais importantes no nicho de ciência de dados.

4.1 Python

A história da linguagem `Python` remonta o final da década de 80 e início dos anos 90, em que existia uma grande necessidade por parte de cientistas no ambiente computacional, em relação à velocidade com que os códigos eram escritos, pois as ferramentas mais populares na época como a linguagem `C`, por exemplo, embora tivessem ótima performance de processamento, envolviam grande esforço de escrita do código por parte do programador, que deveria seguir rígidas regras de estrutura e sintaxe ao construir seu projeto (BOUDREAU, 2019).

Dessa forma, muitos esforços foram realizados no desenvolvimento de linguagens de programação que diminuíssem os esforços de escrita de código, como a linguagem Python. Guido van Rossum, matemático e programador holandês do Centrum Wiskunde & Informatica (CWI), no intuito de unir as comodidades da linguagem `ABC`, outro projeto no qual trabalhara, com a interface do sistema operacional `Amoeba`, criou a linguagem `Python` (VENNERS, 2003).

Lançada inicialmente em 1991, Python rapidamente ganhou popularidade devido à sua sintaxe simples e legibilidade, tornando-se uma linguagem de programação amplamente utilizada em uma variedade de domínios. Sua filosofia de design enfatiza a clareza, o pragmatismo e a facilidade de uso, tornando-a uma opção atraente tanto para iniciantes quanto para programadores experientes. Além disso, a abordagem de Van Rossum de "baterias incluídas", levou ao desenvolvimento de uma ampla biblioteca padrão, fornecendo aos usuários uma gama diversificada de ferramentas e módulos para facilitar o desenvolvimento de aplicativos.

`Python` é uma linguagem de alto nível, de semântica dinâmica, interpretada, modular, multiplataforma e orientada a objetos (ROSSUM; DRAKE, 1995) que foi estruturada com base nos já construídos alicerces da linguagem `ABC` e importou funcionalidades de linguagens como `C`, `Perl`, `Haskel`, `Icon` e `Modula-3`, inclusive com grande parte de seu código fonte escrito na própria linguagem `C` (SOUZA, 2021).

Apesar de ser uma linguagem interpretada, ao invés de compilada, e com um custo computacional maior que a linguagem `C`, `Python` acaba sendo menos eficiente na execução de programas, porém seu dinamismo e facilidade de escrita acabam por compensar esse tempo. Além de uma ótima ferramenta para a ciência de dados, a linguagem `Python` evoluiu para ser uma das linguagens mais usadas dos dias atuais na maioria das áreas da computação,

seja para *scripting* e automação, desenvolvimento web, enquadramento de testes, *big data*, computação gráfica, inteligência artificial, entre outras (ROVEDA, 2020).

Em seus mais de 30 anos, desde o lançamento, `Python` só tem crescido e ganhado cada vez mais aceitação perante a comunidade. E é nesse contexto que em 2009, uma equipe de 4 integrantes do Massachusetts Institute of Technology (MIT), se inserem. Dispostos a criar uma linguagem que, assim como o `Python`, seja de alto nível e de semântica dinâmica para a computação numérica em razão de melhorar funcionalidades de outras plataformas como `Octave`, `MATLAB®`, `R`, `SciPy` e `SciLab` e de performance tão boa quanto `C` e `Fortran`, que em 2012 `Julia` é lançada (BEZANSON *et al.*, 2012).

4.2 `Julia`

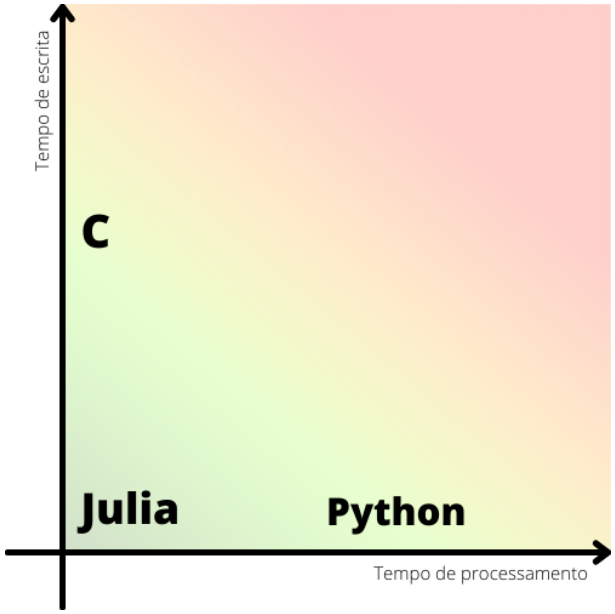
`Julia` vem com a ousada proposta de ser extremamente performática e ao mesmo tempo com uma escrita fluida. Como os próprios idealizadores do projeto dizem – em tradução livre:

> Nós queremos uma linguagem de código aberto, com uma licença liberal. Nós queremos a velocidade do C com o dinamismo do Ruby. Nós queremos uma linguagem que seja homoicônica, com macros verdadeiras como as do Lisp, mas com notação matemática óbvia e familiar como as do Matlab. Nós queremos algo tão utilizável para programação geral como o Python, tão fácil para estatísticas como o R, tão natural para processamento de *strings* como o Perl, tão poderoso para álgebra linear como o Matlab, tão bom para juntar programas como o shell. Algo que seja muito simples de aprender, mas que ainda mantém felizes os hackers mais sérios. Nós queremos que seja interativo e compilado (BEZANSON et al., 2012)[1].

Dados os conceitos dos próprios criadores da linguagem, pode-se elucidar pela Figura 1 a ideia principal de suas palavras a partir de um pequeno diagrama de coordenadas, em que prioriza-se o desempenho sem perder a agilidade de escrita. No presente esquema, a região mais desejada seria a mais próxima da origem possível, representada pela cor verde e, em contra partida, a menos desejada seria a representada pela cor vermelha e mais distante da origem.

[1] No original: “We want a language that’s open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that’s homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.”

Figura 1 – Diagrama de tempo de escrita x tempo de processamento



Fonte: Autoria Própria (2021).

Em alguns *benchmarks* criados pela comunidade é nítida a diferença entre `Python` e `Julia`, para a realização de certas tarefas como ilustrado pela Tabela 1, onde `Julia` se mostra realmente mais rápido em grande parte dos testes.

`Julia` vem evoluindo desde seu lançamento com melhorias contínuas de seu código fonte, que é escrito todo na própria linguagem, e esta recentemente entrou para o grupo dos petaflops, onde estão `C`, `C++` e `Fortran`, três das linguagens mais populares e rápidas existentes atualmente no mundo da ciência de dados (DEVATHON, 2020). Sendo assim, `Julia` provou ser altamente capaz e com um grande potencial futuro, não só na área de ciência de dados, como em diversas outras áreas da computação.

A comunidade `Julia` tem aumentado exponencialmente nos últimos anos e a linguagem vem ganhando bastante território desde então. Assim como a evolução do `Python` se deu, a de `Julia` tem seguido o mesmo caminho, todavia com proporções mais acentuadas devido à democratização da internet, de modo que Fóruns e soluções são criados diariamente. Segundo a Julia Computing (2021) exitem mais de 11,8 milhões de linhas de código em seu registro geral (incluindo documentos e testes). A Figura 2 ajuda na compreensão desse fenômeno ao longo dos anos.

4.2.1 `DrWatson.jl`

Uma das soluções criadas pela comunidade é o alvo desse estudo. A biblioteca `DrWatson.jl`, lançada em 2020, se caracteriza por ser um assistente de projetos científicos em que o usuário possa criar projetos com consistência e rapidez, navegá-los e compartilhá-los com facilidade, gerenciar o código fonte, *scripts*, simulações existentes e pacotes internos

Tabela 1 – Resultado dos testes de benchmark.

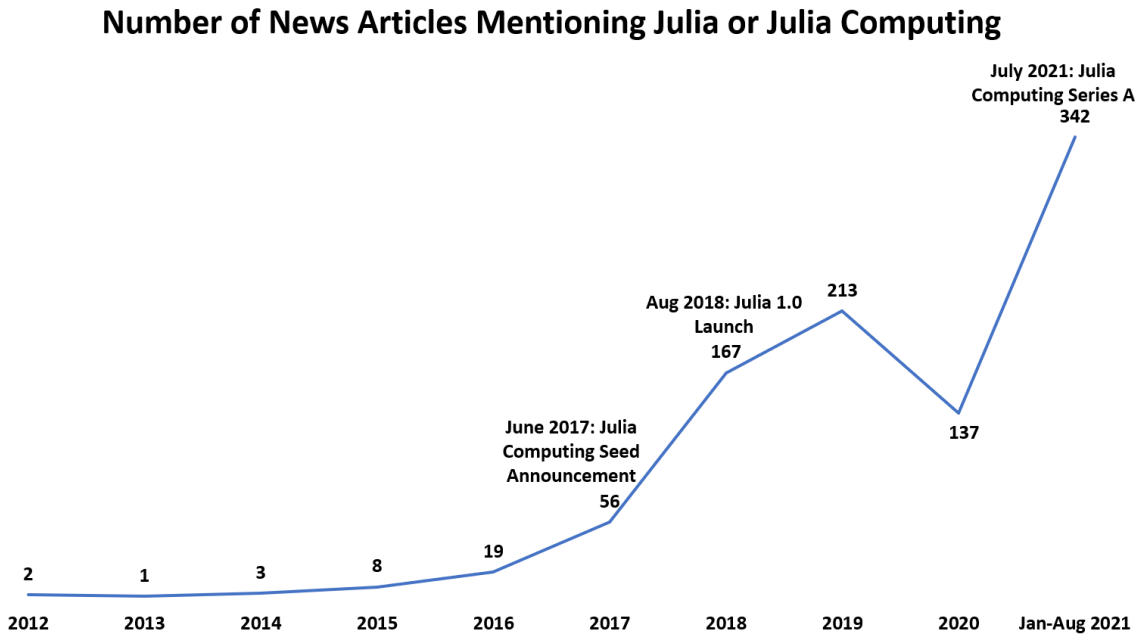
Linguagem	Simulação	Tempo[a]	Memória[b]	CPU 1	CPU 2	CPU 3	CPU 4
Julia	n-body	4,05	218,74	3%	2%	3%	100%
	mandelbrot	1,32	227,836	98%	86%	85%	85%
	spectral-norm	1,11	182,068	80%	79%	80%	98%
	fannkuch-redux	7,64	206,244	95%	95%	94%	95%
	fasta	1,12	193,924	10%	8%	99%	8%
	k-nucleotide	5,12	374,876	63%	72%	38%	49%
	binary-trees	7,43	482,62	69%	100%	78%	72%
	reverse-complement	1,6	677,796	6%	6%	100%	9%
	pidigits	0,97	172,908	10%	98%	10%	9%
	regex-redux	1,68	394,324	99%	7%	5%	71%
Python	n-body	567,56	8,076	0%	0%	0%	100%
	mandelbrot	163,32	12,08	98%	98%	98%	98%
	spectral-norm	120,99	13,424	99%	99%	99%	99%
	fannkuch-redux	352,29	12,232	97%	99%	100%	99%
	fasta	37,32	846,264	10%	67%	83%	30%
	k-nucleotide	46,28	241,108	94%	97%	95%	96%
	binary-trees	48,03	462,732	89%	97%	88%	89%
	reverse-complement	7,2	1.005,184	20%	53%	48%	29%
	pidigits	1,28	12,024	0%	1%	100%	0%
	regex-redux	1,36	111,852	32%	40%	33%	88%

[a] Unidade de tempo medida em segundos, utilizando vírgula como separador decimal, conforme sugere a língua portuguesa.

[b] Unidade de memória medida em Megabyte (MB), utilizando vírgula como separador decimal, conforme sugere a língua portuguesa.

Fonte: Gouy (2021).

Figura 2 – Número de novos artigos mencionando `Julia`



Fonte: Julia Computing (2021).

do projeto a fim de estabelecer reprodutibilidade e e tornar o gerenciamento do projeto científico uma tarefa menos árdua. Seu lema é que o usuário, a partir do uso de alguns comandos e macros, consiga focar mais na ciência e menos em resolver complicações computacionais (DATSERIS *et al.*, 2020a).

`DrWatson.jl` se propõe a criar toda uma arquitetura de projeto composta por diretórios de códigos fonte, dados, documentos, imagens, gráficos, *papers* e *scripts* simplesmente digitando o comando `initialize_project` em conjunto com seus argumentos, e a partir disso outros comandos como `projectdir`, `datadir`, `plotsdir` e similares estarão disponíveis paras serem usados dentro do código de acordo com a necessidade do usuário. Isso criará um ecossistema de onde o usuário poderá executar ações como:

- Controle de versões via `Git`;
- Compartilhamento de repositórios remotos;
- Modelo de estrutura de pastas;
- Gerenciamento de dependências;
- Gerenciamento de dados;
- Rápida ativação de ambiente virtual;
- Criação de *dataframes* com dados de simulações;
- Salvamento de simulações com nomes personalizados.

Ao instanciar o projeto e inserir a macro `@quickactivate "Nome_do_Projeto"` no início de cada *script*, garantirá que todas as suas dependências serão instaladas e ativadas, consequentemente todos os comandos próprios estarão disponíveis para serem usados com o mínimo de esforço (DATSERIS *et al.*, 2020b).

A biblioteca é capaz de realizar tanto simples tarefas de criação de diretórios de projetos, e disponibilização de comandos próprios para o gerenciamento desses diretórios dentro do ambiente de programação, como ainda executar tarefas mais complexas, criando maneiras de salvamento de simulações com seu comando `safesave` que evita a sobrescrição dos dados salvos com o mesmo nome. Segundo Datseris *et al.* (2020a), embora `DrWatson.jl` não seja o único pacote capaz de fazer tais feitos, existindo outros softwares que façam o mesmo serviço com a mesma performance ou até melhor em certos quesitos, ele é o que oferece maior suporte à ampla gama funções exigidas em aplicações científicas em um pacote só, dispensando o uso de várias bibliotecas, sendo fácil de usar e não invasivo.

4.3 Simulações de Ciências Térmicas

Por vezes a realização de cálculos matemáticos manuais de balanço de massa e energia se torna algo muito dispendioso e também complexo, com cenários de resolução de sistemas de equações contendo mais de dez equações, sejam elas equações algébricas ou diferenciais. E é nesse cenário em que a computação numérica programada se insere nas ciências térmicas.

Em ciências térmicas, é comum que o pesquisador utilize linguagens computacionais para simular alguns casos desejados e entender fenômenos naturais. Seja para simular casos de geração de potência, condução de calor, convecção térmica, conforto térmico ou refrigeração.

4.3.1 MEF e Malha

Para qualquer simulação numérica computacional, a resposta exata não é descoberta. O invés disso é criado um algoritmo que executa a discretização de um elemento contínuo pelo Método dos Elementos Finitos (MEF) a fim de encontrar respostas que se aproximem tanto quanto desejado (BATHE, 2007). Nesse tipo de situação, a resolução da malha é o fator mais importante, pois é ela que irá dar precisão aos cálculos e, consequentemente, reduzir os erros.

De acordo com Gomes *et al.* (2012) os resultados obtidos durante a simulação devem ser independentes do tipo de malha utilizada mas sua precisão está diretamente ligada com a qualidade da malha. Porém, algo a ser observado é que quanto mais refinada for a malha, mais poder computacional ela irá demandar e daí se dá a necessidade de uma análise preliminar por parte do pesquisador a respeito da resolução da malha utilizada para que o esforço computacional não seja maior do que o necessário (BATHE, 2007).

Como a resolução da malha é uma entrada manual que normalmente depende muito da experiência do pesquisador, Dyck, Lowther e McFee (1992) utilizaram inteligência artificial para encontrar a melhor condição para a simulação a que se propunha. Mas, ao invés disso, para simulações relativamente pouco custosas computacionalmente, também é possível fazer alguns testes preliminares com vários valores de resolução, objetivando reduzir os erros a um nível aceitável antes de rodar a simulação com uma resolução de malha fixa, no intuito de encontrar respostas de fato.

4.3.2 Planejamento fatorial

Para falar em Planejamento Fatorial é necessário definir antes dois conceitos: níveis e fatores. Níveis podem ser definidos como a quantidade de valores diferentes que uma variável pode assumir e fatores podem ser definidos como as próprias variáveis de entrada do experimento.

O Planejamento Fatorial é uma tática adotada em experimentos quando se quer investigar todas as possibilidades de um espaço de design. Para isso adota-se a regra de que o número de diferentes experimentos possíveis é igual ao produto dos níveis de cada fator (MARINHO; CASTRO, 2005). Ou seja, para se obter o número total de experimentos executados, deve-se multiplicar as possibilidades de cada variável de forma que o nível n de fator 1 é multiplicado pelo nível n de fator 2, que é multiplicado pelo nível n de fator 3, até o nível n do fator k:

$$Experimentos = n_1 \cdot n_2 \cdot n_3 \cdot ... \cdot n_k \tag{4.1}$$

Assim, em uma simulação em que se deseje variar valores de temperatura e pressão, será necessário uma simulação para cada valor de temperatura tal como para cada valor de pressão. Logo, será necessário realizar diferentes combinações de valores de entrada para que seja factível a realização de todos os casos possíveis a serem investigados.

Por exemplo, se os valores de temperatura testados forem entre 500K e 1000K, variando de 50 em 50, serão 11 possíveis simulações com a pressão constante, mas se as pressões também forem variadas dentro de um intervalo, o número de simulações irá aumentar de acordo com a quantidade de valores testados. Matematicamente falando, o número de casos possíveis a serem simulados é igual ao número de valores de temperatura as serem testados multiplicado pelo número de valores de pressão testados. Se forem usados 11 diferentes valores de temperatura e 22 de pressão, serão gerados 242 resultados diferentes no total adotando-se a mesma simulação para todos os casos:

$$Experimentos = n_T \cdot n_P \Rightarrow 242 = 11 \cdot 22 \tag{4.2}$$

Se aplicada a Equação 4.2, de um modo genérico como na Equação 4.1, é possível calcular todos os possíveis casos do planejamento fatorial que serão realizados ao serem alterados os valores de dois ou mais parâmetros de entrada para uma mesma simulação, gerando os respectivos resultados de cada uma e aumentando ou diminuindo as possibilidades de saídas.

4.3.3 Ciclos termodinâmicos

Ciclos reais são difíceis de serem estudados por conta de complexidades presentes no mundo real, como a falta de tempo suficiente para atingir condições de equilíbrio e a dificuldade ou impossibilidade da reprodução de efeitos aleatórios em simulações (ÇENGEL; BOLES, 2013). Para facilitar o estudo analítico deve-se controlar os fatores complicantes e fazer uso de algumas idealizações, pois assim torna-se possível o estudo dos efeitos dos principais parâmetros que regem o ciclo sobre os resultados obtidos sem que o engenheiro se confunda

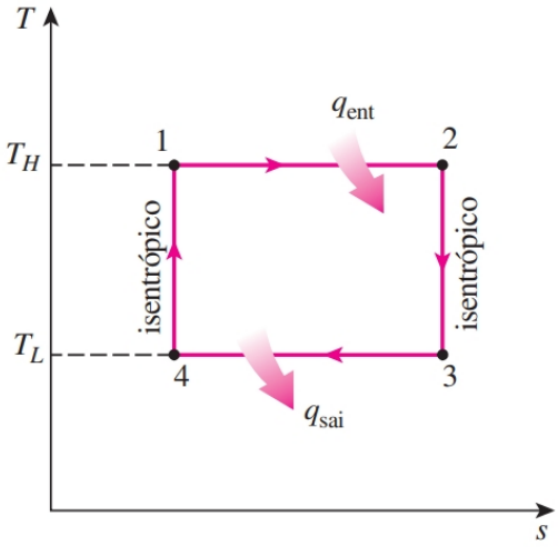
com detalhes. Fazendo assim, com que os estudos do ciclo simplificado também se apliquem a ciclos reais (BRUNETTI, 2018).

As máquinas térmicas, no geral, foram criadas com o objetivo de transformar energia térmica em trabalho através da expansão do fluido de trabalho. E essa transformação é regida pela equação da eficiência,

$$\eta_t = \frac{w_{liq}}{q_{ent}} = 1 - \frac{q_{sai}}{q_{ent}} \quad \text{ou} \quad \eta_t = \frac{W_{liq}}{Q_{ent}} = 1 - \frac{Q_{sai}}{Q_{ent}} \tag{4.3}$$

com argumentos intensivos ou extensivos. Em um diagrama T-s, ilustrado pela Figura 3, a razão entre a área interna do ciclo e a área sob a curva do processo de fornecimento de calor pode, também, expressar essa relação (ÇENGEL; BOLES, 2013).

Figura 3 – Diagrama T-s de um Ciclo de Carnot



Fonte: Çengel e Boles (2013).

4.3.3.1 Ciclo de Carnot

O ciclo teórico mais eficiente possível que pode ser realizado entre uma fonte de calor e um sumidouro com suas respectivas temperaturas T_H e T_L é o Ciclo de Carnot, porém seu funcionamento é significativamente diferente de situações práticas por necessitar tanto de materiais com propriedades inalcançáveis no mundo real, quanto de muito tempo para que os estados entrem em equilíbrio no sistema, o que inviabiliza o seu uso e o torna somente um padrão de comparação. É caracterizado por ser um sistema, ou seja, um ciclo fechado, dividido em quatro processos totalmente reversíveis, sendo o primeiro um fornecimento isotérmico de calor, o segundo uma expansão isentrópica, o terceiro uma rejeição isotérmica de calor e o quarto uma compressão isentrópica (ÇENGEL; BOLES, 2013).

A eficiência térmica aumenta conforme se dá o aumento da temperatura fornecida ao sistema e a diminuição da temperatura rejeitada, porém existem certos limites práticos e teóricos para essas temperaturas. Sendo os práticos, os componentes mecânicos internos para temperaturas mais altas e o ambiente de resfriamento que a máquina térmica está utilizando para temperaturas mais baixas (ÇENGEL; BOLES, 2013). Os teóricos são impostos pela Segunda Lei da Termodinâmica, em que, segundo Kelvin-Planck, "É impossível construir um motor térmico cíclico que transforme em trabalho todo o calor recebido de uma fonte quente". Logo, o calor que sai da máquina não pode ser zero e isso torna impossível a existência de um ciclo com eficiência de 100% (BRUNETTI, 2018).

Em máquinas reais, a expansão do fluido de trabalho é feita pela queima de um combustível dentro das fronteiras do sistema, o que caracteriza um motor de combustão interna. Embora esses tipos de motores operem em um ciclo mecânico, ou seja, o pistão retorna à sua posição inicial a cada revolução, o gás de trabalho não é reutilizado para o início do próximo ciclo como em um sistema fechado. Ele é descartado e outra porção de massa desse fluido é admitida para o reinício do ciclo, caracterizando um volume de controle (ÇENGEL; BOLES, 2013), pois esse gás já utilizado está a uma temperatura bem maior do que a de admissão e seria necessário muito tempo para que ele se resfriasse em volume constante, ou necessitaria de um processo de expansão maior, o que seria inviável dada a arquitetura do mecanismo.

4.3.3.2 Ciclo Otto padrão ar

Sabida a complexidade desses ciclos, como já mencionado anteriormente, devem primariamente ser feitas algumas hipóteses, chamadas de hipóteses do padrão ar, a fim de facilitar o seu estudo e compreensão.

1. O ar é considerado um gás ideal, a baixas temperaturas e em circuito fechado.
2. Todo os processos integrantes do ciclo devem ser reversíveis internamente.
3. A combustão deve ser descrita como um processo de obtenção de calor a partir de uma fonte externa.
4. A exaustão deve ser representada por um processo de retirada de calor até que se tome o seu estado inicial.

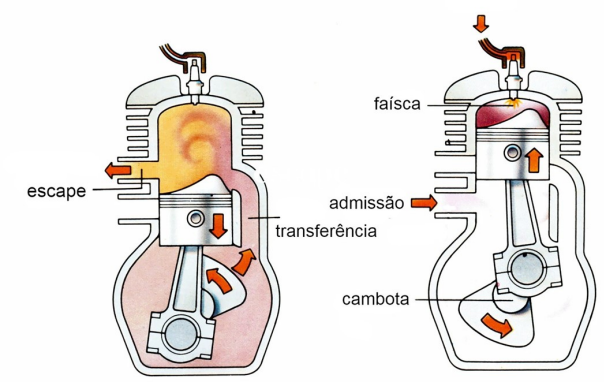
Em ciclos de combustão a pistão existem duas posições fixas principais que ficam se alternando e assim dão movimento à máquina, chamados de Ponto Morto Superior (PMS) e Ponto Morto Inferior (PMI), em que a primeira refere-se ao ponto em que há o menor volume dentro do cilindro, quando o pistão está na parte superior do cilindro, e a segunda refere-se ao ponto de maior volume dentro do cilindro, quando o pistão está na parte inferior do cilindro. A distância entre esses dois pontos é chamada de curso do motor e representada pela letra S.

O volume formado entre o cabeçote do motor e o PMS é chamado de volume morto (V_{PMS}), o volume entre o PMS e o PMI é chamado de volume deslocado unitário (V_{du}), logo o volume deslocado pelo motor a cada volta do virabrequim é chamada de volume deslocado (V_d) e a relação entre o volume do PMI (também chamado de volume máximo deslocado) e o volume morto é chamada de razão de compressão, simbolizada pela letra r (BRUNETTI, 2018). Deste modo, tem-se que:

$$r = \frac{V_{max}}{V_{min}} = \frac{V_{PMI}}{V_{PMS}} \tag{4.4}$$

O ciclo onde ocorre a ignição por centelha, é chamado de Ciclo Otto. Os motores deste ciclo podem ter Dois Tempos (2T) ou Quatro Tempos (4T), como podem ser ilustrados pelas figuras 4 e 5, respectivamente. Obviamente que para estudos também são feitas hipóteses de simplificação, como a admissão somente de ar como um gás ideal, com compressão e expansão isentrópica e entrada e saída instantânea de calor (processo isocórico).

Figura 4 – Ciclo de combustão 2T



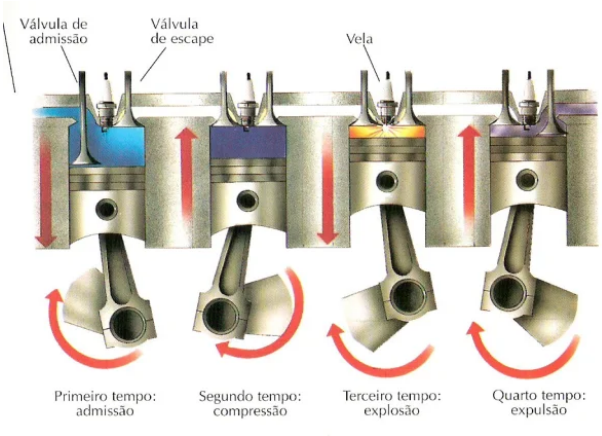
Fonte: Pereira (2019).

Os motores 4T são chamados assim porque cada movimento que o pistão faz em um ciclo, seja para baixo ou para cima, é chamado de tempo. Logo, nesses tipos de motores, tem-se um movimento em que o pistão admite a mistura ar-combustível para dentro do cilindro através da abertura da válvula de admissão, o próximo em que essa válvula é fechada e ele comprime a mistura, o seguinte em que é feita a ignição da mistura por meio de uma centelha emitida pela vela e o último em que os gases de combustão são ejetados da câmara por meio da abertura da válvula de exaustão, para que outro ciclo se inicie (JUNIOR, 2003).

Já nos motores 2T, pode-se perceber que o motor consegue realizar um ciclo completo na metade do tempo que o 4T, onde a mistura é admitida no cárter pelo próprio movimento de compressão do pistão do ciclo anterior ao que está para se iniciar. Em seguida, com a combustão do ciclo anterior, a mistura é pressurizada ainda no cárter pelo próprio movimento descendente do pistão. Ao ser liberado o duto de saída da câmara de combustão, os gases queimados são libertos e logo após o duto de admissão é aberto, onde ocorre a entrada da mistura por diferença de pressão para o início do próximo ciclo. Tudo isso ocorre em uma subida

e uma descida do pistão dentro do cilindro. Enquanto que no 4T, o ciclo acontece em duas subidas e duas descidas alternadas do pistão (BRUNETTI, 2018).

Figura 5 – Ciclo de combustão 4T

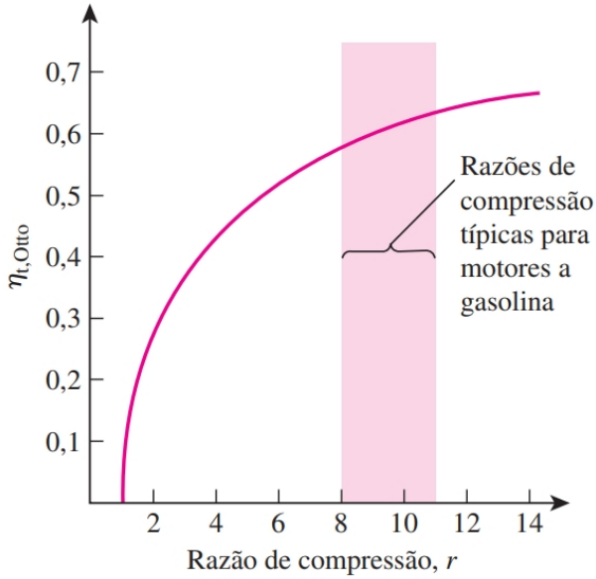


Fonte: Gibi (2013).

Nos ciclos 4T os tempos são nomeados de tempo de admissão, compressão, explosão e exaustão, enquanto que nos ciclos 2T, são nomeados de tempo motor e tempo de compressão.

Os motores de Ciclo Otto adotam uma razão k dos calores específicos c_p/c_v e da mesma forma que a eficiência térmica estava ligada às temperaturas de entrada e saída dos gases no Ciclo de Carnot, aqui η_t, ainda está diretamente ligada tanto às temperaturas como também aos coeficientes k e r, como pode ser observado pela Figura 6 para um $k = 1{,}4$, sendo k o valor da razão dos calores específicos do ar à temperatura ambiente.

Figura 6 – Gráfico r x η_t para $k = 1{,}4$



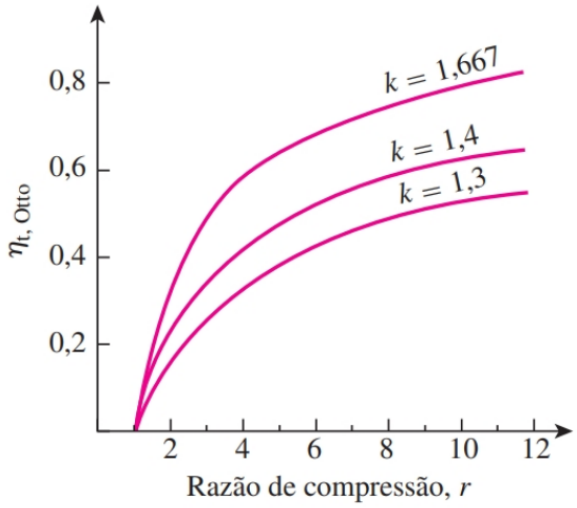
Fonte: Çengel e Boles (2013).

A partir da observação da Figura 6, pode-se também, perceber que a inclinação da curva de eficiência térmica é bastante acentuada para razões de compressão relativamente

mais baixas e seu coeficiente de crescimento vai decaindo conforme as razões de compressão vão aumentando (ÇENGEL; BOLES, 2013). Porém, assim como a temperatura do ciclo tem um limite físico para com os materiais do motor em situações reais, a razão de compressão, em ciclos reais, também não pode ser aumentada indefinidamente como nos Ciclos Otto teóricos, pois os combustíveis em geral, possuem um limiar de autoignição que não deve ser ultrapassado a fim de evitar problemas mecânicos gerados pelas famosas "batidas de pino", como são popularmente conhecidas as autoignições de motores Ciclo Otto e que podem danificar severamente o bom funcionamento da máquina (BRUNETTI, 2018).

Além disso, também é possível perceber a partir da Figura 7 que quanto maior o valor de k, maior também será o valor da eficiência térmica. Como os valores de k estão diretamente ligados ao tamanho das moléculas dos gases de trabalho, gases monoatômicos como argônio e hélio têm um coeficiente k de 1,667, já gases com moléculas maiores como CO_2 ou até misturas como o ar, têm um coeficiente k menor e, consequentemente, também uma eficiência térmica menor.

Figura 7 – Gráfico r x η_t para vários valores de k



Fonte: Çengel e Boles (2013).

Nos motores reais, que operam em condições de trabalho reais, utilizando o ar (uma mistura de vários gases atmosféricos) como fluido de trabalho, ocorre uma redução significativa na razão de calores específicos. Ademais, à medida que a temperatura aumenta, essa razão diminui ainda mais. De acordo com Çengel e Boles (2013), os motores reais a combustão interna que seguem o Ciclo Otto apresentam uma eficiência de apenas 25 a 30%.

Para concluir esse pensamento sobre fatores que diminuem a eficiência de uma máquina térmica, pode-se citar ainda a fração residual de gases $\overline{f}$, em que ao final de toda combustão há o processo de escape dos gases queimados. Porém uma pequena fração de massa desses gases ainda permanece dentro da câmara de combustão, o que acaba por derrubar ainda mais

a eficiência térmica da queima, já que uma quantidade menor de combustível pode ser admitida para a queima (BRUNETTI, 2018). Essa quantidade é dada por:

$$\bar{f} = \frac{m_{res}}{m_{tot}} = \frac{m_{res}}{m_{ar} + m_{comb} + m_{res}} \tag{4.5}$$

4.3.3.3 Ciclo Otto FTHA

Apesar de ciclos como Carnot e Otto serem bastante úteis em construir uma compreensão conceitual sobre máquinas térmicas e as influências dos parâmetros básicos no seu comportamento, ainda estão um tanto quanto longe dos resultados esperados para ciclos reais. Além disso, apresentam limitações em relação aos parâmetros de entrada, o que acaba por diminuir a abrangência dos testes e resultados.

Em busca de maiores precisões nos estudos de modelos padrão a ar, algo que se pode fazer é, aos poucos, introduzir novas hipóteses em algumas das idealizações feitas para evitar tratar efeitos como os da combustão e acrescentar parâmetros geométricos do motor aos cálculos, como as relações de biela-manivela e de curso. Sendo assim, a fim de aproximar cada vez mais o modelo de um ciclo real, tais esforços foram feitos no intuito de atingir uma complexidade intermediária entre os ciclos Otto padrão ar e Ar-Combustível (ciclo com maior complexidade e mais próximo do real).

O Ciclo Otto de Adição de Calor de Tempo Finito, do inglês Finite-Time Heat Addition (FTHA), é uma complexificação do modelo tradicional padrão ar, que inclui a discretização do tempo da combustão e a inclusão de parâmetros geométricos do motor. Além disso, o FTHA também leva em conta o mecanismo biela-manivela, a adição extensa de calor, que é modelada por meio de uma função de liberação de calor cumulativa em dependência do tempo, e os parâmetros de ângulo do virabrequim, permitindo a contabilização de atributos como tempo de ignição e mecanismos de potência (NAAKTGEBOREN, 2017).

Uma das principais vantagens do FTHA é sua capacidade de considerar os efeitos da rotação do motor (N) sobre o trabalho, a temperatura de escape, a pressão média eficaz e a eficiência térmica. Ao contrário do ciclo Otto padrão ar, que desconsidera esses efeitos, o FTHA é capaz de auxiliar na modelagem dessas hipóteses e, consequentemente, na otimização do desempenho do motor (NAAKTGEBOREN, 2017).

No Ciclo Otto FTHA, ao contrário dos modelos como o Ciclo Otto padrão ar, a pressão, o volume e a temperatura variam durante todo o processo de adição de calor, transformando os processos isocóricos e isobáricos (no caso de ciclos Diesel) em processos politrópicos de coeficientes n, com valores que podem variar de $-\infty$ a $+\infty$, dependendo da hipótese adotada (BRUNETTI, 2018).

O modelo é subdividido nos quatro processos canônicos de compressão, adição de calor, expansão e rejeição de calor, com esse último processo sendo idêntico ao do ciclo Otto padrão ar. Os outros são discretizados e subdivididos em pequenos subprocessos politrópicos

de acordo com o tamanho da malha. A determinação dos expoentes politrópicos dos pequenos subprocessos é realizada a partir do algoritmo de resolução. Para isso, utiliza-se a Primeira Lei da Termodinâmica em iterações computacionais pelo método de Newton-Raphson. Esse método parte de uma resposta previamente conhecida e busca descobrir a próxima resposta que deve convergir em uma margem de erro aceitável (NAAKTGEBOREN, 2017).

5 MATERIAIS E MÉTODOS

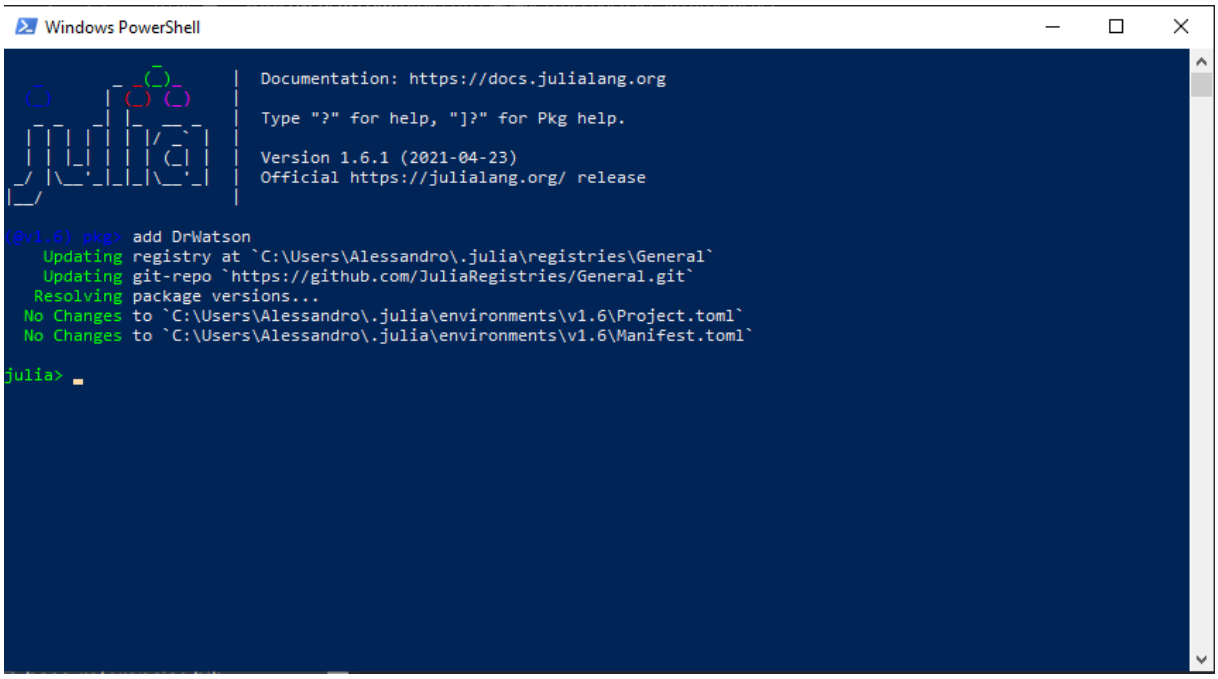
Para esse trabalho, o objetivo é realizar uma pesquisa qualitativa de caráter investigativo a respeito das funcionalidades oferecidas pela biblioteca `DrWatson.jl` seguindo a presente metodologia de pesquisa.

Após a leitura da documentação da biblioteca e algumas partes de seu código fonte para encontrar algumas respostas sobre dúvidas levantadas, consulta de referências tanto a respeito do próprio pacote estudado quanto da base teórica a cerca das simulações a serem executadas, instalação de dependências e preparação do ambiente computacional, foi dado início às atividades práticas e testes.

5.1 Procedimento de criação do projeto

Primeiramente, em ambiente `Windows`, foi instalada uma aplicação que emula uma máquina virtual com o ambiente Linux `Ubuntu` para testes de reprodutibilidade (um dos testes a serem executados em relação ao módulo `DrWatson.jl` estudado), em seguida a linguagem `Julia` foi instalada na máquina `Windows` juntamente com a biblioteca `DrWatson.jl` em ambiente global, como mostra a Figura 8. Por fim, para finalizar o *setup* inicial das máquinas, os mesmos passos da instalação da linguagem e suas dependências foram repetidos de forma análoga para o ambiente `Ubuntu`.

Figura 8 – Instalação da biblioteca em ambiente global

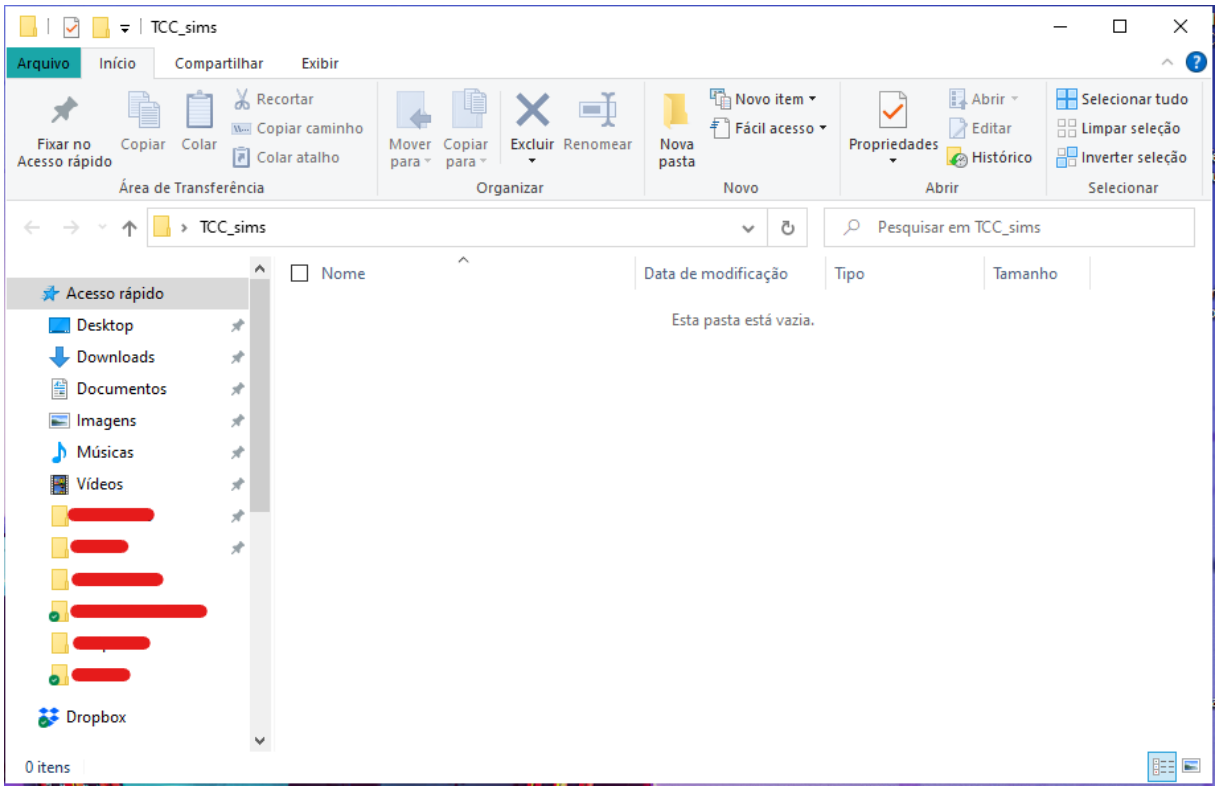


Fonte: Autoria própria (2022).

5.1.1 Inicialização do projeto dentro de um diretório

Após a instalação da biblioteca nas dependências globais da linguagem no computador, para a criação da arquitetura padrão do projeto pelo `DrWatson.jl`, um diretório foi escolhido pelo autor para sedear essa criação como mostra a Figura 9. Lembrando que não é necessariamente obrigatório que o diretório esteja vazio como na Figura 9, podendo existir algumas outras pastas e arquivos, porém é necessário que não exista nenhuma outra pasta com o nome que se deseja dar ao projeto, pois caso exista é necessário incluir o argumento `force=true` ao comando de criação para que essa pasta seja substituída por completo pela pasta do projeto com sua arquitetura padrão.

Figura 9 – Pasta vazia

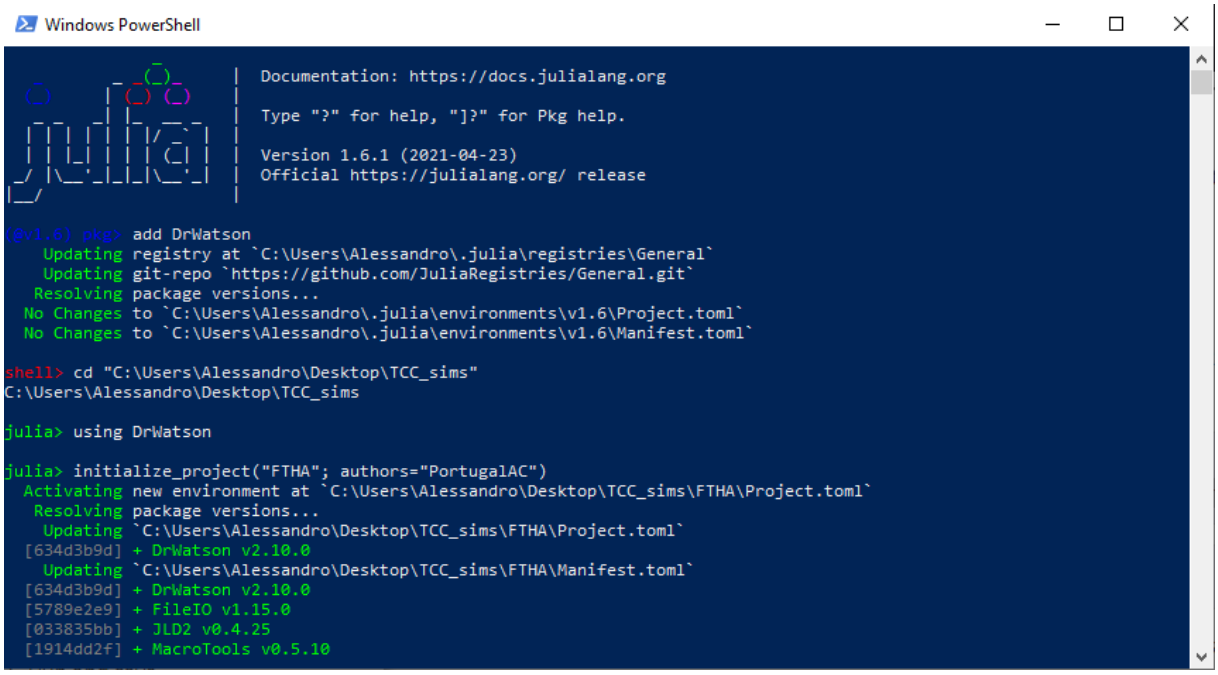


Fonte: Autoria própria (2022).

Em seguida, o diretório foi trocado para dentro do caminho desejado (como manda a documentação), a biblioteca `DrWatson.jl` é chamada do ambiente global e é executado o comando `inicialize_project()` com os argumentos do nome do projeto e autor, respectivamente, e o argumento opcional `force=true`, caso necessário, como ilustra a Figura 10.

Por fim, foi feita a conferência das dependência locais do projeto, as quais são diferentes das dependências globais da linguagem por se tratar de um diretório que contém as próprias dependências com suas respectivas versões. Quando esse ambiente virtual é ativado, é isolado das dependências instaladas na máquina.

Figura 10 – Execução do comando de criação do diretório do projeto com arquitetura padrão



Fonte: Autoria própria (2022).

A partir da visualização das figuras 11 e 12, respectivamente, pode-se perceber que nas dependências globais da linguagem existem mais bibliotecas instaladas do que nas dependências locais do projeto, onde existe apenas a `DrWatson.jl` já que o projeto acaba de ser criado.

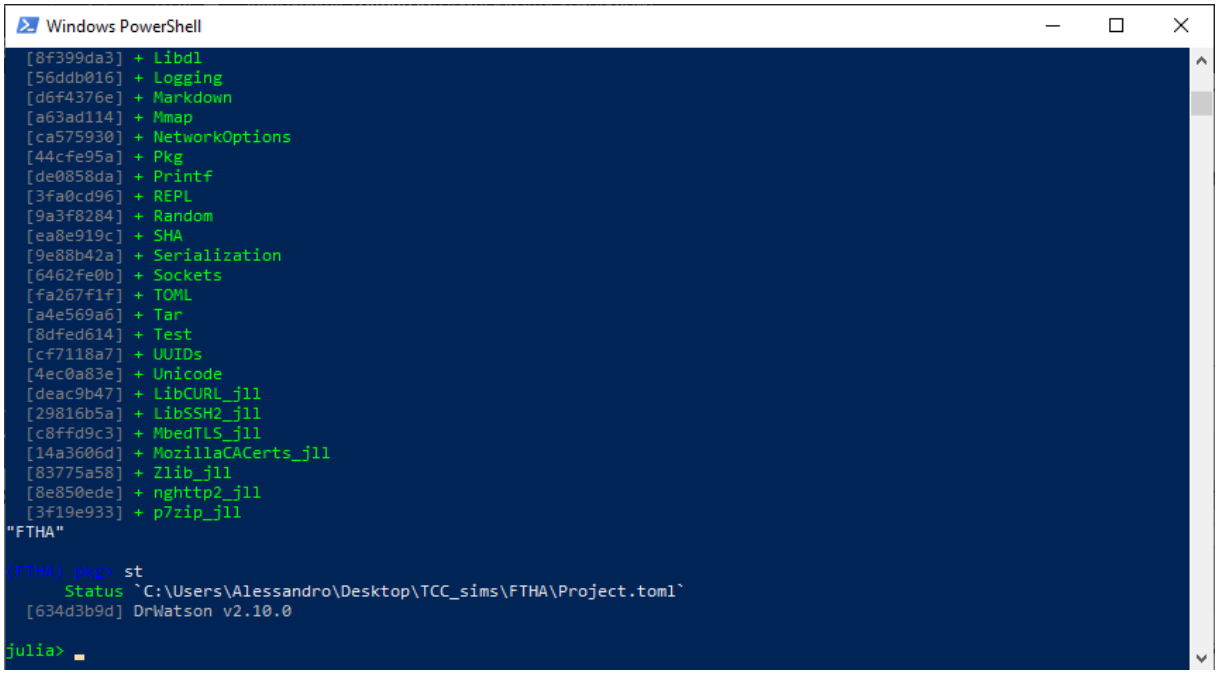
Após a criação bem sucedida do diretório do projeto, é possível observar através da Figura 13 que além de criar toda uma arquitetura estruturada e organizada do projeto, `DrWatson.jl` foi além e transformou uma simples pasta do projeto em um repositório `Git` para que o pesquisador possa fazer uso de ferramentas de versionamento em sua pesquisa. Sendo assim, o software `Visual Studio Code` da `Microsoft`, juntamente com as plataformas `Git` e `GitHub`, foi utilizado para fazer a edição do código das simulações e o gerenciamento do controle de versões em máquina e em nuvem.

5.2 Procedimento de edição dos códigos

Depois das etapa de criação do projeto, é iniciada a etapa de adição de bibliotecas que serão usadas na execução dos códigos e a própria edição dos códigos para que os recursos de `DrWatson.jl` sejam implementados e testados. Foram utilizadas as seguintes dependências nesse projeto:

- `CSV v0.10.4`;
- `DataFrames v1.3.4`;

Figura 11 – Término da execução do comando de criação do diretório do projeto e dependências locais do projeto

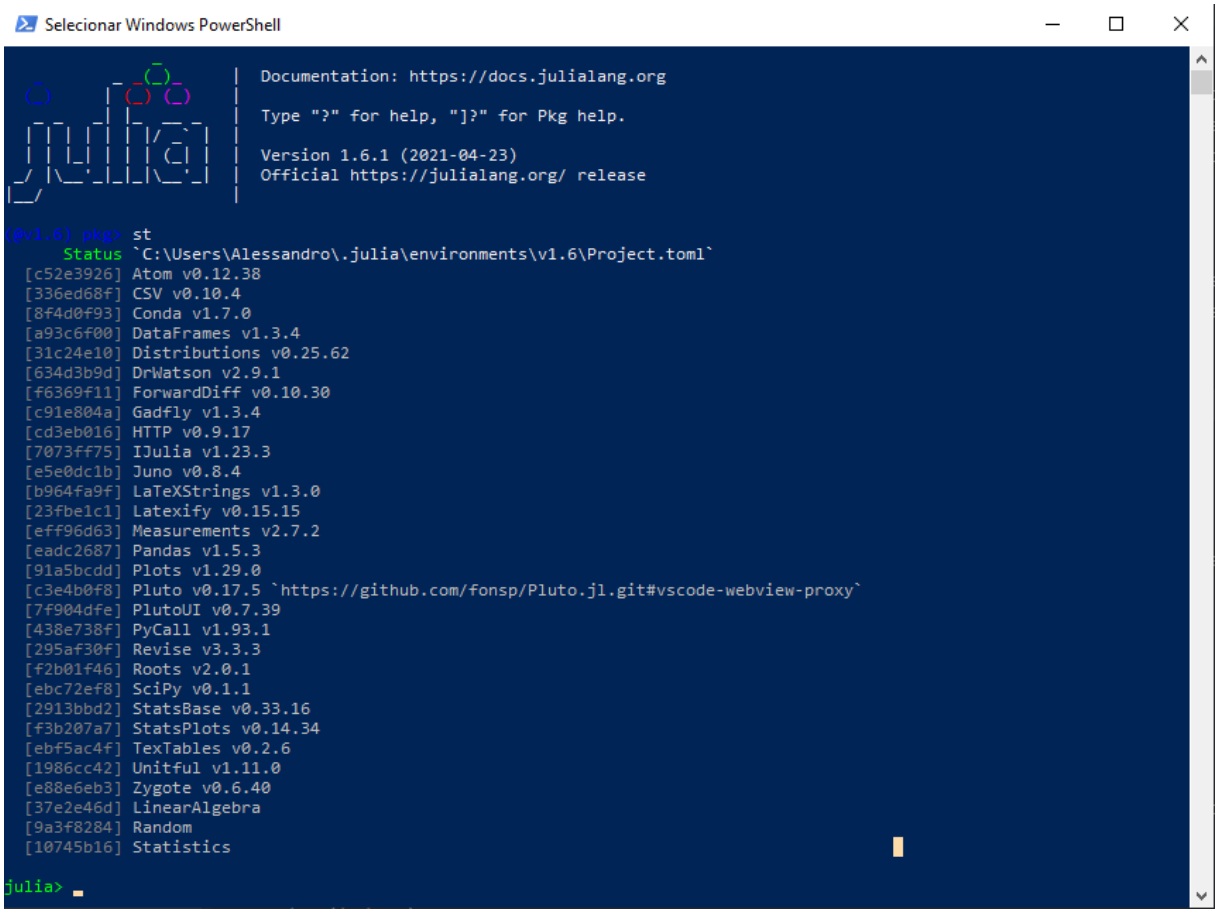


Fonte: Autoria própria (2022).

- `DrWatson v2.9.1;`
- `FTHAlib v0.1.0 'src\FTHAlib';`
- `IJulia v1.23.3;`
- `LaTeXStrings v1.3.0;`
- `Measures v0.3.1;`
- `Plots v1.29.1;`

Embora as duas simulações usadas no experimento envolvam a execução do mesmo código, cada uma tem um objetivo final diferente. Ambas as aplicações utilizam a biblioteca local `FTHAlib.jl`, desenvolvida para resolver o ciclo Otto FTHA. No entanto, a primeira aplicação realiza um teste de malha para determinar os valores de resolução adequados para a discretização angular, garantindo que o erro entre os resultados seja aceitável. Enquanto a segunda aplicação emprega a mesma biblioteca para simular diversas situações de funcionamento do motor, variando o avanço de ignição e a rotação do virabrequim. O objetivo é realizar uma análise posterior e encontrar a otimização para cada rotação em relação aos diferentes avanços de ignição.

Figura 12 – Dependências globais da linguagem



Fonte: Autoria própria (2022).

5.2.1 `FTHAlib.jl`

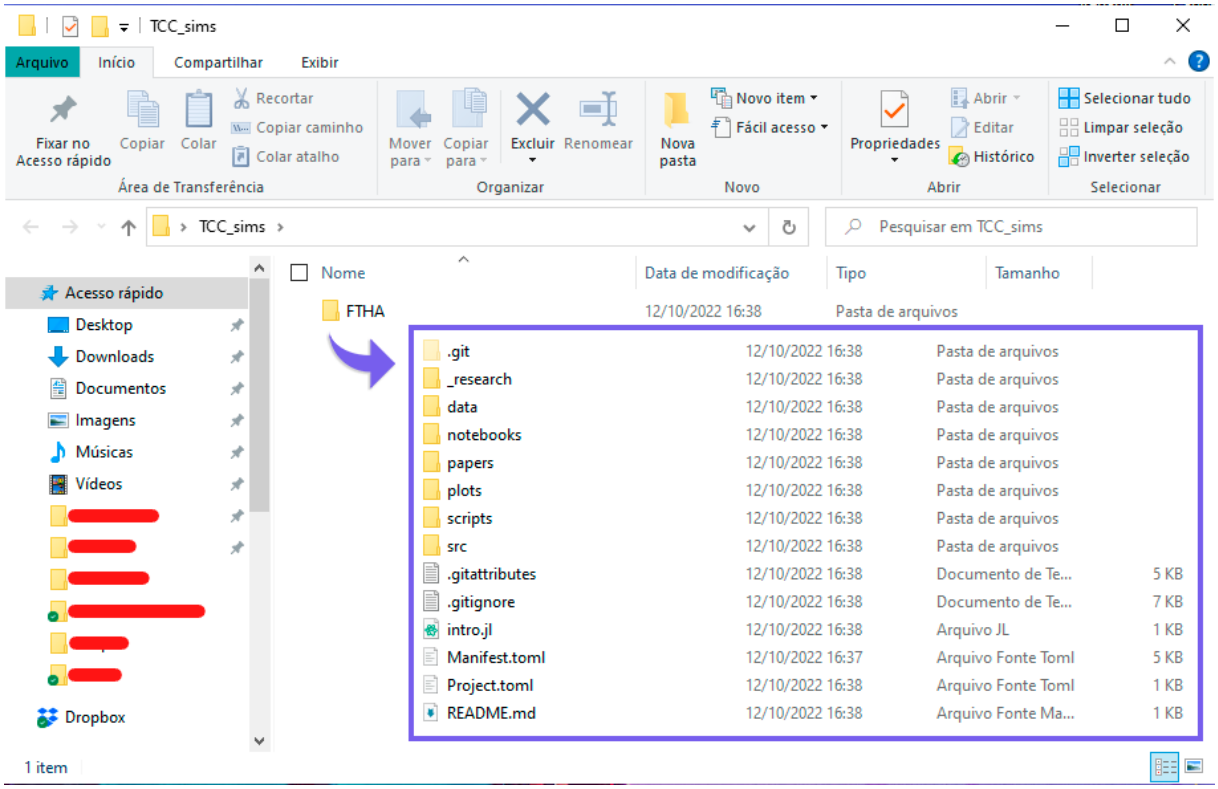
A biblioteca `FTHAlib.jl` foi escrita também em linguagem `Julia` com o objetivo de incluir funções que resolvam equações, façam iterações de convergência através de loops, forneçam respostas diretas e disponibilizem *features* para o usuário utilizar em suas simulações de Ciclo Otto. A partir daí, ele é capaz de encontrar o trabalho do ciclo, eficiência térmica, razão de consumo, as características de cada estado da malha, entre outras facilidades disponibilizadas apenas inserindo os argumentos necessários nas funções certas.

A validação da biblioteca foi feita através de um gabarito de respostas presente no Anexo A, ao final do trabalho, e a biblioteca foi executada diversas vezes até que as respostas atingissem um grau muito baixo de erro em comparação com o gabarito. Ao encontrar essa suficiência, a construção da biblioteca foi dada como concluída e pronta para ser usada nas simulações.

A biblioteca `DrWatson.jl` auxiliou na criação da biblioteca `FTHAlib.jl` com as seguintes funções:

- `@quickactivate`: ativa o ambiente virtual do projeto e apenas usa as dependências locais ao invés das globais da linguagem;

Figura 13 – Arquitetura do diretório do projeto



Fonte: Autoria própria (2022).

- `projectdir()`: retorna o diretório onde o projeto está salvo;
- `datadir()`: retorna o diretório onde os dados das simulações estão salvos;
- `@unpack`: retorna os valores armazenados dentro de um *container* para o ambiente global com os respectivos símbolos sendo traduzidos para variáveis;
- `@dict`: faz o contrário do `@unpack`, aloca parâmetros dentro de um *container* com os símbolos idênticos às variáveis;

5.2.2 Simulação de teste de malha

Previamente ao se iniciar uma simulação numérica em que há a discretização de algo contínuo, é interessante que se faça o teste de malha antes de dar início ao processo de simulação propriamente dito, objetivando-se prevenir erros associados à falta de refinamento da malha. Se houver uma divisão em poucos pontos, tais pontos ficarão muito afastados uns dos outros, gerando erros. Por outro lado, se houver uma divisão em muitos pontos, além de exigir muito poder computacional, a diferença nos resultados para outra simulação relativamente tão precisa quanto, mas com menos pontos, é muito pequena.

Assim sendo, uma das ferramentas de `DrWatson.jl`, que realiza planejamento fatorial, foi aplicada através da biblioteca `FTHAlib.jl` na simulação `FTHA_simulation.jl`, a qual simula o funcionamento do motor, com o propósito de executar o teste de malha com as seguintes resoluções de malha disponíveis no arquivo de entradas `M_entries.jl`: 0,1°, 0,2°, 0,3°, 0,6°, 1°, 2°, 3°, 6°, 10°, 20°, 30°, 60°, a fim de entender qual seria o limiar de refinamento necessário para que a simulação definitiva pudesse ser feita com um erro médio aceitável. Ao final desse teste, como pode ser visto na Figura 14, foi encontrado que abaixo de 10° de resolução de malha o erro percentual já é considerado aceitável por ser menor que 1%.

Após a realização desses testes, a simulação do Ciclo Otto FTHA pode ser executada com segurança e com a certeza de que os valores encontrados estariam alinhados com o esperado.

A partir da visualização da Figura 15, é possível perceber o impacto da resolução da malha e seu refinamento sobre o gráfico de pressão em função do volume específico, pois ao refinar a malha, percebe-se um arredondamento das curvas do ciclo e consequentemente uma maior precisão no cálculo do trabalho e da eficiência térmica da máquina. Ressaltando que o trabalho negativo, apontado pela legenda dos gráficos exibidos na figura, significa que o sentido do fluxo da energia do sistema é direcionado para fora dele.

5.2.3 Simulação Ciclo Otto FTHA

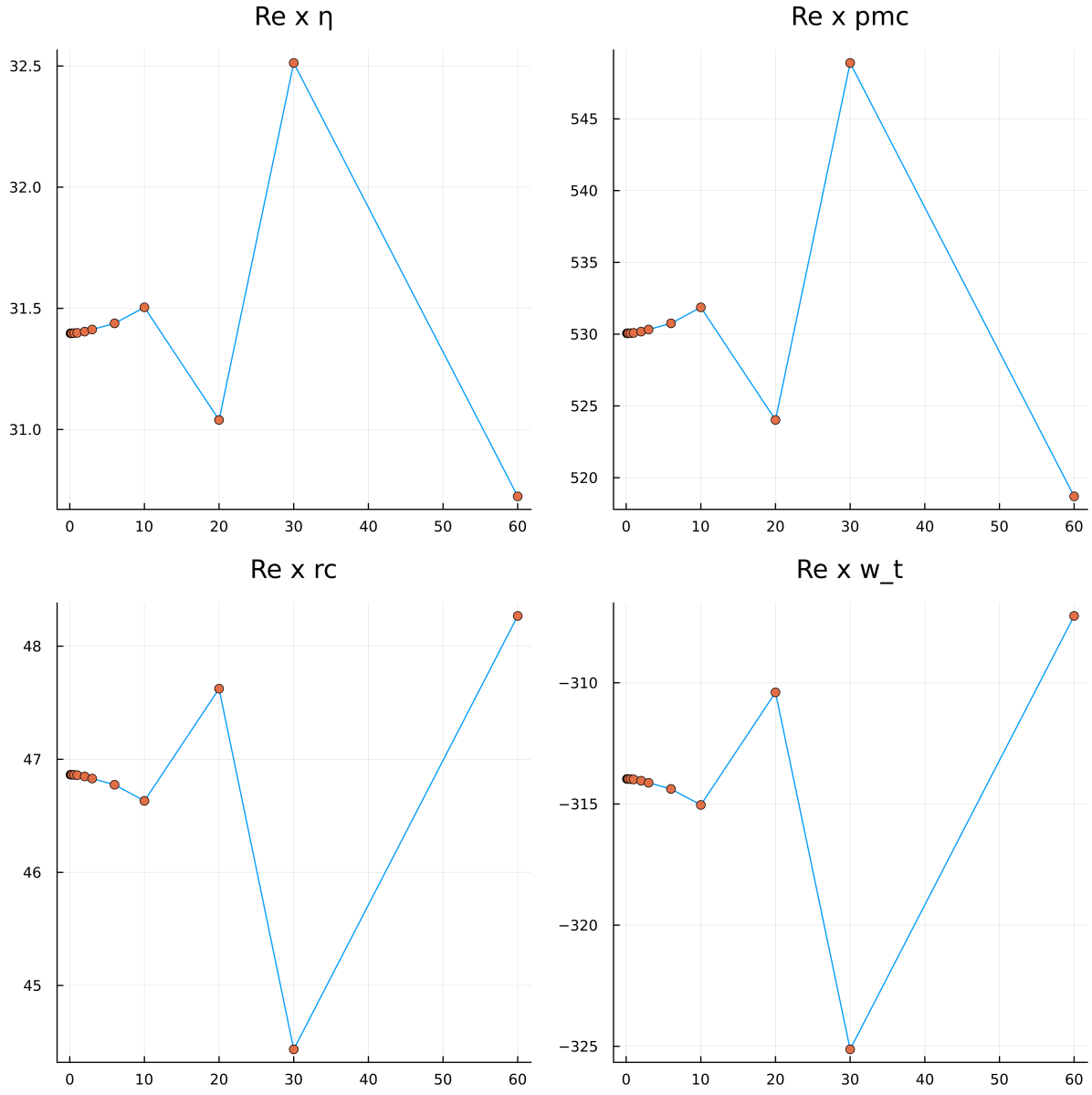
Na presente simulação, o objetivo foi explorar a variação do avanço de ignição, em graus, conjugado com a velocidade angular, em RPM, do virabrequim do motor a fim de encontrar uma otimização entre esses dois fatores que impactam diretamente na eficiência térmica da máquina.

Na análise que estamos considerando agora, o arquivo `RA_entries.jl`, com os dados de entrada da simulação, foi executado no mesmo código que a análise anterior (`FTHA_simulation.jl`) utilizando novamente a biblioteca `FTHAlib.jl` e gerando como saída um gráfico de três eixos plotados na forma de *heatmap*, o que revelou um interessante gradiente de eficiências térmicas em que os picos apresentam um comportamento quase linear.

Os dados de entrada, variados segundo o panejamento fatorial, foram o ângulo de avanço de ignição (θ), variando de -75° a 0° em passos de 5° e a velocidade angular do virabrequim (N), variando de 600 RPM a 5000 RPM em passos de 200 RPM. Os demais parâmetros utilizados na simulação foram mantidos constantes tal como cita o Anexo A.

A partir da Figura 16 é possível identificar através dos resultados dessa simulação que a eficiência aumenta conforme a velocidade angular do virabrequim e o avanço diminuem. A melhor eficiência é encontrada em baixas rotações com um ângulo de avanço por volta de 10° e 5°.

Figura 14 – Gráficos dos erros associados a cada variável de acordo com a resolução da malha



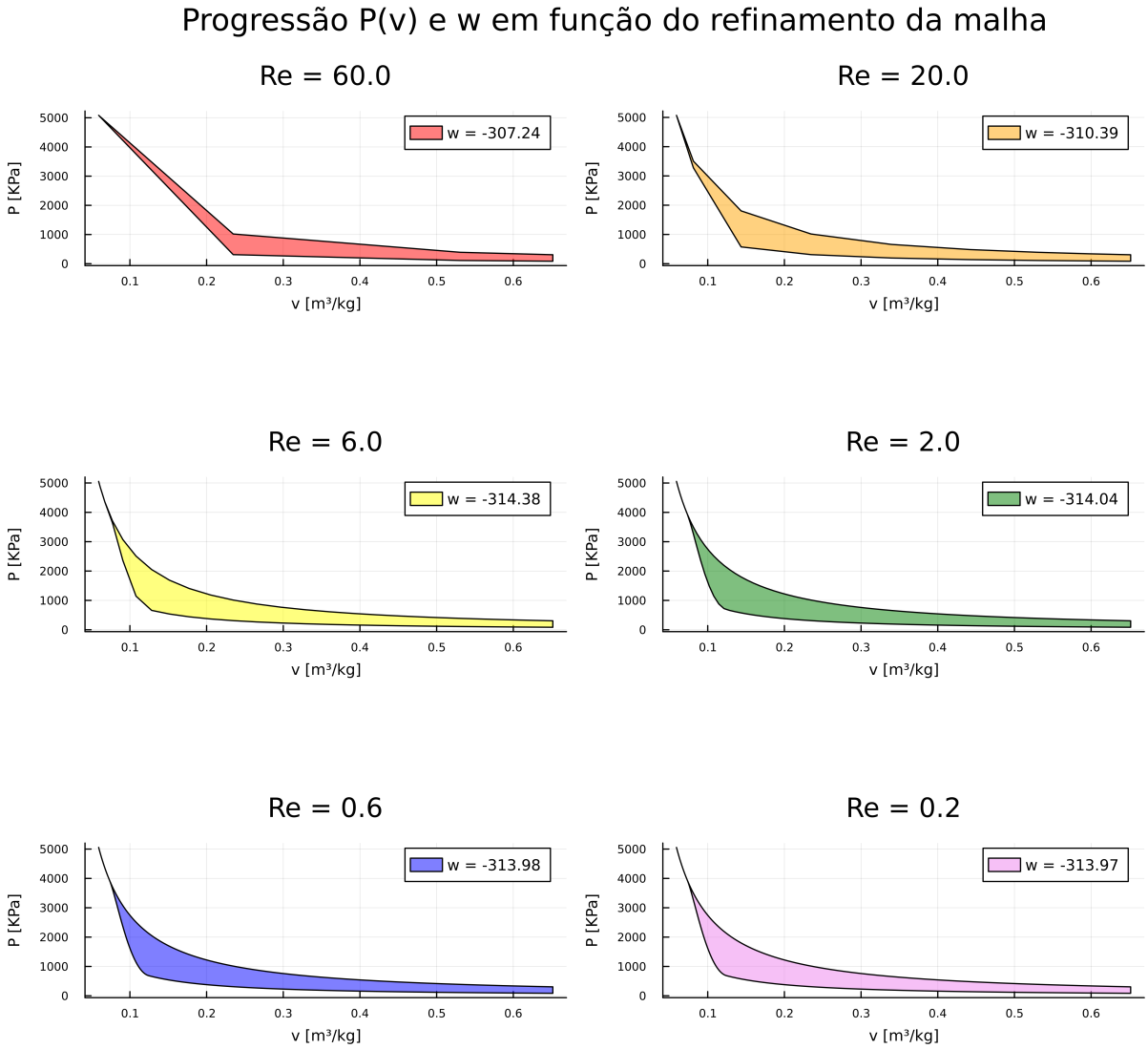
Fonte: Autoria própria (2022).

Já a partir da visualização da Figura 17, é possível observar, com base nas curvas de nível, as delimitações exatas do gradiente de eficiências e qual é a sua tendência de crescimento, visto que a análise busca a otimização.

O algoritmo dessa simulação funciona da seguinte forma:

1. O projeto é ativado utilizando `@quickactivate`;
2. As bibliotecas são importadas para dentro do *script*;
3. Algumas configurações iniciais do projeto são feitas;
4. As entradas iniciais são inseridas em suas respectivas variáveis;

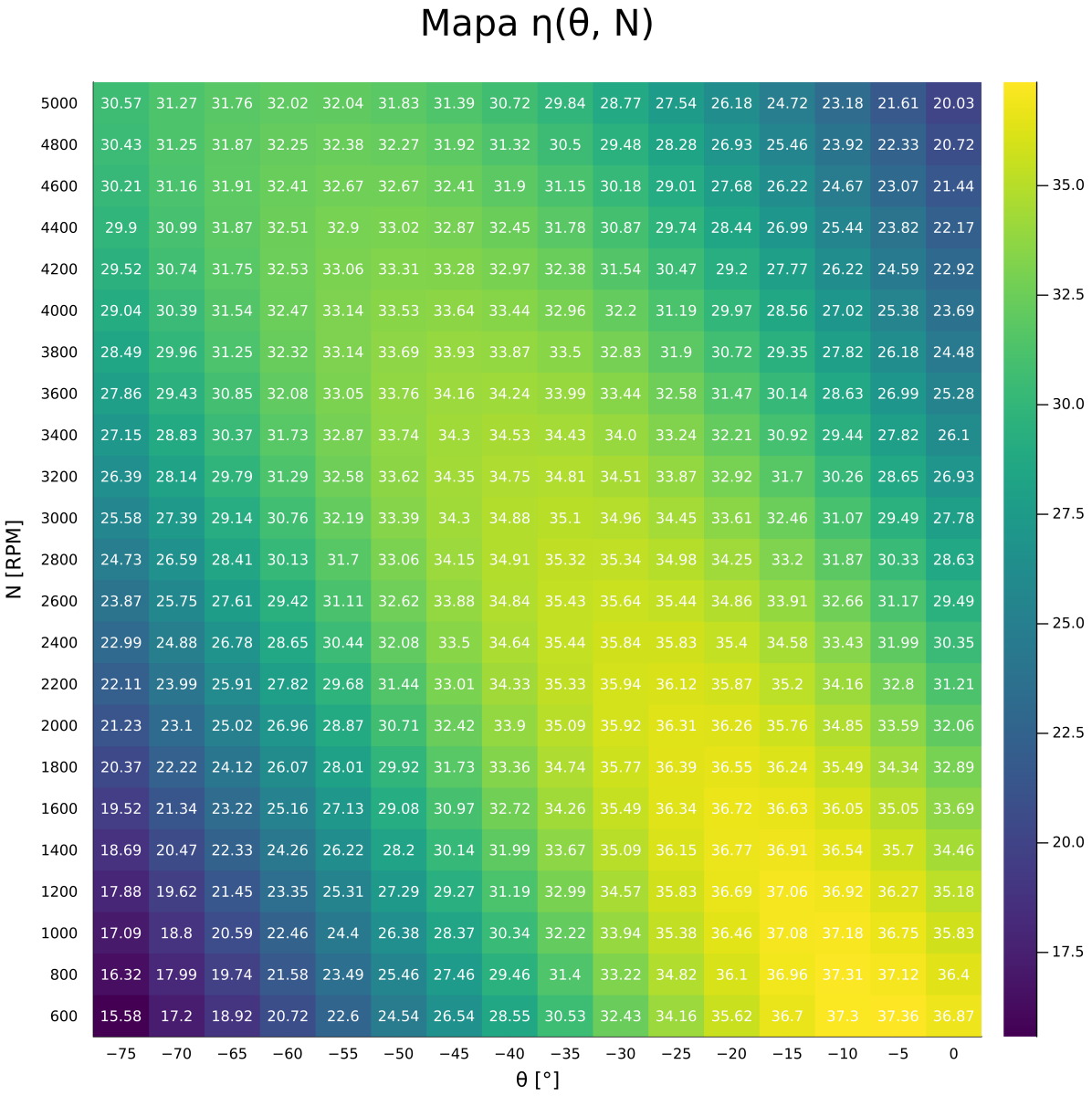
Figura 15 – Gráficos Pv de acordo com as possíveis resoluções e seus respectivos valores de trabalho segundo às áreas dos gráficos



Fonte: Autoria própria (2022).

5. Os parâmetros de entrada são armazenados dentro de um dicionário por meio da macro `@dict`;
6. É feito o processo de planejamento fatorial para todos possíveis casos a serem simulados;
7. A função `produce_or_load()` analisa se alguma simulação idêntica à que está para ser executada já foi feita. Se sim, apenas carrega os resultados, se não, executa a simulação;
8. São inseridas as funções e equações que serão usadas posteriormente na simulação;

Figura 16 – Heatmap do mapa de eficiências em função do avanço de ignição e da velocidade angular do virabrequim



Fonte: Autoria própria (2022).

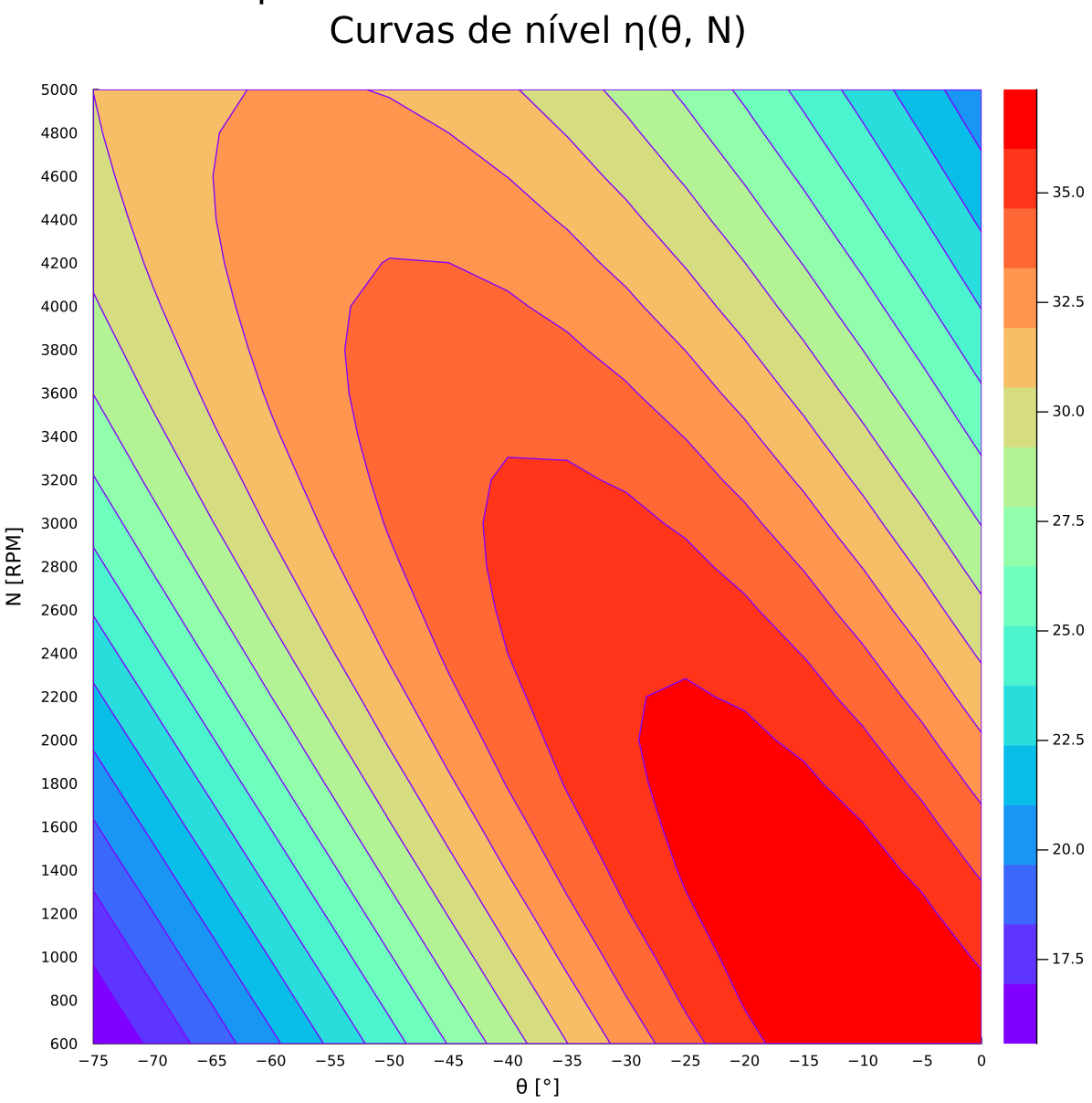
9. São executados os loop para a criação das discretizações dos ângulos, volume específico, curso de pistão e calor específico que entra no sistema;

10. É executado o loop principal de solução do estado de cada ponto discretizado para encontrar os valores de u, w, n, P e T (energia interna específica, trabalho específico, coeficiente politrópico, pressão e temperatura, respectivamente);

11. As respostas são calculadas e armazenadas nas variáveis;

12. Os passos posteriores ao planejamento fatorial se repetem através de um loop até a simulação do último caso definido;

Figura 17 – Curvas de nível de eficiências em função do avanço de ignição e da velocidade angular do virabrequim



Fonte: Autoria própria (2022).

13. Resultados são salvos na pasta `data` por meio da macro `@tagsave` e da função `datadir()`

6 RESULTADOS

Neste capítulo serão expostos os resultados obtidos através das simulações efetuadas utilizando e testando as funcionalidades da biblioteca `DrWatson.jl`, aplicadas às simulações de teste de malha e de otimização da eficiência térmica em função do avanço de ignição e velocidade angular do virabrequim. Foi explorado o máximo possível dos recursos disponibilizados pelo pacote para entender o que ele teria a oferecer em relação a facilitar o trabalho científico que por vezes é repetitivo e exaustivo para o pesquisador.

Algo importante a ser citado é que apesar de as simulações citadas serem utilizadas para explicar alguns conceitos relacionados à biblioteca `DrWatson.jl`, elas não são o foco do trabalho e por isso não serão discutidas a fundo. Serão expostos, de forma qualitativa, os resultados obtidos através do uso do pacote e comentados seus impactos no decorrer da escrita do código.

6.1 Inicialização do projeto

Dentre as funções aplicadas nas simulações testadas, a primeira a ser abordada será a de inicialização de projeto. Função que é executada dentro de um `Terminal` ou de um `Prompt de Comando` antes até de qualquer outra, como já ilustrado pelas figuras 8, 9, 10, 11 e 13. Essa função roda um *script* de criação de pastas e subpastas que servirão futuramente para o armazenamento de dados de entrada e saída para as simulações a serem executadas.

A função `initialize_project()` tem como argumentos não obrigatórios, que caso o usuário não insira manualmente irá se autocompletar com as informações padrões, o `path`, o `name` e mais alguns argumentos opcionais. Além de transformar o diretório em um repositório `Git` que o usuário pode fazer o controle de versões e *coworking* pelo `GitHub`, por exemplo.

- `path` = *String* que identifica o diretório onde o projeto estará instanciado no computador (caso vazio, será usado o diretório onde o usuário executou a função de criação)
- `name` = *String* que identifica o nome do projeto

Essa funcionalidade permite ao cientista economizar tempo para montar um diretório consistente, instanciar o projeto com dependências próprias e compartilhá-lo com sua equipe de forma robusta, mantendo um padrão e evitando *bugs*. Caso queira, também é possível criar outras pastas para o projeto e remover as que não forem usadas.

A diferença entre os métodos manuais e através da biblioteca, foram sentidas ao se tentar criar o repositório de forma manual e posteriormente de forma automatizada por `DrWatson.jl`. Os esforços de criar primeiramente um diretório de projeto com dependências locais da linguagem `Julia`, posteriormente transformá-lo em repositório `Git` e depois

gerar as pastas e subpastas foi substituído por apenas duas linhas de código no `Prompt de Comando`.

6.2 Funções de diretório

Após o diretório de projeto ser criado, podemos fazer uso de outras funcionalidades disponíveis na biblioteca, entre elas estão as funções de exploração de diretórios do projeto. Essas funções permitem que o pesquisador navegue facilmente entre os diretórios do projeto tanto para salvar quanto para carregar dados.

Supondo que o nome escolhido para o projeto foi `FTHA` e que a pasta escolhida para instanciar o projeto foi `C:\Users\projetos\Documents`, obtém-se que:

```
projectdir()    = C:\Users\projetos\Documents\FTHA
datadir()       = C:\Users\projetos\Documents\FTHA\data
plotsdir()      = C:\Users\projetos\Documents\FTHA\plots
scriptsdir()    = C:\Users\projetos\Documents\FTHA\scripts
srcdir()        = C:\Users\projetos\Documents\FTHA\src
```

Inclusive, como argumentos de cada uma das funções, podem ser inseridos, em forma de *strings*, nomes de pastas internas a esses diretórios para direcionar a diretórios secundários aos já pré-estabelecidos. Além de que, é possível criar outras funções customizadas a partir dessas que são padrões.

Observou-se a grande utilidade dessas funções quando se trata da fácil navegação entre diretórios do projeto e de sua capacidade de reprodutibilidade entre máquinas com diferentes sistema operacionais, como `Windows` e `Linux`. Sendo possível a navegação em ambos os sistemas sem que se perca as referências de pastas por serem sistemas com diferentes formas de navegação.

6.3 Ferramenta de Planejamento Fatorial

Como já explicado na seção 4.3.2, é possível utilizar a ferramenta de Planejamento Fatorial para explorar todo um espaço de design. Na simulação de otimização do Ciclo Otto FTHA, no escopo de todos os parâmetros de entrada, os considerados para serem variados dentro de um intervalo definido serão apenas dois: avanço de ignição (θ) e velocidade angular do virabrequim (N). Pois como os outros serão constantes, seus valores serão iguais a 1 na equação que retorna o número total de experimentos a serem realizados, o que não altera em nada no resultado final. Logo, a multiplicação do número de possibilidades de avanço de ignição pelo de velocidade angular do virabrequim, dará a quantidade total de experimentos a

serem rodados usando o mesmo *script* para construir a análise final. Os valores utilizados na simulação e aplicados na equação foram os seguintes:

- Avanço de ignição (θ[°]): $\theta = -75 + 5n, \{\forall n \in \mathbb{N} : 0 \leqslant n \leqslant 15\}$
- Rotação do virabrequim (N[RPM]): $N = 600 + 200n, \{\forall n \in \mathbb{N} : 0 \leqslant n \leqslant 22\}$

$$Experimentos = n_\theta \cdot n_N \Rightarrow 368 = 16 \cdot 23 \tag{6.1}$$

Ao aplicar o número de valores possíveis de cada variável na Equação 6.1, é encontrada a quantidade de experimentos necessários para a mesma simulação com a finalidade de explorar o espaço de design por completo e extrair o mapa de eficiências.

Primariamente, no algoritmo da simulação, foi construído um dicionário com todas as variáveis e seus respectivos valores. Os parâmetros que serão mantidas durante todos os experimentos, têm apenas valores constantes armazenados em suas memórias, já os valores das variáveis que irão testar várias possibilidades durante os experimentos devem ser guardados em vetores, e as variáveis que devem levar um vetor em sua memória mas que não devem variar, são incluídas dentro de outro vetor.

A biblioteca `DrWatson.jl` também disponibiliza *features* para facilitar essa inclusão dos parâmetros de entrada e suas respectivas variáveis dentro dos dicionários, que são as macros `@dict` e `@strdict`. A diferença de uma para a outra é que na macro `@dict` o dicionário é criado com símbolos e na `@strdict` é criado com `strings`, podendo os dicionários criados serem convertidos de um para o outro através das função `tosymboldict()` e `tostringdict()`.

Para o armazenamento de tipo *container*, como são os dicionários e as tuplas nomeadas, também pode ser usada a macro @ntuple, porém essa aplicação não foi feita nesse estágio do código. E assim como dicionários podem ser convertidos entre si, também é possível a conversão entre tuplas nomeadas para dicionários e vice-versa através das funções `ntuple2dict()` e `dict2ntuple()`.

Além disso, é possível utilizar a macro `@onlyif` para inserir valores dentro de um *container* dependendo da condição inserida como argumento. Se a condição for satisfeita, a macro retorna um valor, senão retorna outro valor ou conjunto de valores, como um vetor.

Após os dicionários terem sido criados, a função `dict_list()` é utilizada para executar o método do Planejamento Fatorial, distribuindo cada variação das entradas entre os casos a serem executados nos experimentos e, também, a função `dict_list_count()` para contar quantos experimentos serão executados. É nessa etapa que ao inserir um dicionário construído com valores constantes e vetores de variação, obtém-se como saída um vetor com vários dicionários (cada um com apenas valores constantes responsáveis pelas entradas de cada caso a ser testado). Esse vetor retornado pela função `dict_list()` será usado por um loop que é

responsável pela repetição da simulação para extrair e aplicar os valores de cada dicionário nos cálculos.

Para exemplificar esse processo, a Figura 18 mostra o processo de declaração das variáveis e seus respectivos valores, a inserção dessas variáveis dentro de um dicionário tal como descrito e a aplicação do Planejamento Fatorial para distribuir todas as possibilidades de entradas para os experimentos.

Figura 18 – Exemplo de Planejamento Fatorial

```
In [1]:
using DrWatson

a = [[-10, 10]]
b = [20, 30, 40]
c = ["oi", "tchau"]

parametros = @dict a b c

merge!(parametros, @dict d = [0, @onlyif(:b == 30, 1)])
```

```
Out[1]: Dict{Symbol, Vector{T} where T} with 4 entries:
  :a => [[-10, 10]]
  :b => [20, 30, 40]
  :d => Any[0, DependentParameter{Int64}(1, #1)]
  :c => ["oi", "tchau"]
```

```
In [2]:
tparametros = dict_list(parametros)
```

```
Out[2]: 8-element Vector{Dict{Symbol, Any}}:
 Dict(:a => [-10, 10], :b => 20, :d => 0, :c => "oi")
 Dict(:a => [-10, 10], :b => 30, :d => 0, :c => "oi")
 Dict(:a => [-10, 10], :b => 40, :d => 0, :c => "oi")
 Dict(:a => [-10, 10], :b => 30, :d => 1, :c => "oi")
 Dict(:a => [-10, 10], :b => 20, :d => 0, :c => "tchau")
 Dict(:a => [-10, 10], :b => 30, :d => 0, :c => "tchau")
 Dict(:a => [-10, 10], :b => 40, :d => 0, :c => "tchau")
 Dict(:a => [-10, 10], :b => 30, :d => 1, :c => "tchau")
```

Fonte: Autoria Própria (2022).

6.4 Função de extração de variáveis

A macro `@unpack`, por mais que não seja nativa da biblioteca `DrWatson.jl`, é uma dependência muito utilizada nas simulações. Seu papel é desencapsular as variáveis de dentro do *container* para o ambiente inserido com o objetivo de serem usadas nos cálculos da simulação. A Figura 19 ilustra como funciona o processo de extração das variáveis de dentro dos *containers*. Pode-se perceber que como as variáveis foram extraídas para dentro de um loop, quando chamadas em ambiente global não foi encontrado nada correspondente na memória e um erro foi retornado, indicando que existe uma separação entre o ambiente global do *script* e o ambiente onde se deseja extrair os valores, algo que é desejável a fim de evitar conflitos.

Figura 19 – Exemplo de desencapsulamento de variáveis do container

```
In [1]:
using DrWatson

entradas = Dict(
    :a => [1, 2, 3],
    :b => [4, 5, 6],
    :c => [7, 8, 9]
)
```

```
Out[1]: Dict{Symbol, Vector{Int64}} with 3 entries:
  :a => [1, 2, 3]
  :b => [4, 5, 6]
  :c => [7, 8, 9]
```

```
In [2]:
t_entradas = dict_list(entradas)

respostas = []
for i in t_entradas
    @unpack a, b, c = i
    append!(respostas, a*b*c)
end

println(respostas)
```

```
Any[28, 56, 84, 35, 70, 105, 42, 84, 126, 32, 64, 96, 40, 80, 120, 48, 96, 144, 36, 72, 108, 45, 9
0, 135, 54, 108, 162]
```

```
In [3]:
a
```

```
UndefVarError: a not defined

Stacktrace:
 [1] top-level scope
   @ :0
 [2] eval
   @ .\boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base .\loading.jl:1094
```

Fonte: Autoria Própria (2022).

6.5 Função para carregar ou rodar simulação

Visando poupar esforços computacionais e a obtenção das respostas de maneira rápida, a função `produce_or_load()` ou a macro `@produce_or_Load` podem ser usadas para fazer uma varredura no diretório configurado para o salvamento das simulações com base nas variáveis e seus parâmetros de entrada, tentando encontrar os resultados de possíveis casos idênticos já rodados. Caso não encontre, seu papel é rodar e salvar os resultados com informações relacionadas ao repositório `Git` inclusas.

Como é perceptível pela Figura 20, na simulação de otimização do Ciclo Otto FTHA, essa função foi usada para conferir os resultados de todas as simulações já feitas, obter os resultados já disponíveis e executar os que não foram encontrados.

6.6 Funções de salvamento

Muitas vezes o usuário quer salvar suas simulações em um diretório de dados e gostaria que no nome desses documentos estivessem inclusas as variáveis usadas e os valo-

Figura 20 – Screenshot do loop de execução da simulação do Ciclo Otto FTHA

```
#-------------------------------------------------------------------#
#                            Simulação                              #
#-------------------------------------------------------------------#

# Criação do ambiente de simulação
dados = @dict gas rv Vd LR z θ Δt_c N T₀ P₀ q_ent Ns Nq Re αLim ϵ_v ϵ_w hoje agora
list_of_results = []

# Loop de execução do planejamento fatorial
function simular()
    for (i, entradas) in enumerate(dict_list(dados))
        config = merge!(entradas, @dict i n_sims = dict_list_count(dados) vv = takevector(dados))
        data, file = produce_or_load(solver, simsdir(diretorio), config; accesses=takevector(dados))
        append!(list_of_results, [data])
    end
end

#-------------------------------------------------------------------#
#                            Execução                               #
#-------------------------------------------------------------------#

simular()
```

Fonte: Autoria Própria (2022).

res de entrada. Para isso a biblioteca `DrWatson.jl` oferece uma funcionalidade chamada `savename()` que permite que sejam salvas simulações utilizando o dicionário de variáveis de entrada no intuito de criar nomes personalizados.

Na simulação de otimização do Ciclo Otto FTHA, antes da implementação da função `produce_or_load()` que substitui todo esse processo de salvamento, a função `savename()` combinada com `datadir()` e `wsave()` ou `@tagsave` era usada para tal finalidade. Essas funcionalidades disponibilizam ao usuário fácil acesso ao diretório de salvamento e fácil nomeação dos arquivos de dados.

Todo esse sistema funciona da seguinte forma: a função `savename()` recebe como argumentos o dicionário de entradas e a extensão do arquivo de dados como `string`. Também poderão ser utilizados alguns argumentos chave que irão editar a forma de escrita e/ou filtrar algumas variáveis a serem utilizadas nessa nomeação de acordo com as preferências do usuário. Além disso, é possível definir, através da função `default_prefix()` um prefixo padrão customizado que será utilizado para nomear todos testes executados.

Em seguida, a função `savename()`, com seus devidos argumentos definidos, é posta dentro da função `datadir()` em conjunto com os nomes das pastas do diretório escolhido e todos esse conjunto é inserido como argumentos da função `wsave()` ou da macro `@tagsave`, juntamente com os resultados da simulação para serem enviados ao banco de dados.

A partir da visualização da Figura 21, é possível perceber um exemplo do processo inicial de definição das variáveis, do processo intermediário de criação da simulação e o processo final

Figura 21 – Exemplo de simulação com salvamento por substituição

In [1]:
```julia
using DrWatson

safe = false

ent = Dict(
    :a => [1, 2],
    :b => [3, 4]
)


function sim(params)
    @unpack a, b = params
    return @strdict ans = a*b
end

function run(all)
    params = dict_list(all)

    for i in params
        if safe
            safesave(datadir("sims", "sim_test", savename(i, "jld2")), sim(i))
        else
            wsave(datadir("sims", "sim_test", savename(i, "jld2")), sim(i))
        end
    end
end

run(ent)
```

In [2]:
```julia
readdir(datadir("sims", "sim_test"))
```

Out[2]:
```
4-element Vector{String}:
 "a=1_b=3.jld2"
 "a=1_b=4.jld2"
 "a=2_b=3.jld2"
 "a=2_b=4.jld2"
```

Fonte: Autoria Própria (2022).

de execução e salvamento dos resultados na pasta de dados com os nomes personalizados de acordo com cada experimento criado pelo Planejamento Fatorial. É possível notar ainda que ao alterar o valor da variável `safe` é possível salvar com substituição ou com renomeação. Quando usada, a função `safesave()` evita que o usuário acidentalmente sobrescreva simulações já rodadas anteriormente e que não deveriam ser apagadas. Vide Figura 22.

Como uma espécie de complemento das funções de diretório e das funções da salvamento, também existem funções para ler os diretório e analisar se ainda falta algo a ser submetido pelo `Git`, como a função `readdir()` e `isdirty()`. A função `readdir()` lê o diretório introduzido como argumento e retorna os arquivos contidos, já a função `isdirty()` lê o repositório e retorna, na forma de um valor lógico (verdadeiro ou falso), se ainda existem alteração a serem submetidas.

Ao executar a simulação de otimização do Ciclo Otto FTHA, muitas vezes era exibida no terminal a mensagem de que as alterações do repositório precisavam ser submetidas, porém não era algo desejado de ser visualizado junto aos resultados, já que essa informação de repositório "sujo", naquele momento, era sabida. Então através do uma operação lógica e da função `isdirty()`, o aviso foi desligado.

Figura 22 – Exemplo de simulação com salvamento por renomeação

In [3]:
```
using DrWatson

safe = true

ent = Dict(
    :a => [1, 2],
    :b => [3, 4]
)


function sim(params)
    @unpack a, b = params
    return @strdict ans = a*b
end

function run(all)
    params = dict_list(all)

    for i in params
        if safe
            safesave(datadir("sims", "sim_test", savename(i, "jld2")), sim(i))
        else
            wsave(datadir("sims", "sim_test", savename(i, "jld2")), sim(i))
        end
    end
end

run(ent)
```

In [4]:
```
readdir(datadir("sims", "sim_test"))
```

Out[4]:
```
8-element Vector{String}:
 "a=1_b=3.jld2"
 "a=1_b=3_#1.jld2"
 "a=1_b=4.jld2"
 "a=1_b=4_#1.jld2"
 "a=2_b=3.jld2"
 "a=2_b=3_#1.jld2"
 "a=2_b=4.jld2"
 "a=2_b=4_#1.jld2"
```

Fonte: Autoria Própria (2022).

6.7 Função de coleta de resultados

Após as simulações serem rodadas, a última etapa é coletar os resultados obtidos a fim de poder realizar as análises necessárias. Para isso existem duas funções de `DrWatson.jl`: `collect_results()` e `wload()`. A diferença entre elas, como pode ser visto na Figura 23, é que a primeira traz todos os resultados contidos no diretório e segunda traz apenas os resultados contidos em um dos arquivos do diretório.

A função `collect_results()`, que necessita ser usada juntamente com a biblioteca `DataFrames.jl`, retorna, em forma de tabela, os resultados de todos os testes simulados e armazenados dentro da pasta da simulação correspondente. Cabe ao usuário, a partir desse ponto, filtrar os resultados que interessam às suas análises. Além disso, se for a primeira vez que a função está sendo executada, ela irá salvar, no diretório da simulação, um arquivo com todos os resultados dos testes agrupados, para que das outras vezes não seja necessário carregar um por um, apenas esse compilado será utilizado. Caso seja necessário, ela pode apenas incluir novos resultados e carregar os anteriores.

Figura 23 – Exemplo de carregamento de resultados para análise

```
In [1]:
1 using DataFrames
2
3 df = collect_results(datadir("sims", "sim_test"))
```

Out[1]: 8 rows × 2 columns

	ans	path
	Int64?	String?
1	3	C:\\Users\\Alessandro\\Dropbox\\TCC-PortugalAC\\01-codes\\01-FTHA\\FTHA\\data\\sims\\sim_test\\a=1_b=3.jld2
2	3	C:\\Users\\Alessandro\\Dropbox\\TCC-PortugalAC\\01-codes\\01-FTHA\\FTHA\\data\\sims\\sim_test\\a=1_b=3_#1.jld2
3	4	C:\\Users\\Alessandro\\Dropbox\\TCC-PortugalAC\\01-codes\\01-FTHA\\FTHA\\data\\sims\\sim_test\\a=1_b=4.jld2
4	4	C:\\Users\\Alessandro\\Dropbox\\TCC-PortugalAC\\01-codes\\01-FTHA\\FTHA\\data\\sims\\sim_test\\a=1_b=4_#1.jld2
5	6	C:\\Users\\Alessandro\\Dropbox\\TCC-PortugalAC\\01-codes\\01-FTHA\\FTHA\\data\\sims\\sim_test\\a=2_b=3.jld2
6	6	C:\\Users\\Alessandro\\Dropbox\\TCC-PortugalAC\\01-codes\\01-FTHA\\FTHA\\data\\sims\\sim_test\\a=2_b=3_#1.jld2
7	8	C:\\Users\\Alessandro\\Dropbox\\TCC-PortugalAC\\01-codes\\01-FTHA\\FTHA\\data\\sims\\sim_test\\a=2_b=4.jld2
8	8	C:\\Users\\Alessandro\\Dropbox\\TCC-PortugalAC\\01-codes\\01-FTHA\\FTHA\\data\\sims\\sim_test\\a=2_b=4_#1.jld2

```
In [2]:
1 wload(datadir("sims", "sim_test", readdir(datadir("sims", "sim_test"))[1]))
```

```
Out[2]: Dict{String, Any} with 1 entry:
  "ans" => 3
```

Fonte: Autoria Própria (2022).

A função `wload()` carrega os resultados de apenas um dos testes rodados pela simulação. O que o usuário deve inserir com argumento em seus parenteses é o diretório onde se encontram os resultados da simulação e o nome do teste desejado. A função irá carregar os dados e retornar em formato de dicionário, que pode ser armazenado dentro de uma variável.

Um ponto muito interessante observado é que se o usuário não quiser usar as função de salvamento e carregamento posteriormente, ele pode evitar isso por meio da função `produce_or_load()` ou da macro `@produce_or_load`, que aceita em seus argumentos as entradas da simulação e a função utilizada para executar o loop dos testes. E como argumentos chaves, aceita os mesmos usados para as funções de salvamento, que serão usados para o mesmo fim.

6.8 Reprodutibilidade

Além das funções que a biblioteca proporciona ao cientista, também foi testada a reprodutibilidade do código em outras máquinas. Para isso, após instanciar o projeto em uma máquina virtual com `Linux Ubuntu` instalado, as simulações foram executadas. Para esse processo, que normalmente é um pouco desgastante ao usuário por ter que preparar sua máquina tal como a máquina do autor, percebeu-se que pouco trabalho foi feito. Apenas foi necessário instalar a biblioteca `DrWatson.jl` no ambiente global da linguagem `Julia`, ativar o projeto e instanciá-lo. A partir daí, estava tudo preparado para rodar as simulações tal como feito na outra máquina.

7 CONCLUSÃO

É possível observar pelos resultados apresentados, que a biblioteca possui boas ferramentas para resolução de problemas relacionados ao compartilhamento de projetos, salvamento e carregamento dados. Ao serem usadas na escrita de simulações, percebeu-se ótimas contribuições por parte do pacote que propiciaram o aumento de produtividade e o ganho de tempo.

Acredita-se que pela atipicidade de bibliotecas com o propósito de auxiliar o pesquisador da forma como `DrWatson.jl` faz, esse trabalho gere relevância por estudar, explorar e compartilhar as experiências obtidas ao aplicar a ferramenta, algo pouco explorado por trabalhos. E, por isso, os objetivos iniciais do trabalho de aplicar à biblioteca a simulação e investigar suas consequências foram considerados alcançados. Por mais que as simulações em si não sejam o foco principal do trabalho, sendo pouco aprofundadas durante os capítulos, o necessário para elucidar seus conceitos e a aplicação de `DrWatson.jl` foi assegurado.

Assim, sabendo-se de todas as dificuldades enfrentadas pelo cientista na hora de escrever e executar suas simulações, dificuldades essas que quase nunca são relatadas em trabalhos científicos, muitos pesquisadores passam por problemas de salvamento, carregamento de dados e compartilhamento de projetos, e poucos expõem essas dificuldades no trabalho. Então pode-se dizer que `DrWatson.jl` cumpriu seu papel no quesito investigado de diminuir os esforços e melhorar a robustez do projeto.

REFERÊNCIAS

BATHE, K.-J. Finite element method. *In*: **Wiley Encyclopedia of Computer Science and Engineering**. [*S.l.*: *s.n.*], 2007. ISBN 9781439852347.

BEZANSON, J. *et al.* Julia: A Fast Dynamic Language for Technical Computing. **Computing Research Repository**, 2012. Disponível em: http://arxiv.org/abs/1209.5145.

BOUDREAU, E. **C For Data Science?. Within the trade of Data-Science, there...**. 2019. Disponível em: https://towardsdatascience.com/c-for-data-science-8321d6509484. Acesso em: 24 de nov. de 2021.

BRUNETTI, F. **Motores de Combustão Interna - Vol. 1**. 2. ed. São Paulo, SP: Editora Blucher, 2018. ISBN 9788521218135. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788521212942/. Acesso em: 17 de nov. de 2021.

ÇENGEL, Y.; BOLES, M. **Termodinâmica**. 7. ed. Porto Alegre, RG: AMGH Editora, 2013. ISBN 9786071507433. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788580552010/. Acesso em: 17 de nov. de 2021.

DATSERIS, G. *et al.* DrWatson: the perfect sidekick for your scientific inquiries. **Journal of Open Source Software**, The Open Journal, v. 5, n. 54, p. 2673, 2020. ISSN 2475-9066. Disponível em: https://doi.org/10.21105/joss.02673.

DATSERIS, G. *et al.* **DrWatson Workflow Tutorial · DrWatson**. 2020. Disponível em: https://juliadynamics.github.io/DrWatson.jl/dev/workflow/. Acesso em: 4 de nov. de 2021.

DEVATHON. **Julia vs Python in 2020. Despite such big numbers, the situation... | by Devathon | Medium**. 2020. Disponível em: https://medium.com/@devathon_/julia-vs-python-in-2020-d2dc2c2ef3f. Acesso em: 3 de nov. de 2021.

Driven Data. **Cookiecutter Data Science**. 2018. Disponível em: https://drivendata.github.io/cookiecutter-data-science/#cookiecutter-data-sciencehttps://drivendata.github.io/cookiecutter-data-science/. Acesso em: 25 de nov. de 2021.

DYCK, D. N.; LOWTHER, D. A.; MCFEE, S. Determining an approximate finite element mesh density using neural network techniques. **IEEE Transactions on Magnetics**, v. 28, n. 2, p. 1767–1770, 1992. ISSN 19410069.

GIBI, R. **Motores a Pistão | abekwar**. 2013. Disponível em: https://abekwar.wordpress.com/2013/04/09/motores-a-pistao/. Acesso em: 17 de nov. de 2021.

GOMES, M. d. N. *et al.* Análise de malhas para geração numérica de ondas em tanques. **VII Congresso Nacional de Engenharia Mecânica - CONEM 2012**, n. 2009, 2012.

GOUY, I. **Julia vs Python 3 - Which programs are fastest? | Computer Language Benchmarks Game**. 2021. Disponível em: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/julia-python3.html. Acesso em: 3 de nov. de 2021.

Julia Computing. **Newsletter August 2021 - Julia Computing Completes $24 Million Series A Fundraise and Former Snowflake CEO Bob Muglia Joins Julia Computing Board of Directors - Julia Computing**. 2021. Disponível em: https://juliacomputing.com/blog/2021/08/newsletter-august/. Acesso em: 3 de nov. de 2021.

JUNIOR, L. C. M. **MÁQUINAS TÉRMICAS I - MOTORES DE COMBUSTÃO INTERNA**. [*S.l.*]: Unijuí, 2003.

MARINHO, M.; CASTRO, W. Planejamento Fatorial: Uma Ferramenta Poderosa Para Os Pesquisadores. **Congresso Brasileiro de Ensino de Engenharia**, p. 1–9, 2005.

NAAKTGEBOREN, C. An air-standard finite-time heat addition Otto engine model. **International Journal of Mechanical Engineering Education**, v. 45, n. 2, p. 103–119, 2017. ISSN 20504586.

PEREIRA, P. **Que futuro para os motores a 2 tempos nas motos? - Opiniões - Andar de Moto Brasil**. 2019. Disponível em: https://www.andardemoto.com.br/opinioes/44004-que-futuro-para-os-motores-a-2-tempos-nas-motos/. Acesso em: 17 de nov. de 2021.

PERKEL, J. M. **Programming: Pick up Python**. Nature Publishing Group, 2015. 125–126 p. Disponível em: https://www.nature.com/articles/518125a. Acesso em: 25 de nov. de 2021.

ROSSUM, G. V.; DRAKE, F. L. **Python Reference Manual**. Amsterdam, The Netherlands, 1995. v. 22, 9117–9129 p. Disponível em: http://www.python.org/doc/ref/.

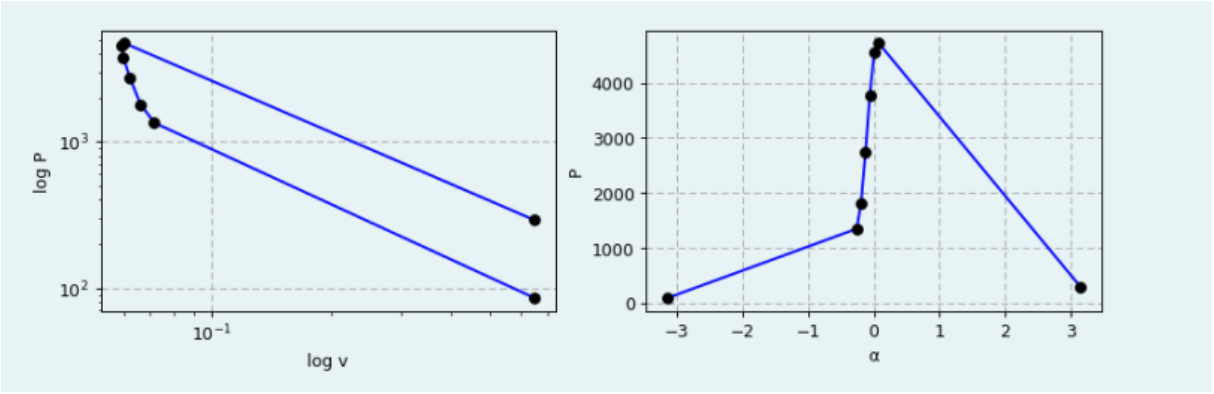
ROVEDA, U. **O que é Python, para que serve e por que aprender?** 2020. Disponível em: https://kenzie.com.br/blog/o-que-e-python/. Acesso em: 3 de nov. de 2021.

SOUZA, W. **Saiba mais como o Python surgiu**. 2021. Disponível em: https://digitalinnovation.one/artigos/saiba-mais-como-o-python-surgiu-AHGI60. Acesso em: 3 de nov. de 2021.

VENNERS, B. **The Making of Python**. 2003. Disponível em: https://www.artima.com/articles/the-making-of-python. Acesso em: 3 de nov. de 2021.

ANEXO A – Gabarito de validação

Considere um motor quadrado de razão volumétrica 11:1 e cilindrada de 4100cm³, com razão de comprimentos biela/manivela igual a 4.2, 6 cilindros, avanço de ignição de -15° em relação ao PMS e com tempo de combustão completa da mistura ar-combustível de 4000μs em uma condição de operação a 800RPM. Utilizando a formulação de tempo finito de combustão com hipóteses do padrão a ar desenvolvida em sala, modele o ciclo termodinâmico deste motor utilizando o fluido CO_2 (M = 44.0098 kg/kmol, R = 0.188923 kJ/kg·K) como um gás ideal com calores específicos variáveis $\bar{c}_p(T) = \sum_i \bar{a}_i T^i$, com $\bar{a}_0 = +23.965kJ/kmol \cdot K^1, \bar{a}_1 = +0.054685kJ/kmol \cdot K^2, \bar{a}_2 = -3.0227e{-}05kJ/kmol \cdot K^3, \bar{a}_3 = +6.0233e{-}09kJ/kmol \cdot K^4$, de sorte que $u(T) = \sum_i b_i T^{i+1}/(i+1), b_0 = (\bar{a}_0 - R_u)/M, b_{i>0} = \bar{a}_{i>0}/M$, desde o ângulo do eixo de manivelas $\alpha = -\pi$ até o ângulo $\alpha = +\pi$. Faça 1 passo para processos isentrópicos e 5 passos para o processo de adição de calor. Faça 5 iterações para a convergência do expoente politrópico utilizando como limiar para processos isocóricos a variação específica de volume de 1.0e-8 m³/kg e 5 iterações para a convergência da temperatura pelo algoritmo de Newton-Raphson, com chute inicial de $T^0 = u/b_0$. A condição de admissão é de 20.0°C e 85.0kPa, para $\alpha = -\pi$, e a adição específica total de calor é de 1000.0kJ/kg. Determine:



A massa do sistema, em mg, é de 1153.6

A duração angular da combustão, δ, em °, é de 19.2

A razão de consumo de trabalho do ciclo, em %, é de 34.081

A pressão média efetiva do ciclo, em kPa, é de 623.89

A eficiência térmica, em %, é de 36.955

Quando α = -15°, a pressão, em MPa, é de 1.3461

Quando α = -15°, a temperatura, em °C, é de 237.63

O coeficiente politrópico do processo que termina com α = -15° é de 1.2516

Quando α = -15°, o trabalho específico acumulado pelo sistema, em kJ/kg, é de 163.43

Quando α = -11.16°, a pressão, em MPa, é de 1.802

Quando α = -11.16°, a temperatura, em °C, é de 357.86

O coeficiente politrópico do processo que termina com α = -11.16° é de 3.6312

Quando α = -11.16°, o trabalho específico acumulado pelo sistema, em kJ/kg, é de 172.06

Quando α = -7.32°, a pressão, em MPa, é de 2.7316

Quando α = -7.32°, a temperatura, em °C, é de 626.47
O coeficiente politrópico do processo que termina com α = -7.32° é de 6.7823
Quando α = -7.32°, o trabalho específico acumulado pelo sistema, em kJ/kg, é de 180.84
Quando α = -3.48°, a pressão, em MPa, é de 3.779
Quando α = -3.48°, a temperatura, em °C, é de 925.21
O coeficiente politrópico do processo que termina com α = -3.48° é de 8.5782
Quando α = -3.48°, o trabalho específico acumulado pelo sistema, em kJ/kg, é de 188.29
Quando α = 0.36°, a pressão, em MPa, é de 4.5466
Quando α = 0.36°, a temperatura, em °C, é de 1152.5
O coeficiente politrópico do processo que termina com α = 0.36° é de 16.472
Quando α = 0.36°, o trabalho específico acumulado pelo sistema, em kJ/kg, é de 191.06
Quando α = 4.2°, a pressão, em MPa, é de 4.7249
Quando α = 4.2°, a temperatura, em °C, é de 1232.9
O coeficiente politrópico do processo que termina com α = 4.2° é de -2.3505
Quando α = 4.2°, o trabalho específico acumulado pelo sistema, em kJ/kg, é de 186.53
Quando α = 180°, a pressão, em MPa, é de 0.29007
Quando α = 180°, a temperatura, em °C, é de 727.24
O coeficiente politrópico do processo que termina com α = 180° é de 1.1718
Quando α = 180°, o trabalho específico acumulado pelo sistema, em kJ/kg, é de -369.55