INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE TECNOLOGIA EM ELETRÔNICA INDUSTRIAL

LUCAS GABRIEL COLIADO BANDEIRA

FLUXO DE DISTRIBUIÇÃO E DOCUMENTAÇÃO DA BIBLIOTECA PYTHON MAGPYLIB

FLORIANÓPOLIS, 2019

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE TECNOLOGIA EM ELETRÔNICA INDUSTRIAL

LUCAS GABRIEL COLIADO BANDEIRA

FLUXO DE DISTRIBUIÇÃO E DOCUMENTAÇÃO DA BIBLIOTECA PYTHON MAGPYLIB

Trabalho de conclusão de curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Tecnólogo em Eletrônica Industrial

Orientador:
Prof. Samir Bonho

FLORIANÓPOLIS, 2019

Bandeira, Lucas Gabriel Coliado
Fluxo de distribuição e documentação da biblioteca Python magpylib /
Lucas Gabriel Coliado Bandeira; orientador, Prof. Samir Bonho – Florianópolis, SC, 2019.
66 p. : il. color.

Trabalho de Conclusão de Curso (Eng. Eletrônica) – Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina. Inclui referências.

1. Desenvolvimento de Software. 2. Magnetismo. 3. DevOps.
I. Bonho, Samir. II. Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina. III. Fluxo de distribuição e documentação da biblioteca Python magpylib

FLUXO DE DISTRIBUIÇÃO E DOCUMENTAÇÃO DA BIBLIOTECA PYTHON MAGPYLIB

LUCAS GABRIEL COLIADO BANDEIRA

Este Trabalho foi julgado adequado para obtenção do Título de Tecnólogo em Eletrônica Industrial em Dezembro de 2019 e aprovado na sua forma final pela banca examinadora do Curso Superior de Tecnologia em Eletrônica Industrial do Instituto Federal de Educação Ciência, e Tecnologia de Santa Catarina.

Florianópolis, 08 de Dezembro de 2019.

Banca Examinadora:

Samir Bonho, M. Sc.

Alexandre Leizor Szczupak, Dr.

Renan Augusto Starke, Dr.

AGRADECIMENTOS

Gostaria de agradecer a todo o corpo docente do Departamento de Eletrônica IFSC do campus Florianópolis, por ter providenciado a minha formação profissional e a garra necessária para a concepção deste trabalho.

Ao meu Orientador Samir Bonho, que se dedicou em me atender com atenção e cuidado durante a concepção deste trabalho.

Ao Professor Fernando Luiz Rosa Mussoi, que demonstrou interesse e atenção ao andamento de todos os trabalhos que realizei com ele na medida do possível.

Agradeço a todo o corpo de técnicos e administradores do Departamento de Eletrônica IFSC do campus Florianópolis, por manterem um padrão de excelência na credibilidade da educação e dos laboratórios e equipamentos da instituição do campus.

Ao Marcelo Ribeiro e a empresa *Silicon Austria Labs*, que colaboraram com o Departamento de Eletrônica para me fornecer a excelente oportunidade de demonstrar a qualidade dos discentes da Instituição para o mundo afora.

Ao Dr. Michael Ortner, que me ofereceu respeito de todos os calibres, conhecimento, bom-humor e supervisão profissional exemplar durante o desenvolvimento deste trabalho durante minha estadia em Villach, Austria.

Ao Tarcis Aurélio Becher, por ter sido um incrível parceiro e colega de trabalho na aventura profissional e pessoal que experienciamos juntos.

Ao Leonardo Persike Martins, que me auxiliou na viagem e durante o processo de chegada na Austria.

Agradeço a minha família, por ter me dado suporte quando realmente necessário através dos anos.

Ao meu Tio Jorge Bandeira, pelo investimento emocional e organizacional que me providenciou desde muito tempo, inclusive durante a execução deste trabalho.

“To fight the abyss, one must know it.”

(BOURASSA, C.; Darkest Dungeon, 2016)

RESUMO

A utilização de ferramentas, particularmente *software*, na área de ciência e tecnologia é algo comum e utilizado para economizar tempo de profissionais ao fornecer auxilio durante certas práticas. Para isso, é imprescindível que um *software* de uso industrial possua um certo grau de confiabilidade, usabilidade e qualidade, caso contrário se tornará um impedimento àquele que for usufruir do mesmo. Neste trabalho, foi analisada a ferramenta desenvolvida pelo pesquisador Doutor Michael Ortner, *magpylib*, para cálculo de valores de campo eletromagnéticos e auxilio em projetos de microeletrônica. A ferramenta, apesar de funcional, inicialmente sofria em aspectos organizacionais e de desempenho. Após a análise, foram desenvolvidas funcionalidades e sistemas automatizados para validação e entrega da ferramenta, além de geração automática de documentação. Resultados foram demonstrados, e ao final foram divulgadas algumas das respostas da comunidade de *software* ao lançamento.

Palavras-chaves: Desenvolvimento de Software. Magnetismo. *Python*. DevOps

ABSTRACT

The usage of tools, particularly software, in the fields of science and technology is something common and used to spare working time of professionals by providing them aid during certain practices. Due to that, it is vital that industrial grade software possesses a certain level of trustworthiness, usability, and quality, otherwise it may turn into a hinderance to its user. This work analyzed a tool developed by the researcher Dr. Michael Ortner, magpylib, which calculates the values of magnetic fields for aiding in microelectronic system design. The tool, albeit functional, initially lacked in organizational and performance aspects. After the analysis, functionalities and automated systems for validations and delivery of the software were developed, alongside automatic documentation generation. Results have been demonstrated, along some of the responses by the software community.

Key-Words: Software Development. Magnetism. *Python*. DevOps

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

VCS	*Version Control System* - Sistema de Controle de Versão
CPU	*Central Processing Unit* - Unidade de Processamento Central
CI	*Continuous Integration* - Integração Contínua
CD	*Continuous Delivery* - Entrega Contínua
GCC	*GNU Compiler Collection* - Coleção de Compiladores GNU
AGPLv3	*GNU Affero General Public License* - Licença Geral Pública GNU Affero
docstring	*Documentation String* - String de Documentação, usado para comentar funções e classes em *Python*
IDE	*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado
Superclasse	- Classe que tem seus atributos herdados por outra classe

LISTA DE SÍMBOLOS

T	Tesla - Unidade de Densidade de Fluxo Magnético
A	Ampere - Unidade de Corrente Elétrica
VA	Volt-Ampere - Unidade de potência elétrica
H	Henry - Unidade de Indutância Elétrica
$\frac{H}{m}$	Henry por metro
$\frac{A}{m}$	Ampere por Metro

SUMÁRIO

1 INTRODUÇÃO

Projetos de sistemas que usufruem de sensoriamento magnético estão se tornando mais rotineiros e cobiçados pela indústria, uma vez que estes sistemas eliminam toda uma camada de complexidade dada por conexões físicas. Projetistas de microeletrônica, atualmente, possuem apenas simuladores de campos magnéticos comerciais de alto custo computacional, utilizando o Método dos Elementos Finitos para calcular campos magnéticos em áreas. Entretanto, para sensoriamento, é desejado apenas alguns pontos, algo facilmente atingível por cálculos analíticos encontrados em literatura.

Inicialmente concebido pelo Doutor Michael Ortner, cientista sênior da empresa de pesquisa e desenvolvimento *Silicon Austria Labs*, *magpylib* é uma ferramenta que soluciona problemas de custo computacional encontrados pelo pesquisador quanto aos seus projetos de microeletrônica envolvendo simulações de campos de ímãs permanentes. O protótipo desta ferramenta já foi utilizado para a fundamentação de trabalhos envolvendo sistemas de detecção de posição e orientação que utilizavam ímãs permanentes e sensoriamento de campo magnético, como o de SILVA (2018) e MARTINS (2019).

Este trabalho descreve o processo de gerenciamento de projeto utilizado na preparação, validação e lançamento oficial do *software* de simulação de campos magnéticos de ímãs permanentes, *magpylib*. O propósito desta ferramenta é auxiliar no projeto de sistemas reais envolvendo microeletrônica, e também para a didática de eletromagnetismo.

Parte do trabalho, como as *superclasses* de orientação física e os modelos matemáticos analíticos de ímãs simples, foram reaproveitados do protótipo da ferramenta. A evolução do trabalho se deu majoritariamente em adicionar novos recursos, gerir a equipe de desenvolvimento, implementar sistemas para controle de qualidade do *software*, automatizar processos de documentação, conduzir *feedback*, comunicar uma licença apropriada aos *stakeholders* do projeto, e também lançar a ferramenta para *download*. Isto providenciará avanço qualitativo considerável sobre o protótipo existente. Para isso, foi utilizada a linguagem de programação *Python 3.6*, com o pacote de computação científica *numpy* (Numpy Developers, 2019) e a biblioteca gráfica *matplotlib* (John Hunter, 2019).

1.1 Justificativa

Na área da ciência e tecnologia, inclusive na eletrônica, preza-se pela clareza e a confiabilidade em ferramentas cujo propósito fundamental é acelerar a atividade de um projeto. O processo de documentação, distribuição e garantia de qualidade

ditará tanto a efetividade quanto o valor de uma tecnologia ou ferramenta no mercado. Na indústria, engenheiros prezam pela qualidade e versatilidade de suas ferramentas, enquanto gerentes e chefes de empresas prezam pelo custo-benefício que se abstrai pelo tempo e investimento nessas tecnologias. A invenção de uma ferramenta com o intuito de distribuição dentro de seu nicho industrial, por então, não apenas deve prestar uma utilidade, mas seguir certos padrões para que se torne interessante tanto para engenheiros como líderes de projeto.

Até o ínicio do projeto, o *software* do Doutor Michael Ortner não se apresentava de modo exemplar; a documentação limitava-se a comentários no código fonte, a versão era atualizada manualmente, e se provava um desafio manter o ritmo de desenvolvimento de novos recursos quando não era possível garantir a qualidade das modificações com facilidade. Estas características diminuiam o valor da ferramenta na indústria.

1.1.1 Objetivo Geral

O trabalho a seguir visa o desenvolvimento de uma versão oficial do *software magpylib*.

1.1.2 Objetivos Específicos

Os objetivos específicos do trabalho podem ser compreendidos em duas categorias a seguir:

a) Organizacional

   a. Definir requisitos funcionais e não-funcionais;
   b. Implementar fluxo de trabalho em equipe com *software*;
   c. Obter feedback dos usuários;
   d. Comunicar uma licença para distribuição.

b) Desenvolvimento

   a. Acoplar controle de versão;
   b. Implementar sistemas de garantia de qualidade;
   c. Adicionar funcionalidades;
   d. Automatizar entrega de versões ;
   e. Corrigir problemas;
   f. Produzir documentação.

1.2 METODOLOGIA

O início do projeto foi indicado por uma explanação dos objetivos da ferramenta a ser entregue. Houve um processo de familiarização com certos conceitos de computação gráfica e manipulação vetorial, e do ambiente de desenvolvimento da mesma. Isso pavimentou o caminho para a coleta de requisitos adicionais para desenvolver o *software*, que já se encontrava utilizável em estado de protótipo.

A partir disto, foram estudados conceitos pertinentes a programação em *Python 3*, como ambientes virtuais de desenvolvimento, as bibliotecas de *software numpy* e *matplotlib* que já eram utilizadas como dependências da ferramenta, desenvolvimento dirigido por testes, módulos de processamento paralelo e análise de desempenho do programa.

Para possibilitar um desenvolvimento confiável e enquadrar a ferramenta ao padrão de mercado de *softwares*, houve um estudo aprofundado sobre organização de projetos de *software* em *Python 3*. Esse estudo resultou na instalação de um controle de versão (*VCS*) através do *software git* na plataforma *GitHub*, a descrição de todo o fluxo de trabalho para adição de novas versões ao *software*, a validação automática de testes de unidade, a padronização de comentários *docstring*, e a atualização e entrega automática para a web de múltiplas versões da documentação utilizando o *software Sphinx* com o serviço *ReadTheDocs*.

Estabilizado o procedimento para desenvolvimento, novos recursos e funcionalidades foram agregados a ferramenta, como o processamento paralelo de campos de múltiplos ímãs, processamento simultâneo de diversos “cenários” para simulação de um movimento, posicionamento e leitura de sensores tridimensionais de posicionamento livre, visualização de direção de vetores de magnetização de campo, marcadores visuais com texto, e a adição de um modelo para momento magnético. A parte de verificação matemática dos modelos físicos implementados foi realizada por uma equipe local utilizando fórmulas encontradas em literatura.

Com o interesse acadêmico e corporativo em mente, também foi realizado um estudo do aspecto legal de distribuição *software*, seguido pela tomada de decisão para o licenciamento *Affero GNU Public License version 3 (AGPLv3)* possibilitando a abertura do código da ferramenta. Finalmente, houve o lançamento do projeto em canais de distribuição *Anaconda Cloud* e *PyPi*, disponibilizando o download gratuito e utilização de qualquer versão do *software* para cientistas e engenheiros mundo afora através de apenas um comando.

2 FUNDAMENTAÇÃO TEÓRICA

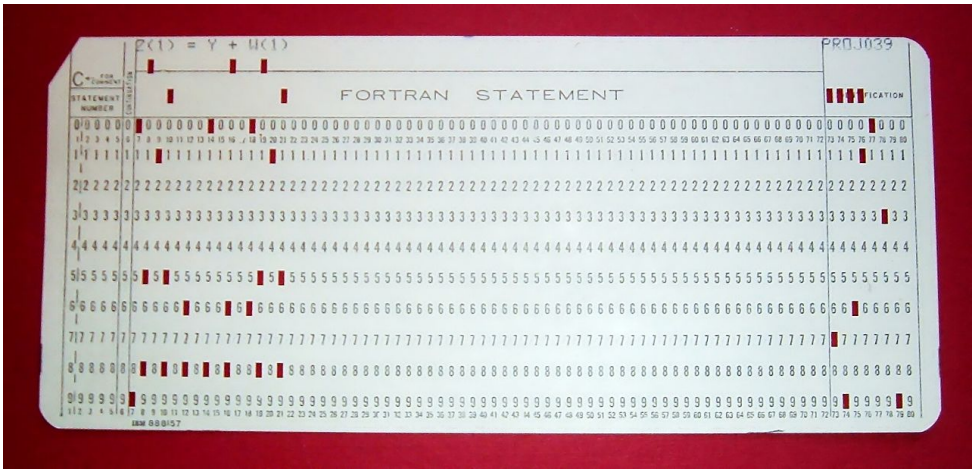
Neste capítulo comenta-se sobre alguns dos conceitos fundamentais para a compreensão este trabalho.

2.1 Sobre Programação de Computadores

Computadores são máquinas que podem ser instruídas para executar sequencias de código aritmético ou operações lógicas de modo automatizado através de programação. Unidades Centrais de Processamento (CPU) possuem uma lista de instruções definidas por um projetista que então são utilizadas pelo programador para compor seus programas.

Antigamente, estas máquinas eram inteiramente mecânicas, e evoluíram para sistemas eletrônicos. A programação destas máquinas para executarem suas tarefas computacionais eram realizadas com cartões perfurados, onde a organização de perfuração denominava a instrução a ser executada. A Figura 1 demonstra um programa para uma equação matemática em cartão perfurado.

Figura 1 – Programa em cartão perfurado



Fonte: (Wikipedia contributors, 2019)

Com o avanço de capacidade de armazenamento, programas hoje em dia são realizados e armazenados na memória do próprio computador. São armazenados números que correspondem aos níveis lógicos a serem excitados para a realização de cada instrução na CPU.

2.1.1 Código de Máquina

Código de Máquina é o nível mais próximo da realidade eletrônica de um computador. Um programa em forma de código de máquina é muitas vezes referenci-

ado como “o binário”, ou “executável”, em alusão aos números binários que compõem cada instrução.

O conceito de abstração dá-se por trabalhar com ideias ao invés de eventos. Com isto em mente, programação de computadores é altamente susceptível à abstração.

Muitas vezes, código de máquina é algo muito próximo da realidade eletrônica, porém distante da interpretação humana. Isto causa confusão durante o trabalho de programação. Para auxiliar nestes eventos, foram desenvolvidas abstrações; mnemônicos que substituem os números das instruções por símbolos e programas que traduzem estes mnemônicos para código binário.

Estes mnemônicos próximos ao código são referidos como linguagem *assembly*, onde conjuntos de letras se referem a uma operação da CPU. A listagem de um programa descrito em *assembly* é considerado código fonte. Programas que traduzem código fonte *assembly* para código binário, ou programa executável, são chamados de *Assemblers*. A figura 2 apresenta um exemplo genérico de mnemônicos assembly e suas instruções equivalentes em código de máquina.

Figura 2 – Tabela de Instruções de uma linguagem assembly e seus mnemônicos equivalentes

Binary	Mnemonic	Name	Action(s)
0000XXXXXXXXXXXX	LODD	Load Direct	AC<-M[X]
0001XXXXXXXXXXXX	STOD	Store Direct	M[X]<-AC
0010XXXXXXXXXXXX	ADDD	Add Direct	AC<-AC+M[X]
0011XXXXXXXXXXXX	SUBD	Subtract Direct	AC<-AC-M[X]
0100XXXXXXXXXXXX	JPOS	Jump on pos.	IF AC>= 0 : PC<-X
0101XXXXXXXXXXXX	JZER	Jump on zero	IF AC= 0 : PC<-X
0110XXXXXXXXXXXX	JUMP	Jump uncond.	PC <- X
0111CCCCCCCCCCCC	LOCO	Load constant	AC <- C

Fonte: (John S. Loomis, 2019)

2.1.2 Compiladores

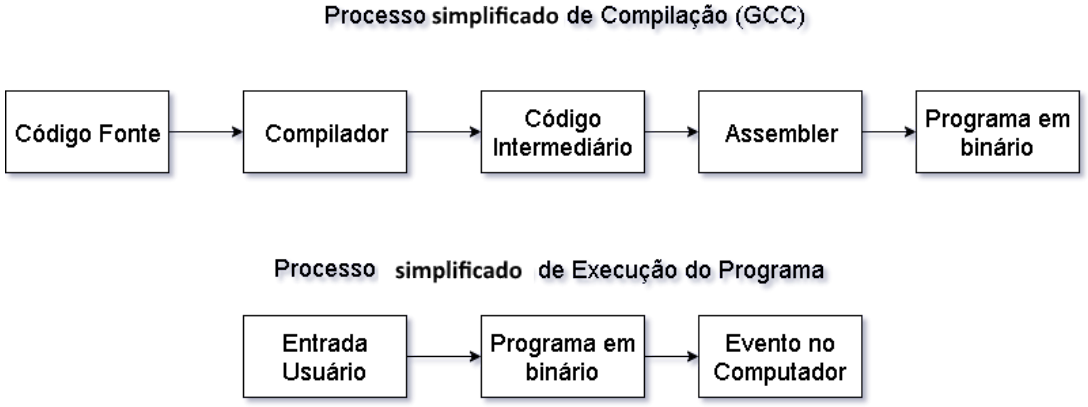
Enquanto códigos fonte listados em assembly são mais compreensíveis que códigos puramente numéricos, ainda provam-se suficientemente complicados e repetitivos quando o intuito do programador era realizar algo simples, sem preocupação excessiva com a forma exata que a sequência de operações deve ter pra se alcançar a realização.

Felizmente, a abstração de código é algo que pode ser realizado iterativamente, adicionando camadas de abstração. Isto é referido em jargão como nível “alto”, referente a abstração do evento eletrônico que ocorre na máquina. Linguagens de programação com alto nível de abstração de código podem apresentar uma única palavra

como o agrupamentos de centenas de milhares de mnemônicos, por exemplo, executando diversos eventos em um computador, necessários para imprimir um valor em uma tela.

Compiladores são programas de computador que realizam a tradução linguagens de alto nível para uma linguagem intermediária, geralmente de nível menor. Muitas vezes, esta linguagem intermediária é uma versão da linguagem *assembly* para uma arquitetura alvo de CPU. Esta linguagem intermediária, então, é traduzida para código executável pelo *Assembler*, discutido anteriormente. Este código executável é então considerado um programa independente (AHO, 2006). O conceito de entrada e saída de um compilador executando sua função é ilustrado pela Figura 3.

Figura 3 – Exemplificação simplificada do compilador Gnu Compiler Collection (GCC)



Fonte: Autor

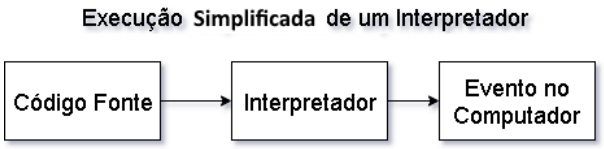
2.1.3 Interpretadores

Compiladores permitem criar código executável a partir de código fonte de linguagens de alto nível de abstração, entretanto isso requer que todo o código fonte seja processado e traduzido completamente.

Enquanto compiladores são programas que criam código executável independente, interpretadores são programas que recebem uma entrada em formato de código fonte, ou *script*. Cada linha de um *script* é então compreendido pelo interpretador, que gera as instruções uma linha de cada vez. O conceito de entrada e saída de um interpretador executando sua função é ilustrado pela Figura 4.

Isso é interessante para programas cujos passos intermediários até um evento de saída são de interesse, como em desenvolvimento matemático. No entanto, programas executados em interpretadores tendem a ser mais lentos do que quando traduzidos completamente para código de máquina. (AHO, 2006)

Figura 4 – Interpretação simplificada de código



Fonte: Autor

2.2 Paradigmas de Programação de Computadores

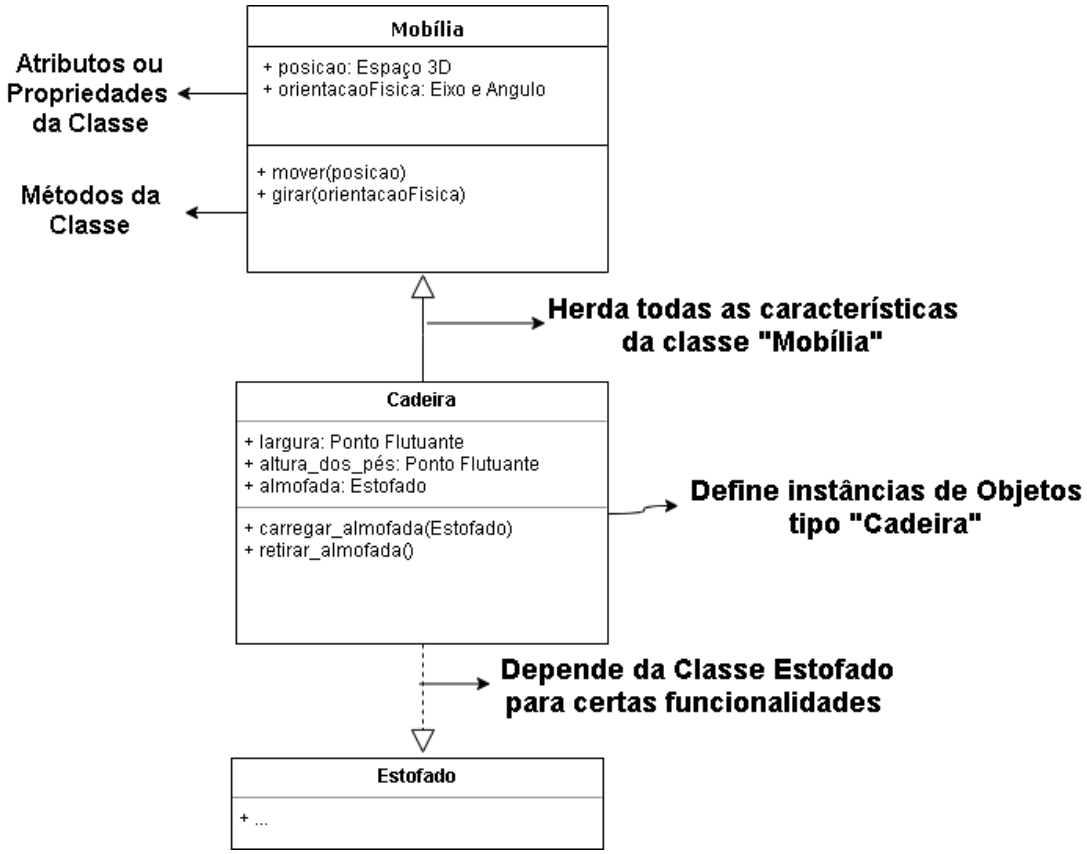
Paradigmas de programação são técnicas de abstração durante o ato de programação. Uma linguagem de programação pode aderir a um ou mais paradigmas, permitindo que o programador possa encapsular estruturas ou executar rotinas de maneiras específicas e organizadas com o intuito de facilitar a resolução de problemas. (ROY, 2004)

2.2.1 Paradigma Orientado à Objetos

Quando uma linguagem de programação diz que adere ao paradigma de Orientação à Objetos, implica-se que o programador é encorajado a utilizar de técnicas e estruturas que o permitem a trabalhar com o conceito de objeto e hierarquia de herança. Um objeto é uma instância de uma classe, que define suas propriedades únicas. Uma classe pode herdar propriedades de outras classes, onde a classe superior é chamada de *superclasse*.

A Figura 5 exemplifica um diagrama de classe usado em programação Orientada a objetos para possibilitar a criação de objetos tipo “cadeira”. Estes objetos são classificados como mobília, e dependem da classificação de “estofado”.

Figura 5 – Esquema de classe “Mobília”e “Cadeira”



Fonte: Autor

2.3 Sobre *Python*

Python é uma linguagem de programação de alto nível de abstração, utilizando o interpretador *CPython* como sua implementação padrão. *Python* é multi-paradigma, encorajando o uso de programação orientada à objetos e de outros paradigmas para que o programador tenha a liberdade de decidir a maneira mais interessante de resolver diversos problemas. Introduzida em 1991 por Guido van Rossum, *Python* possui uma gramática simples e poderosa.

2.3.1 Pacotes *Python*

Uma das características de *Python* é a possibilidade de compor novas classes, rotinas e comandos com facilidade. Conjuntos destas composições, criadas por padrão ou por usuários, são chamados de pacotes.

Pacotes de computação científica de alta performance, como *scipy* e *numpy*(Numpy Developers, 2019), foram produzidos e disponibilizados gratuitamente, trazendo a atenção da comunidade acadêmica e de pesquisa industrial.

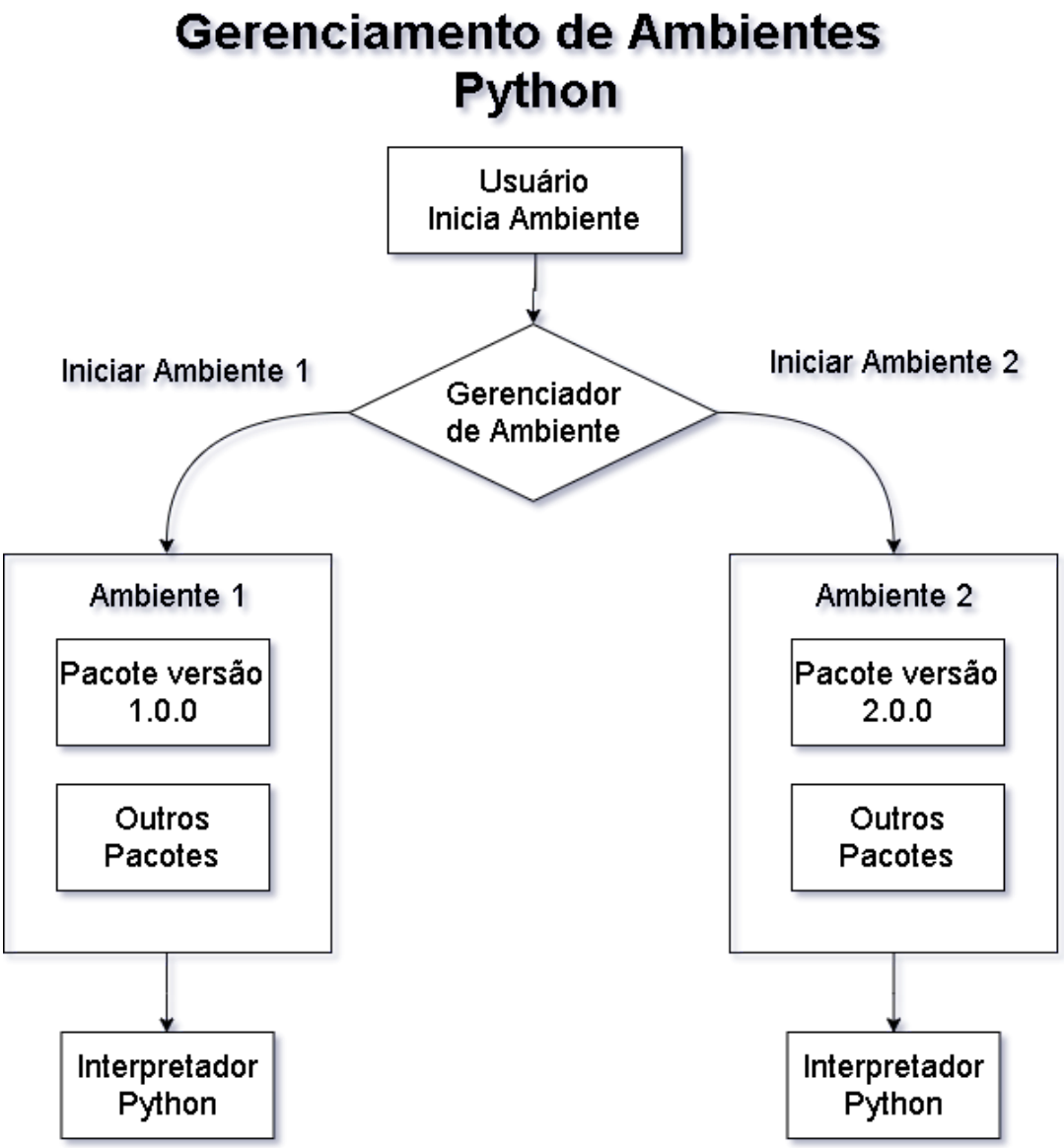
2.3.2 Ambientes *Python*

Para armazenar seus pacotes, *Python* goza de um conceito chamado ambiente. Este conceito permite que o desenvolvedor crie ambientes com diferentes pacotes para propósitos diferentes, evitando colisões do interpretador com pacotes de versões diferentes durante a execução de scripts *Python*.

Gerenciadores de pacote como *conda*(INC, 2019b) auxiliam na criação e instalação de pacotes em ambientes definidos pelo desenvolvedor. Um desenvolvedor pode ter um ambiente para desenvolvimento gráfico com inúmeros pacotes para este propósito, e um para desenvolvimento matemático com poucos pacotes.

A Figura 6 apresenta um diagrama de como ambientes são gerenciados por gerenciadores como *conda* e *virtualenv*.

Figura 6 – Esquema de carregamento de ambiente de pacotes *Python*



Fonte: Autor

2.4 Sobre Controle de Versão de *Software*

Sistemas de Controle de Versões (*VCS*, do inglês *Version Control System*) permitem que desenvolvedores de *software* mantenham acesso a todas as versões de um projeto em desenvolvimento, facilitando a organização e coordenação de membros de equipe durante o desenvolvimento do *software*. Um VCS preferencialmente dá suporte em evitar condições de corrida durante modificações simultâneas do projeto, permitindo que membros da equipe trabalhem em paralelo sobre o mesmo código sem perder suas modificações individuais durante a integração (ZOLKIFLI; NGAH; DERAMAN, 2018).

2.4.1 Controle de Versão com *git*

O *software git*(CONSERVANCY, 2019) é um sistema de controle de versão distribuído. Isto significa que o código fonte completo se encontra em um local em rede chamado de repositório remoto, reconhecido pelo programa. Este sistema é ilustrado pela Figura 7.

Figura 7 – Ilustração do Sistema *git*

Fonte: Autor

Desenvolvedores executam uma ação chamada *clone* para obterem a cópia mais recente deste repositório, que então é carregado para o computador como uma pasta de arquivos local.

Após a obtenção da cópia local, são realizadas as modificações de código fonte em suas máquinas. Ao final da modificação atual, é realizado um processo de organização chamado *add*, informando *git* sobre todas as alterações que compõem aquela mudança, denominada *commit*. Todo *commit* acompanha uma mensagem no histórico, que deve ser descritiva e relevante às alterações que ela agrupa. Após isso, o Desenvolvedor envia o seu *commit* utilizando o comando *push* para o repositório remoto, onde aparece como um ponto no histórico.

Caso hajam múltiplos desenvolvedores trabalhando no mesmo projeto, *git* mantém o controle de colisões do histórico. Se *commits* afetarem os mesmos arquivos, é informado ao último Desenvolvedor que enviou sua modificação que este deve ajustar os conflitos manualmente antes de ter suas mudanças no histórico remoto.

2.5 Sobre Testes de *Software*

Para garantir que as funcionalidades implementadas comportam-se da maneira esperada pelo autor, são realizados testes. O tipo mais básico de teste possível é o chamado teste de unidade, onde módulos principais do código fonte são verificados separadamente para validar seu funcionamento especificado (HUIZINGA, 2007) .

Estes módulos são pequenos blocos de entrada e saída que testam o comportamento dos menores elementos do código fonte. A saída então é comparada com o esperado, definido pelo desenvolvedor, de modo que se detecte ocorrências de comportamento inesperado advindo da implementação atual.

2.5.1 Integração Contínua (CI)

Integração Contínua é um conceito recente de certificação de qualidade em desenvolvimento de *software* que envolve um *VCS*, testes, e um servidor. Este servidor de *CI* se atualiza com uma cópia *VCS* a cada nova modificação e as testa automaticamente. O servidor tem um canal de notificações que informa se as modificações causaram erros durante os testes.

2.5.2 Entrega Contínua (CD)

Entrega Contínua é um conceito semelhante a Integração Contínua, onde o servidor encarrega-se de atualizar os canais de distribuição de *software* a cada atualização no *VCS* que passe pela etapa de testes, efetivamente automatizando a distribuição do *software* a cada versão nova.

2.6 Sobre Licenças de *Software*

Todo *software* é sujeito a propriedade intelectual. Por padrão, um *software* e seu código fonte são de propriedade e de direitos reservados ao autor, mesmo que este seja disponibilizado em rede (GitHub, 2019). É necessário que toda atividade envolvendo *software* e código tenha claras definições de licenciamento para que o *software* possa ser desenvolvido, vendido e utilizado de maneira correta para proteger tanto os usuários quanto os direitos do autor.

Software de código aberto é um conceito onde o código fonte de um *software* se encontra licenciado de modo que ele é disponibilizado a qualquer agente para que este possa copiar, distribuir e usufruir do mesmo. Isto ainda mantém os direitos intelectuais do autor.

Licenças de código aberto podem conter cláusulas que obrigam certas práticas. Uma licença que exige a ação de *Copyleft* demanda que cada cópia do *software* distribuído acompanhe também uma cópia da própria licença. Muitas vezes, *Software* derivado de um *software* de código aberto deve, por obrigação de licença, também ser aberto. Certas cláusulas podem trazer problemas para empresas que pretendem manter certas implementações em segredo, no evento de que um desenvolvedor utilize parte de código aberto no desenvolvimento corporativo.

Algumas licenças de código aberto podem ser descritas na Tabela 1, informando algumas das cláusulas sobre redistribuição, modificação, e proteção de patentes do autor pertinente a software licenciado.

Licenças de Código Aberto			
Nome	Redistribuição	Modificações	Proteção de Patentes
Apache License	Livre	Livre	Sim
Eclipse License	Limitado	Limitado	Sim
GNU General Public License	Copyleft	Copyleft	Sim
Creative Commons BY-SA	Copyleft	Copyleft	Não

Tabela 1 – Exemplo de licenças de código Aberto (OPENSOURCE.ORG, 2020)

2.7 Magnetismo

Magnetismo é o fenômeno onde um material ou objeto causa uma força, descrita como magnética, que repele ou atrai outros materiais e objetos susceptíveis.

Este fenômeno é consequência de um movimento de cargas elétricas, algo que também surge em condutores elétricos de diversos tipos. Estas cargas em movimento são descritas em campos vetoriais, denominados campo magnético. (CALLISTER, 2011)

2.7.1 Ímãs

Um ímã é um material ou objeto que produz um campo magnético passivamente. Imãs permanentes são uma classe de ímã onde o material foi magnetizado.

A maioria dos materiais que podem ser magnetizados e que são atraídos por ímãs são referidos como ferromagnéticos, que podem ser considerados leves ou duros.

2.7.2 Dipolos Magnéticos

Dipolos magnéticos são a representação teórica de um ímã fundamental, ou momento magnético. Estes possuem o nome de dipolo devido ao fato de terem um polo norte e um polo sul, apresentando dois polos.

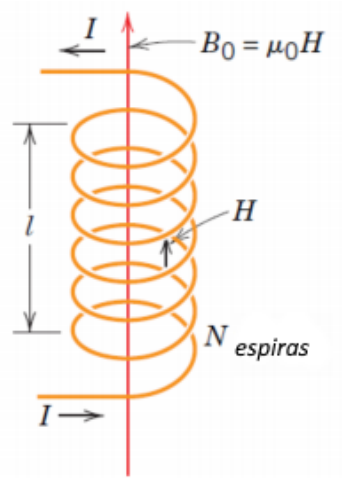
Um elétron rotacionando sobre seu eixo pode ser abstraído como um dipolo, ou momento magnético. Quando os momentos magnéticos dos elétrons de um material estão alinhados em uma mesma direção, é possível abstrair isso como vários pequenos dipolos cujos campos serão superpostos. Caso hajam alinhamentos suficientes, o

material será um ímã de dois polos. Com isso em mente, todo ímã é considerado um dipolo.

2.7.3 Definição de Campos Magnéticos

Campos magnéticos são descritos como campos vetoriais. A intensidade do campo pode ser descrito pela letra H. O equacionamento de um campo gerado pela bobina da Figura 8 é dado pela Equação 2.1, a seguir.

Figura 8 – Ilustração de uma bobina magnetizada



Fonte: Adaptado de (CALLISTER, 2011)

$$H = \frac{NI}{l} \tag{2.1}$$

Onde H é a intensidade do campo magnético, em $\frac{A}{m}$, N é o número de espiras da bobina, I é a magnitude da corrente em Amperes, e l é o comprimento em da bobina em metros.

A densidade de fluxo magnético, chamado também de indução magnética, é designado pela letra B, expressa pela Equação 2.2

$$B = \mu H \tag{2.2}$$

Onde B é a densidade de fluxo magnético dado em T, μ é a permeabilidade magnética do material em $\frac{H}{m}$.

3 DESENVOLVIMENTO

Neste capítulo serão descritos os passos tomados durante a execução do projeto.

3.1 Desenvolvimento do Projeto Proposto

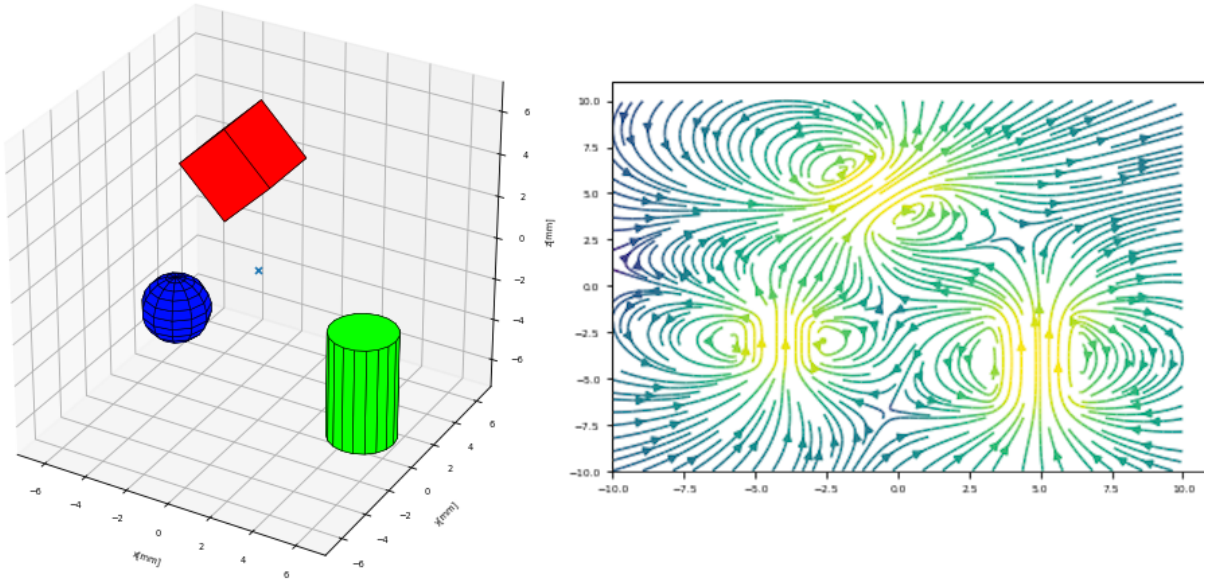
Nesta seção será discutido o desenvolvimento do trabalho proposto, sendo também expostos os requisitos e uma visão geral do sistema implementado.

3.1.1 Do estado inicial do projeto

A biblioteca de *software magpylib* que foi desenvolvida é voltada para a simulação de valores específicos de campo eletromagnéticos de ímãs fundamentais em um espaço cartesiano tridimensional. Isso tem o objetivo de auxiliar no desenvolvimento de projetos microeletrônicos com a presença de sensores de campo eletromagnético tridimensionais e ímãs (SILVA, 2018). A implicação do projeto é de que uma ferramenta deve não só realizar uma computação, mas ser relativamente eficiente, simples de instalar, usar e testar. Inicialmente, foi providenciado um protótipo da biblioteca de *software* feita em *Python 3*. Esta versão inicial já incluia:

a) Fórmulas analíticas para calcular campos de ímãs fundamentais a partir de valores de posição inseridos, incluindo:
    a. Modelos de ímãs Cubóides, Cilíndricos, Esféricos;
    b. Modelos de Linhas de Corrente e Correntes Circulares.
b) Um sistema de aglomeração de ímãs para superposição de campos,
c) Um sistema de manipulação cartesiano para posicionamento, rotação e orientação de ímãs e aglomerações
d) Um sistema de visualização do formato e posições tridimensionais destas aglomerações

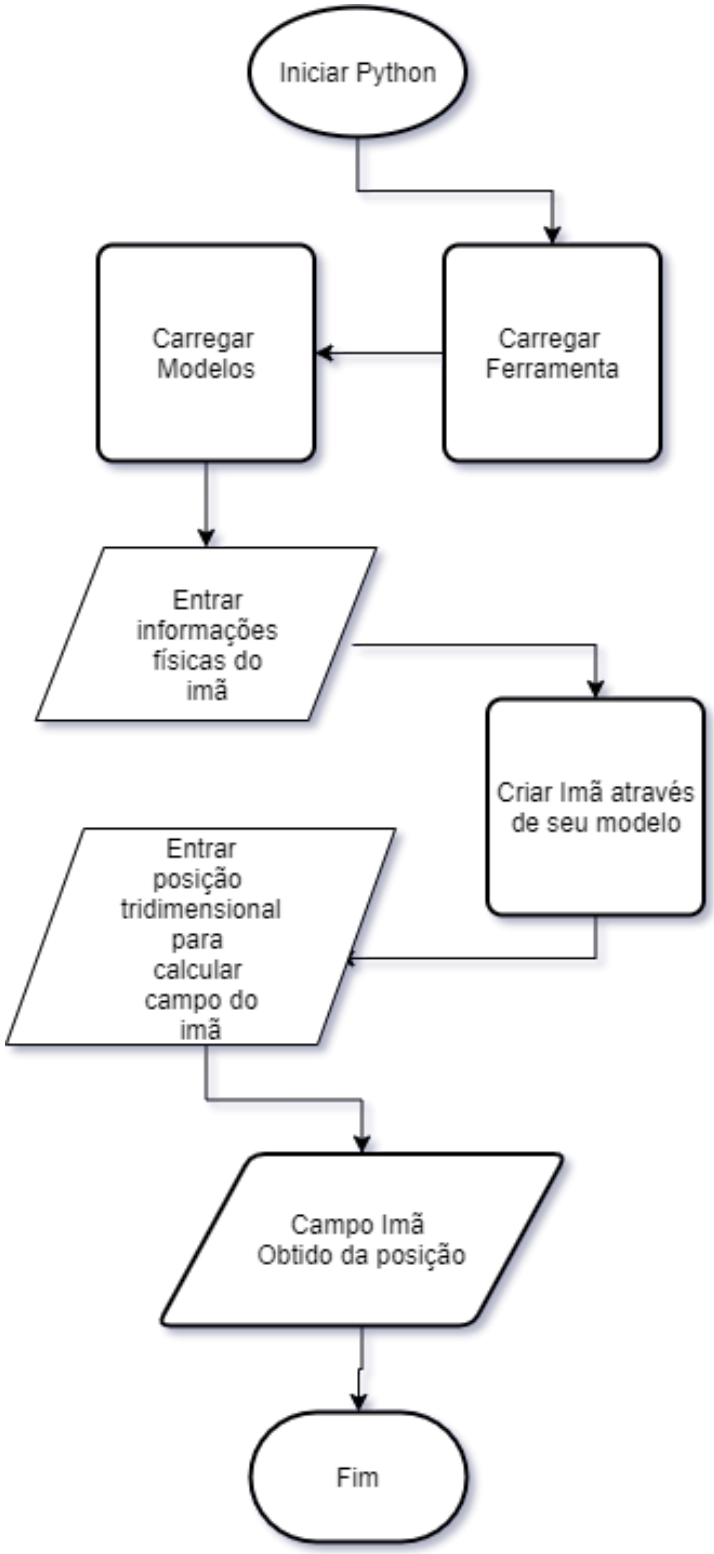
Figura 9 – Visualizador do Programa; uma aglomeração (Collection) de ímãs fundamentais Cubóide, Esfera e Cilindro a esquerda, e a análise de campo B superposto a partir de uma varredura no eixo horizontal X.



Fonte: Autor

Com o protótipo, funções básicas da ferramenta já eram de bom proveito para a equipe interna, permitindo que o usuário inserisse a definição de alguns ímãs e a posição desejada, fazendo com que o *software* retornasse o valor de campo magnético em militesla (mT). A visualização de aglomerados de 1 ou mais ímãs já estava funcional como é observável na Figura 9. O fluxo de uso básico da ferramenta pode ser observado na Figura 10.

Figura 10 – Fluxo de uso básico da Ferramenta



Fonte: Autor

Para a implementação da linguagem de programação *Python 3* sobre o interpretador *CPython*, a ferramenta deveria ser um módulo/pacote, entretanto ela estava sendo utilizada como um *script* de usuário dentro de projetos de design, o que não só dificultava a instalação mas causava conflitos entre versões do *software*. A distribuição das versões de *software* original era através dos diretórios da biblioteca compactados em um arquivo de formato *ZIP*, enviados manualmente por canais internos de *e-mail*.

Não haviam testes para as funções ou métodos da ferramenta. O funcionamento do *software* era validado através de alguns procedimentos manuais a cada versão. Isso coloca em risco a confiabilidade da ferramenta, já que com o acréscimo de funções também ocasionaria no acréscimo de testes e também o acréscimo de erro humano com o procedimento repetitivo (HUIZINGA, 2007)

A documentação da ferramenta dava-se apenas por comentários *docstring* no código fonte. Isso oferecia algumas explicações e exemplos minimalistas, mas era necessário abrir o código fonte ou utilizar um painel de ambiente de desenvolvimento para estudar seu conteúdo. Isso provou-se insuficiente para que usuários novos da biblioteca pudessem utilizar a ferramenta sem o auxílio constante dos desenvolvedores.

Para o processo de design de projetos microeletrônicos, ainda haviam funcionalidades a desejar. Certas rotinas comuns ainda eram implementadas manualmente pelos usuários com *Python 3*, o que era uma distração para o projetista da microeletrônica. Uma rotina pertinente era de investigação de um movimento, uma rotação ou translação de um ímã através do espaço. A coleta de cada amostra para cada posição tornava-se complicada e temporalmente custosa devido aos arranjos de rotinas em *Python 3*, o que não só era ineficiente mas complexo demais para um usuário sem conhecimento aprofundado da linguagem.

Não havia uma ciência sobre a licença de *software* a ser utilizada para o projeto, apesar da intenção do desenvolvedor original de distribuir o *software* da empresa gratuitamente.

3.1.2 Dos requisitos

Reconhecendo o funcionamento existente da ferramenta, foi realizada uma reunião sobre os possíveis requisitos a serem introduzidos. Estes requisitos adicionais foram separados em funcionais e não-funcionais (SOMMERVILLE, 2006). Destes observam-se:

Requisitos funcionais:

a) Calcular múltiplas posições de campo para ímãs individuais e aglomerados (*Collection*).

b) Trabalhar com o modelo matemático de um momento de dipolo magnético;

c) Calcular campos específicos durante o movimento de ímã ;

d) Opcionalmente mostrar a direção de corrente ou o vetor de magnetização de modelos na interface de visualização do sistema;

e) Opcionalmente mostrar marcadores na interface de visualização do sistema;

f) Inserção de sensores tridimensionais independentes, verificando ímãs individuais ou um aglomerado (*Collection*) de ímãs, para cálculo relativo.

Requisitos não-funcionais:

a) Ser intuitiva ao usuário, com estrutura e nomes que seguem uma lógica proposta

b) Ser instalável como um pacote *Python 3*

c) Ser difícil de utilizar erroneamente, checando valores de entrada para o usuário e fornecendo mensagens de ajuda.

d) Seguir um padrão com comentários descritivos para cada função

e) Manter a documentação sempre atualizada e acessível para usuários e desenvolvedores com internet

f) Ter canais de distribuição acessíveis pela internet e sincronizadas com todas as versões.

g) Possuir robustez, testes que protegem o código fonte de erro humano por parte dos desenvolvedores

h) Demonstrar performance eficiente, respeitando todos os requisitos anteriores

i) Funcionar em ambientes que aceitam *Python 3* e suas bibliotecas científicas.

3.1.3 Preparação para o Desenvolvimento

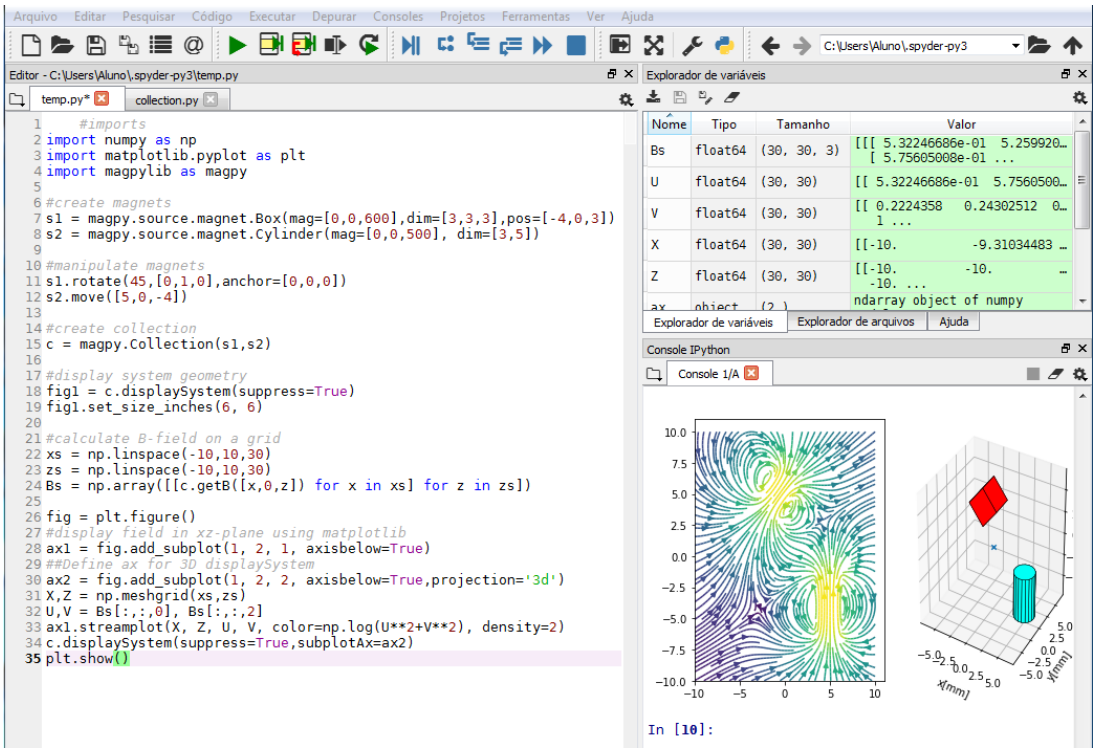
Para a execução do projeto, era necessário a instalação de *softwares* de desenvolvimento diversos. Havia um computador pessoal da empresa com o sistema operacional *Windows 7 Pro*, que não fornecia acesso administrativo ao usuário padrão, complicando esta necessidade. Programas de desenvolvimento eram instalados por *tickets* enviados ao setor de informática da empresa que, ao receber a mensagem, entravam com permissão para efetuar a instalação de aplicativos dentro do tempo de uma semana.

Como a longo prazo os pedidos de instalação se tornariam um impedimento no processo de desenvolvimento, foi realizado o pedido para a instalação do *software hypervisor VirtualBox* (ORACLE, 2019) . Através deste, fora realizada a instalação de uma máquina virtual contendo o sistema operacional *Lubuntu 32bits* (LUBUNTU, 2019). Isto permitiu o acesso administrativo ao usuário padrão dentro deste ambiente virtualizado, permitindo o processo de desenvolvimento de maneira mais livre e agilizada. O Trabalho do projeto se inicializou com o recebimento da última versão da ferramenta

em formato *zip* por canal de *e-mail* interno. Para utilizar todos os recursos desta versão inicial, era necessário um interpretador *Python 3* instalado na máquina com alguma versão não definida da biblioteca de *Python 3* numérica *numpy* e da biblioteca gráfica *matplotlib*.

Foi instalada a distribuição científica Anaconda3 (INC, 2019a), que forneceu todos os requerimentos da versão atual da ferramenta. Esta distribuição também incluiu dois ambientes de desenvolvimento integrados (*IDEs*): *Spyder* (SPYDER, 2018) e *Visual Studio Code* (MICROSOFT, 2019). O ambiente de desenvolvimento pode ser compreendido pela Figura 11.

Figura 11 – *Spyder IDE* executando a ferramenta, carregado pelo *Anaconda*, calculando a aglomeração de campo magnético no eixo X de um cubo e um cilindro em um espaço cartesiano tridimensional.



Fonte: Autor

3.2 Implementação de Controle de Versão

A distribuição das versões de *software* original se dava por diretórios da biblioteca compactados em um arquivo de formato *zip*, enviados manualmente por canais internos de e-mail. Enquanto “eficaz” em pequena escala, isso apresentava os seguintes problemas:

a) Sem auxílio do desenvolvedor, não era possível ter certeza de qual versão do *software* encontrava-se no arquivo *zip*, ocasionando em problemas tanto na distribuição como na atualização do *software*.

b) Trabalhar sobre o *software* em uma equipe de duas ou mais pessoas era algo complexo, requisitando uma organização que obrigava os desenvolvedores a se separarem ao invés de trabalharem juntos por medo de sobrescrever mudanças.

c) Enxergar modificações aprovadas ou em revisão dependia inteiramente de documentação perfeita da mesma, permitindo que erros não documentados passassem despercebidos.

Como solução, foi proposto pelo autor a utilização do *software* de controle de versão *git* aplicado a uma conta do serviço *GitHub*, identificados por suas logomarcas na Figura 12. Em áreas próximas ao desenvolvimento, a combinação da ferramenta com o serviço é comumente confundida como uma só.

Figura 12 – Logomarca do *software VCS git* à esquerda, e a logomarca do serviço de hospedagem de repositórios *git*, *GitHub*, à direita.

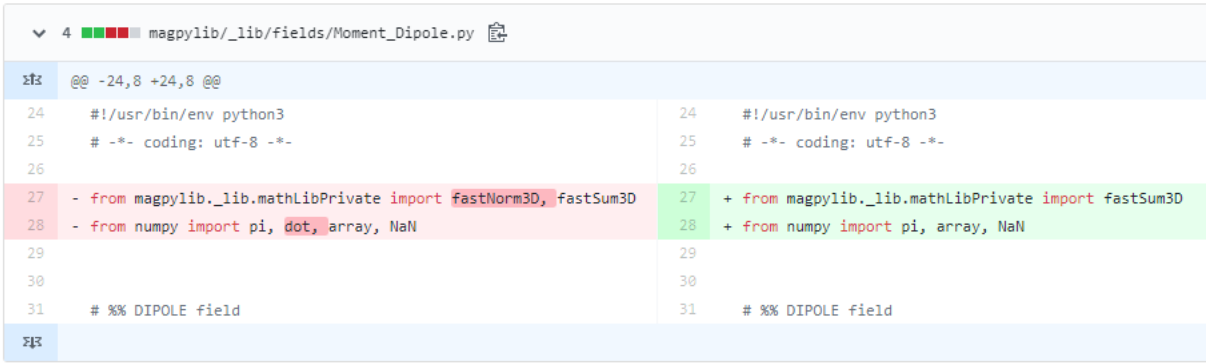
Fonte: https://git-scm.com/

Esta conta seria criada para apenas para o projeto. A implementação inicial se deu da seguinte forma:

1. Criar uma conta para administração do projeto em um serviço de hospedagem de repositórios *git* (*GitHub*)
2. Criar um repositório remoto privado para o projeto através da interface do serviço de hospedagem de repositórios *git*
3. Instalação do *software git* na máquina local do desenvolvedor
4. Utilizando *git* na máquina local, clonar repositório remoto utilizando as credenciais da conta administrativa, criando um repositório local na máquina. Este repositório local é representado por uma pasta no sistema do desenvolvedor.
5. Adicionar todos arquivos da versão inicial do *software* para dentro do repositório local.
6. Confirmar (*add, commit*) mudanças no repositório local.
7. Enviar (*push*) mudanças confirmadas do repositório local para o repositório remoto com as credenciais da conta administrativa.

Com a conclusão dessas etapas iniciais, foi possível rastrear qual a versão sendo disponibilizada aos membros da equipe. Manter o controle de quaisquer mudanças confirmadas pelos desenvolvedores tornou-se trivial, já que o sistema anota qualquer modificação realizada, como apresentado na Figura 13. O sistema possui uma interface *web*, e informa mudanças realizadas em parte de um arquivo de código em duas colunas (*split*). A coluna esquerda representa a versão antiga do arquivo, e a coluna direita representa a versão nova. As linhas com os sinais “-”apresentam-se elementos apagados enquanto os sinais “+”são as modificações introduzidas. O repositório remoto serve também como modo de backup do projeto.

Figura 13 – Interface no serviço *GitHub*.



Fonte: Autor

3.2.1 Planejamento de Fluxo de Trabalho

A implementação *VCS* no projeto já estava efetiva, entretanto ainda era necessário discutir uma organização para garantir a sua utilização eficiente entre os membros da equipe. A interface web do serviço *GitHub* providencia os seguintes recursos para facilitar a organização de um time de desenvolvedores:

a) Página de Tarefas do Projeto

b) Página de Revisão de Mudanças Propostas

c) Página de visualização de ramos de versões.

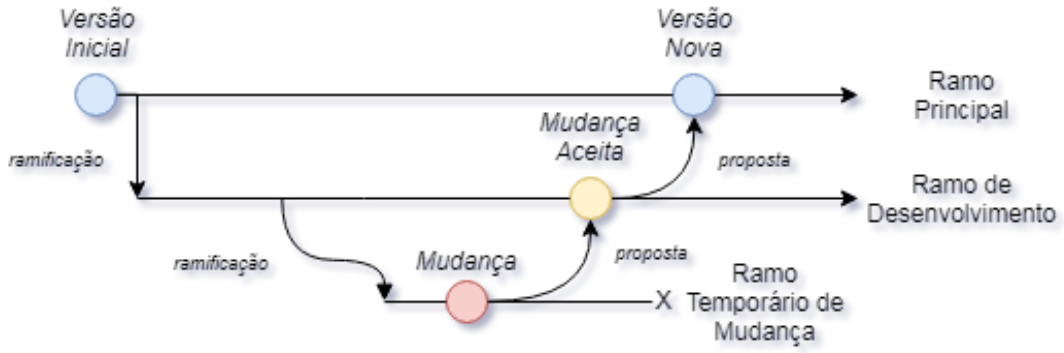
Em reuniões, o seguinte fluxo foi proposto para organizar o processo de desenvolvimento e manter o histórico do *VCS* organizado:

1. Ter um ramo (*branch*) de versão principal.
2. Ter um ramo de versão em desenvolvimento.
3. Para qualquer tarefa de mudança, criar um ramo temporário para realizar aquela tarefa.
4. Para qualquer modificação concluída, realizar uma proposta de mudança (*pull request*), seguida pela revisão de outros desenvolvedores. Uma proposta pode conter várias mudanças (*commits*).
5. Todas as propostas aceitas são aplicadas (*merge*), primeiramente, ao ramo de desenvolvimento. O ramo principal não será modificado.
6. Confirmada a conclusão e robustez da versão do ramo de desenvolvimento, se cria uma proposta para atualização do ramo principal, com uma revisão adicional.

7. Cria-se uma identidade (*tag*), marcando a nova versão no histórico.

Na Figura 14, está ilustrado o Fluxo de *VCS* proposto. Um ramo é representado por setas horizontais, significando histórico de mudanças do projeto no tempo. Setas para baixo demonstram ramificações (*branch*), copiando o histórico do ramo original. Círculos representam modificações (*commits*) realizadas em cada ramo. Setas para cima demonstram propostas de mudanças de um ramo ao outro (*pull request*), juntando os históricos de mudanças dos ramos (*merge*). 'X' representa a deleção do ramo.

Figura 14 – Fluxo de VCS proposto.

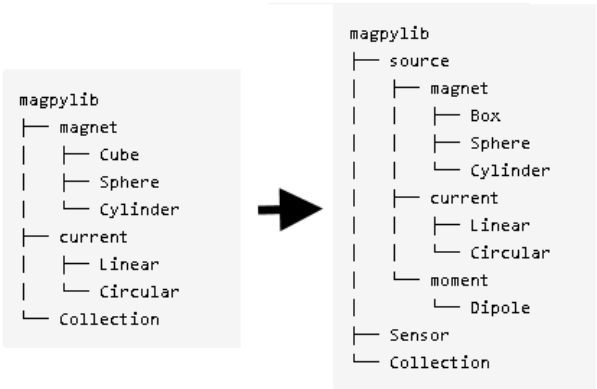


Fonte: Autor

3.3 Organização dos módulos das classes

A segunda tarefa foi discutir e implementar uma nova estrutura de chamada de classes do *software*. A ferramenta utiliza o paradigma orientado a objetos para carregar os modelos matemáticos em variáveis, e a obtenção dos campos é realizada através dos métodos dos objetos criados

Figura 15 – Organização original dos módulos, à esquerda, e organização final, à direita.



Fonte: Autor

Como visto na Figura 15, foi inserido um novo módulo chamado *source*, onde todas as fontes de campo magnéticos se encontrariam. Um módulo chamado

Sensor estava planejado para implementação, e foi inserido. Os comentários foram padronizados com o padrão *Numpy* (NUMPY, 2019)

3.3.1 Entendendo e Adicionando um Modelo de fonte de campo

Na ferramenta, um modelo de ímã é dado pelas seguintes características:

a) Construtor do objeto;

b) Atributos de dimensão, posição, vetor de magnetização e orientação do objeto;

c) Método com a fórmula de cálculo de campo;

d) Métodos para manipulação espacial;

e) Dados de visualização tridimensional.

Todos os modelos de atributos eram localizados em uma estrutura comum que era herdado por objetos de mesma origem. Destes eram:

a) Imãs Homogêneos (*magnet*);

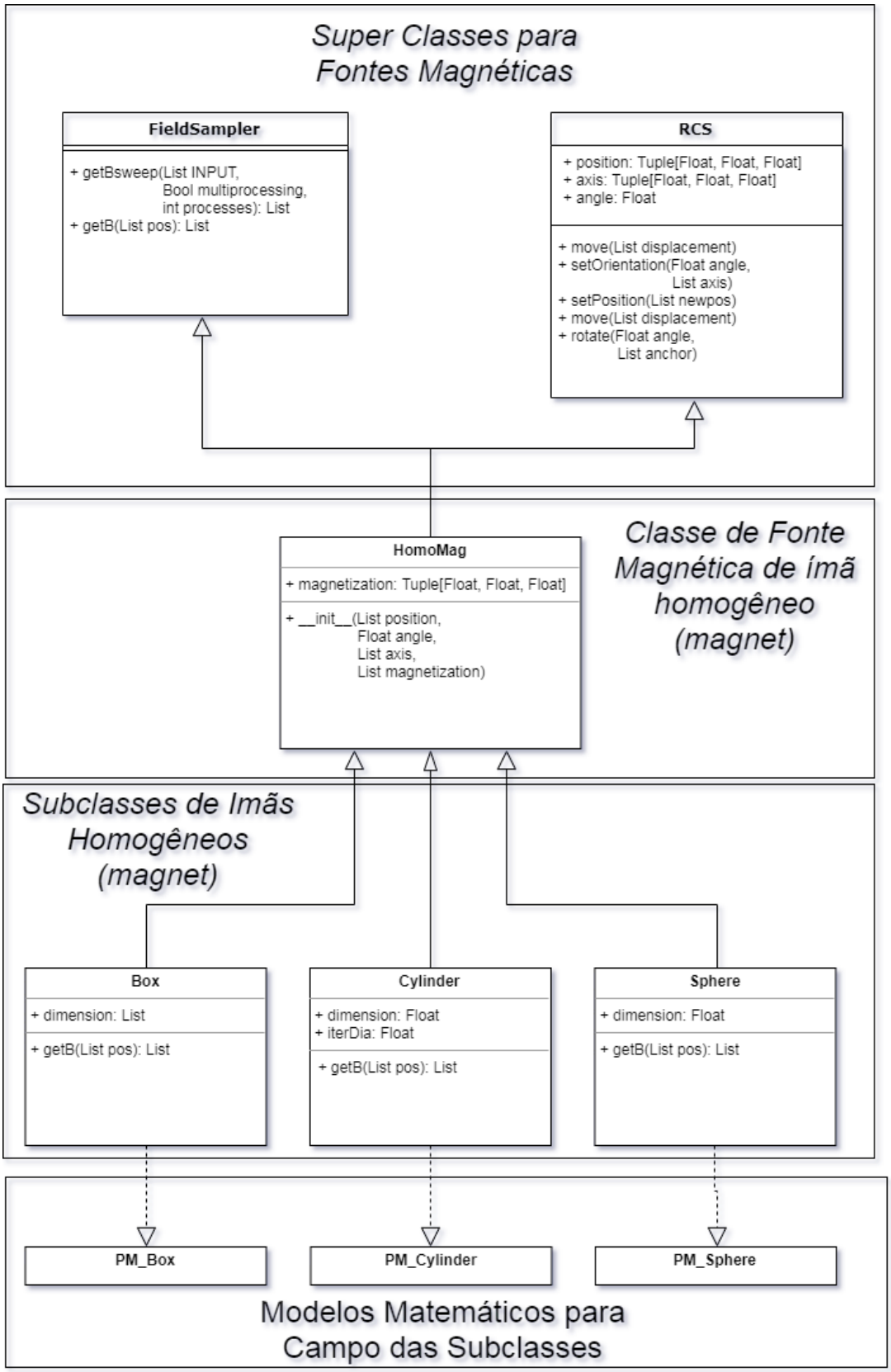
b) Correntes (*current*);

c) Momentos (*moment*).

A visualização do diagrama de blocos para ímãs homogêneos pode ser visualizada na Figura 16. Observa-se a modularização dos atributos de objeto para manipulação, e também das dependências matemáticas. Isso permite que o desenvolvimento destes módulos sejam trabalhados separadamente.

Os métodos de manipulação espacial são dados por uma superclasse existente chamada RCS (do inglês, *Relative Coordinate System*), providenciando uma maneira conveniente de rotacionar, deslocar e orientar os atributos do modelo em um espaço tridimensional. Uma superclasse chamada *FieldSampler* contém as rotinas para cálculo de campo, que então são re-implementadas para cada subclasse durante a herança enquanto mantendo um padrão definido. As subclasses implementam os atributos de dimensão física daquele tipo de ímã, e dependem dos modelos matemáticos para cálculo de campo para aquele tipo.

Figura 16 – Diagrama de classes para fontes de ímãs Homogêneos

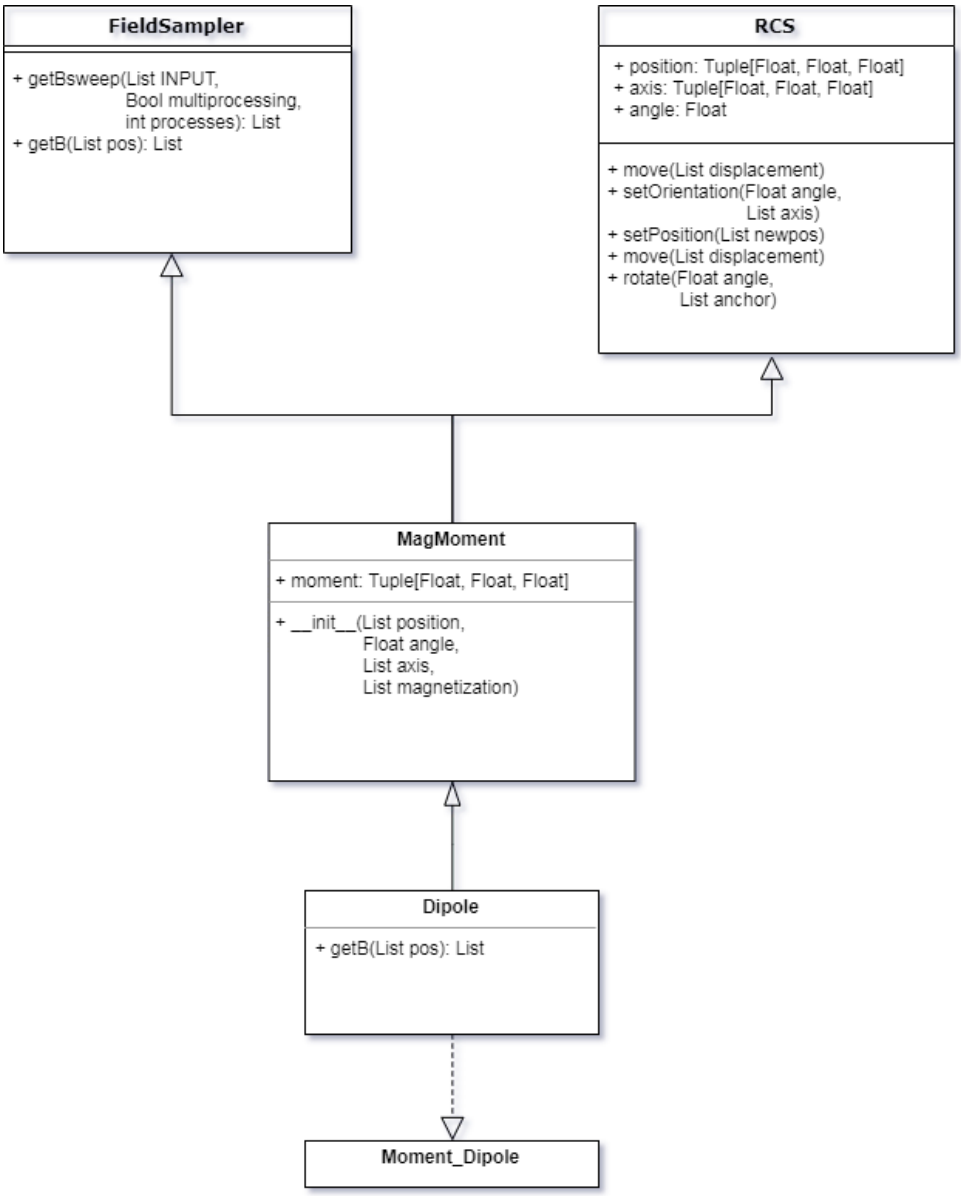


Fonte: Autor

O Modelo de um dipolo genérico, ou momento magnético, era de interesse para os usuários da ferramenta, logo foi extraída uma fórmula analítica para cálculo de campo de um dipolo a partir de literatura (JACKSON, 1975) e transformada em um método de cálculo de campo chamado *Moment_Dipole*. A estrutura desta nova classe de Momentos e sua subclasse dipolo é dada pela Figura 17.

Ilustrou-se o diagrama de classe de momentos magnéticos *MagMoment*, herdando superclasses base, *RCS* e *FieldSampler*, implementando a subclasse dipolo. Esta subclasse depende do modelo matemático de um dipolo, que foi implementado em um módulo chamado *Moment_Dipole*.

Figura 17 – Implementação da classe dipolo



Fonte: Autor

3.4 Implementação em *Python*

Esta seção é dedicada ao trabalho realizado no projeto utilizando Python.

3.4.1 Paralelizando Cálculos em *Python*

Um recurso desejado pelos usuários era a de realização de cálculos para vários pontos de campo simultaneamente, pois o tempo computacional de uma rotina sequencial para isso era demasiadamente custoso. Assim, foi realizado um estudo sobre maneiras de alcançar um cálculo paralelo utilizando múltiplos núcleos do processador.

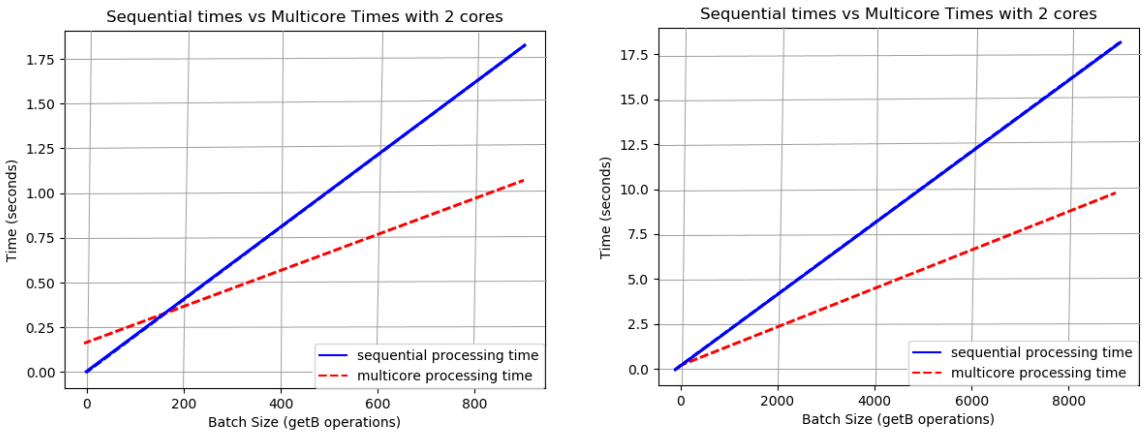
O módulo *multiprocessing* da biblioteca padrão *Python* disponibiliza ao usuário uma interface de paradigma funcional para a distribuição de tarefas em 1 ou mais núcleos do processador. Foi realizada a construção de uma interface para o usuário que permite a inserção de vários campos chamada *getBsweep()*. Esta função possui dois modos, como requisitado pelos usuários:

1. Modo Sweep. Este modo recebe várias posições de campo em uma lista. Cada ponto é enfileirado para um núcleo da CPU, calculado, e retornado ordenadamente.

2. Modo Motion. Este modo recebe várias posições de campo, de posição e de orientação. A implementação disto consistia em modificar ou manter a posição do sensor, da fonte, e também da orientação, permitindo que fossem calculados passos discretizados de um movimento ou rotação de imã ou sensor.

A análise de desempenho foi realizada em acompanhamento com a Figura 18. À direita mostra a perspectiva de tempo para 10.000 pontos calculados. Observa-se desvantagem no tempo de execução caso hajam menos de 100 pontos devido ao overhead de preparação algoritmo

Figura 18 – Benchmark de tempo de execução utilizando 2 núcleos, do método criado pelo autor, em linha tracejada, versus o método sequencial dos usuários, em linha sólida



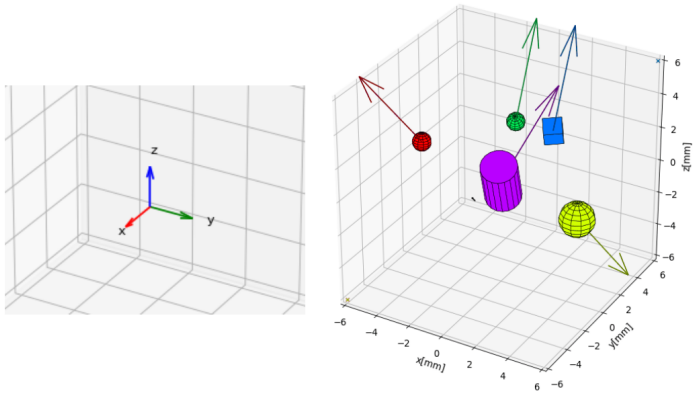
Fonte: Autor

3.4.2 Atualizações e Adições de Funcionalidades Gráficas

A ferramenta utilizava de uma biblioteca de visualização de gráficos em eixos *plots* chamada *matplotlib*. Esta biblioteca contém suporte para *plots* de eixos tridimensionais, providenciando uma interface em 3D na qual é possível utilizar linhas e áreas fechadas. A ferramenta já possua uma interface para os elementos existentes.

O trabalho foi realizado sobre a estrutura já existente e sobre a biblioteca gráfica escolhida. Foi inserido um visualizador de vetores para cada objeto, onde o vetor de magnetização era normalizado e transformado em uma flecha utilizando a função *quiver*, uma funcionalidade da biblioteca gráfica. Foi adicionado também um identificador de direção de corrente, e funções de retorno para armazenamento automático de gráficos. Foi também adicionado um sistema de marcadores, permitindo o usuário a marcar pontos e escrever texto ou valores em posições marcadas no gráfico. Estes podem ser visualizados na Figura 19.

Figura 19 – Visualização de um Sensor com marcadores à esquerda, e de vetores de magnetização de imãs à direita



Fonte: Autor

3.4.3 Criando Testes de Unidade

A realização de testes do *software* foram implementadas com a ferramenta *pytest*. Esta ferramenta permite que o desenvolvedor crie *scripts* que apenas agrupa funções de teste em *Python*. Cada função apenas verifica se certo fluxo de execução (definido pelo autor) tem o resultado esperado, permitindo que fluxos principais para funcionalidades básicas sejam listados.

Demonstrado pela Figura 20, foi definido uma entrada de interesse. A ferramenta tentará calcular um campo dentro do dipolo com esta entrada, o que deve resultar em uma singularidade. Deste fluxo, é esperado da implementação que ocorra um alerta *RuntimeWarning* e o retorno do valor *NaN* em todos os eixos do campo ao detectar a singularidade. Testes como esse foram implementados para todos os métodos básicos e também para diversos casos específicos.

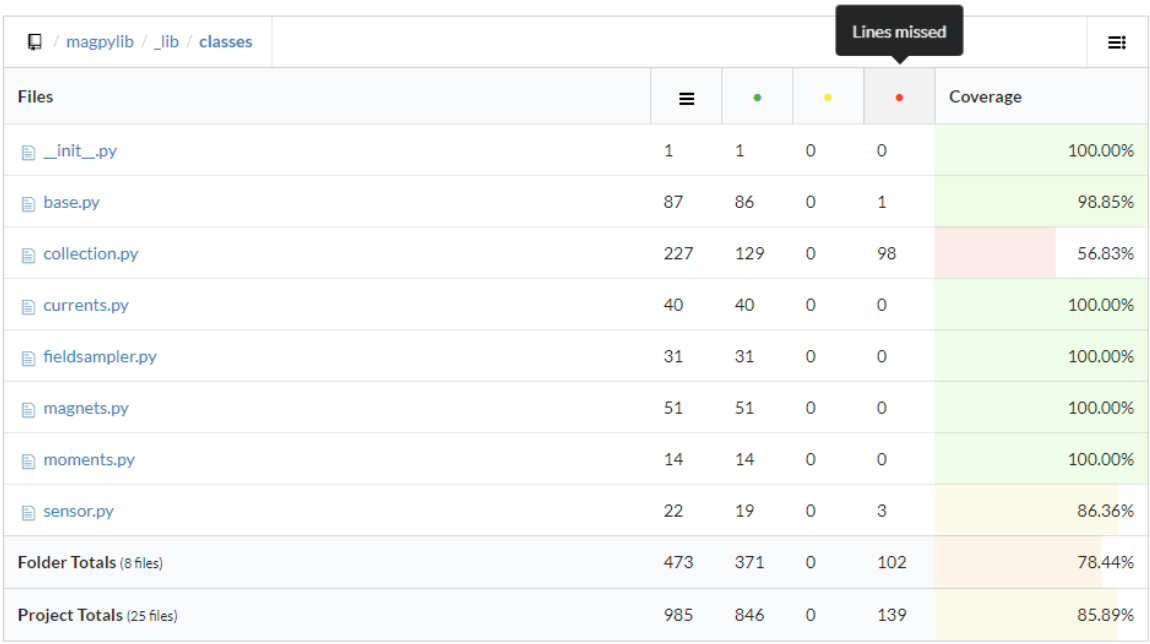
Figura 20 – Uma função de teste localizada no agrupamento de testes do modelo matemático de campo do dipolo

```
6   def test_Bfield_singularity():
7       # Test the result for a field sample on the dipole itself
8       # Expected: NaN
9       from numpy import array
10
11      # Definitions
12      mag=array([-1,2,-3])
13      calcPos = array([0,0,0])
14
15      # Run
16      with pytest.warns(RuntimeWarning):
17          results = Bfield_Dipole(mag,calcPos)
18          assert all(isnan(axis) for axis in results)
```

Fonte: Autor

Ao configurar e chamar a ferramenta, são identificados estes agrupamentos de teste, e todas as funções são executadas. Relatórios de cobertura de testes do código fonte são gerados após a execução da ferramenta, apontando se houve comportamento inesperado em algum teste específico, e listando também as linhas do código fonte que ainda não foram testadas. Estes relatórios podem ser interpretados por serviços de cobertura de testes, como demonstrado pela Figura 21.

Figura 21 – Um dos visualizadores de relatório de testes, codecov.io



Fonte: (CODECOV, 2019)

Através de um visualizador de relatório, é possível facilmente verificar os arquivos do projeto cujas linhas não estão sendo testadas, oferecendo um porcentual de cobertura. Na Figura 21 este porcentual está na coluna *Coverage*, à direita.

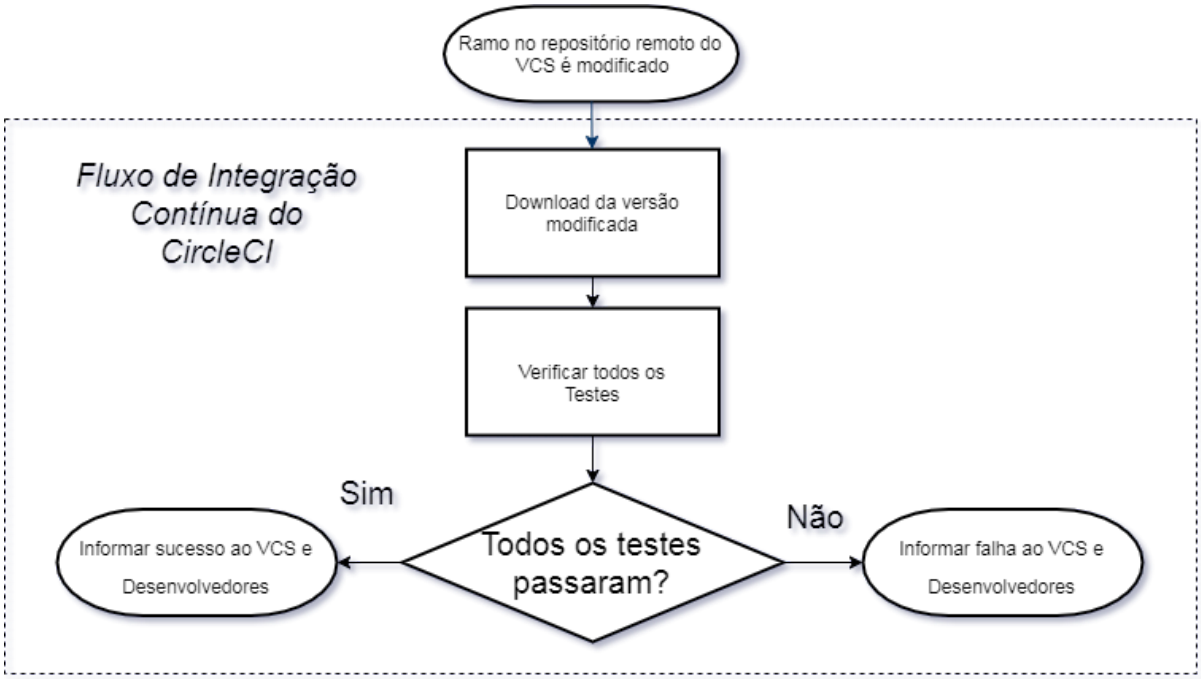
3.4.4 Implementando Integração Contínua

De modo a certificar que não foram introduzidos erros explícitos ao *software* durante o desenvolvimento, é necessário que todos os módulos básicos sejam validados ao menos uma vez a cada mudança introduzida (HUIZINGA, 2007). Para garantir a isso, foi realizada a implementação do conceito de Integração Contínua (CI). Integração Contínua implicita o uso de um *VCS* para ativar uma rotina em um servidor dedicado, que então realiza o download da versão mais recente no repositório remoto. Este servidor, então, segue uma rotina definida pelo desenvolvedor.

Para este projeto, o servidor dedicado para Integração Contínua utilizado foi do serviço *CircleCI*(INC., 2020). A rotina definida era de instalar o ambiente necessário, executar os testes, e verificar se houve erros. Um arquivo no *VCS* é dedicado para

os passos de integração contínua no *CircleCI*, que é configurado para observar o repositório do projeto. O fluxograma deste sistema implementado se encontra na Figura 22.

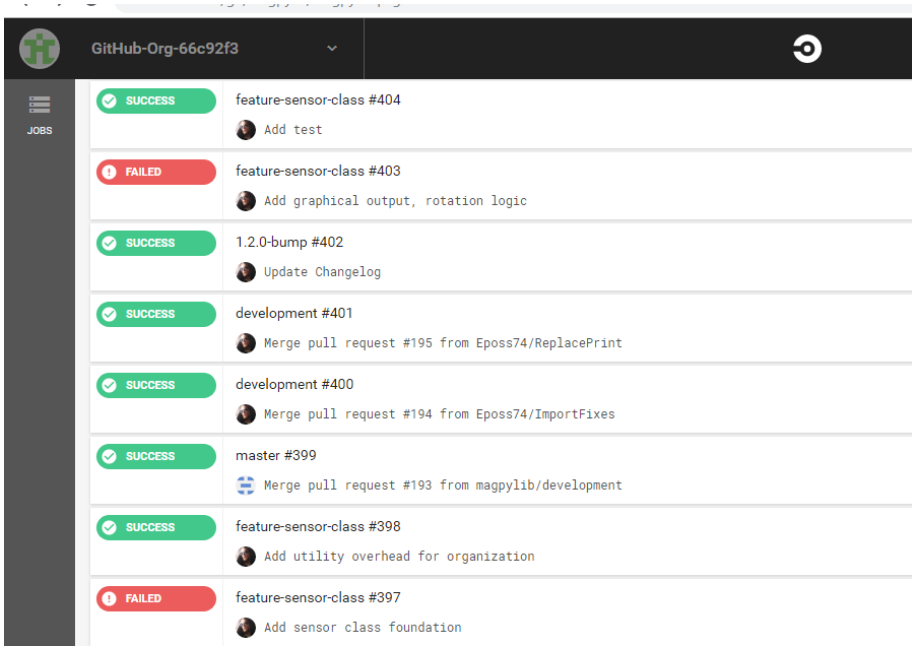
Figura 22 – Fluxograma do sistema de Integração Contínua implementado.



Fonte: Autor

Assim, toda modificação no repositório remoto em qualquer ramo do projeto é testada contra a base de testes fornecida para aquele ramo. Modificações no *VCS* que causam os testes a falharem no servidor dedicado são marcadas em vermelho, como mostrado na Figura 23.

Figura 23 – Página web do serviço CircleCI, mostrando resultados de testes para cada modificação realizada



Fonte: Autor

3.4.5 Documentando falhas com testes

Com a realização de testes, é possível definir requerimentos antes da execução da implementação, o que agilizou o desenvolvimento de novas funcionalidades. Além disso, esta estrutura permitiu capturar erros sutis porém críticos durante a implementação do projeto. Alguns modelos matemáticos apresentavam comportamento inesperado com certos parâmetros aceitáveis. Foi possível documentar este comportamento ao criar “anti-testes”, que testavam se a entrada resultava no resultado errôneo. Isso permitia que o projeto continuasse sendo testado para outras mudanças sendo realizadas em paralelo a correção de *bugs*, já que este comportamento e os passos para sua reprodução foram documentados e entendidos pelo sistema como algo esperado.

Assim, se fosse encontrada a solução para o *bug*, o resultado da funcionalidade estaria corrigido; o anti-teste iria falhar ao comparar o resultado correto com o errôneo, e os desenvolvedores seriam notificados para atualizar o teste com o resultado correto.

Com isto, uma correção do código foi submetida por um co-desenvolvedor, mas foi descoberto que este consertava um problema semelhante, diferente do que foi

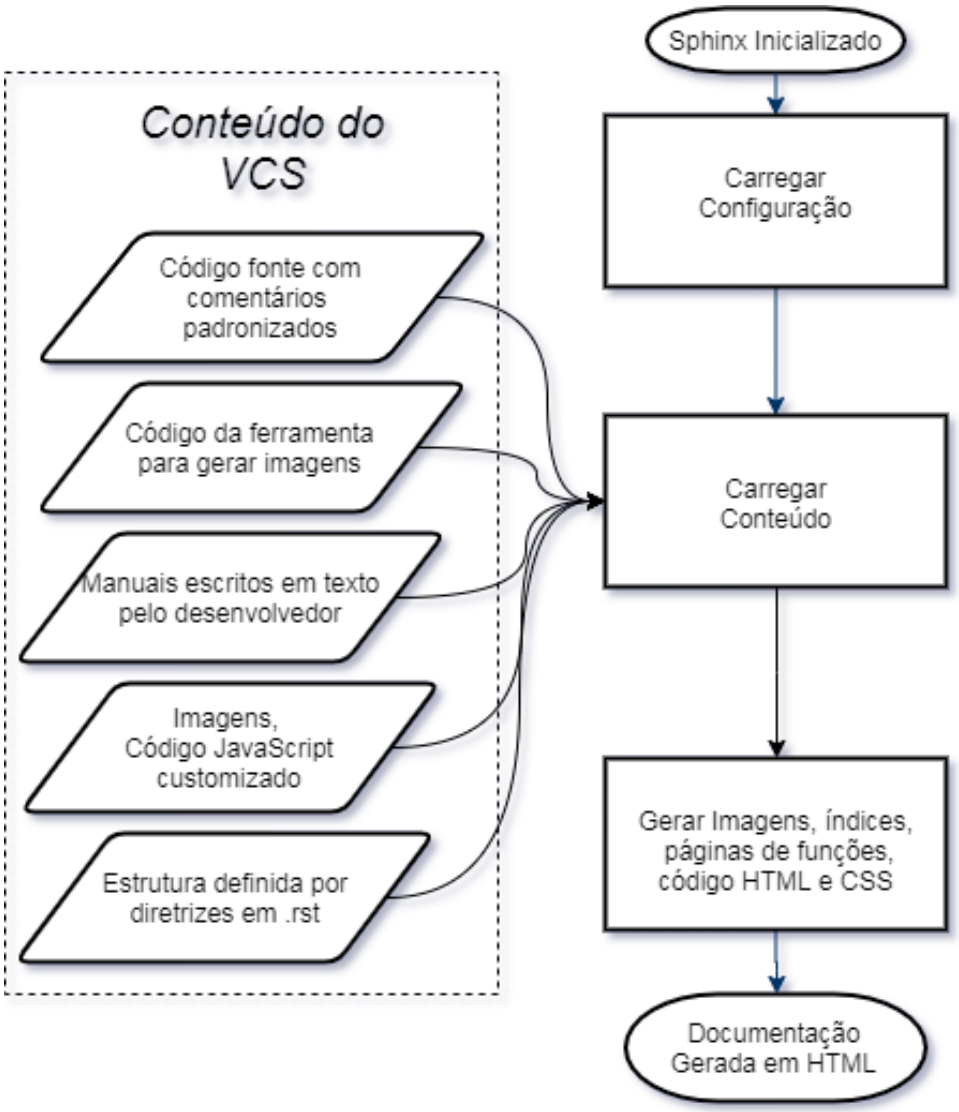
documentado, já que o anti-teste ainda estava com o mesmo comportamento. Isso permitiu que o co-desenvolvedor organizasse-se para continuar a correção, encontrando a solução definitiva. Assim, através de testes e integração contínua foi possível identificar e corrigir este e outros problemas encontrados durante o desenvolvimento do projeto

3.4.6 Gerando Documentação

Um dos propósitos do projeto não era apenas de inserir funcionalidades, mas documentar as existentes e futuras de maneira acessível e fácil de retrabalhar. Foi proposto pelo autor um gerador de documentação *Python* com o *software Sphinx*(BRANDL; TEAM, 2020). A configuração do gerador permitia que fossem geradas páginas web contendo comentários de código estruturados por um padrão e indexados para pesquisa. Além de páginas com conteúdo estático, o gerador acomodava página escritas manualmente para confecção de guias e tutoriais e imagens estáticas ou geradas pela execução da ferramenta sendo documentada.

O processo de coleta e execução do gerador de documentação *Sphinx* foi diagramado na Figura 24.

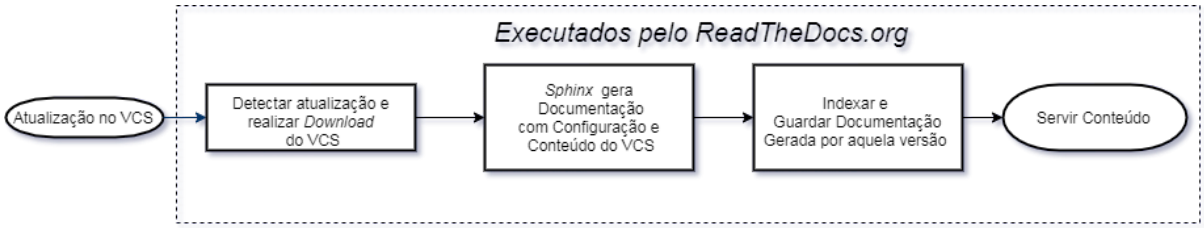
Figura 24 – Processo do Gerador de Documentação em *Python*, Sphinx



Fonte: Autor

Para manter a documentação atualizada e realizar distribuição, foi utilizado um serviço chamado *ReadTheDocs*(DOCS, 2020), que ligava-se ao sistema *VCS* organizado pelo autor e automaticamente gerava a documentação para o ramo principal e de desenvolvimento utilizando Sphinx. Este procedimento foi diagramado na Figura 25.

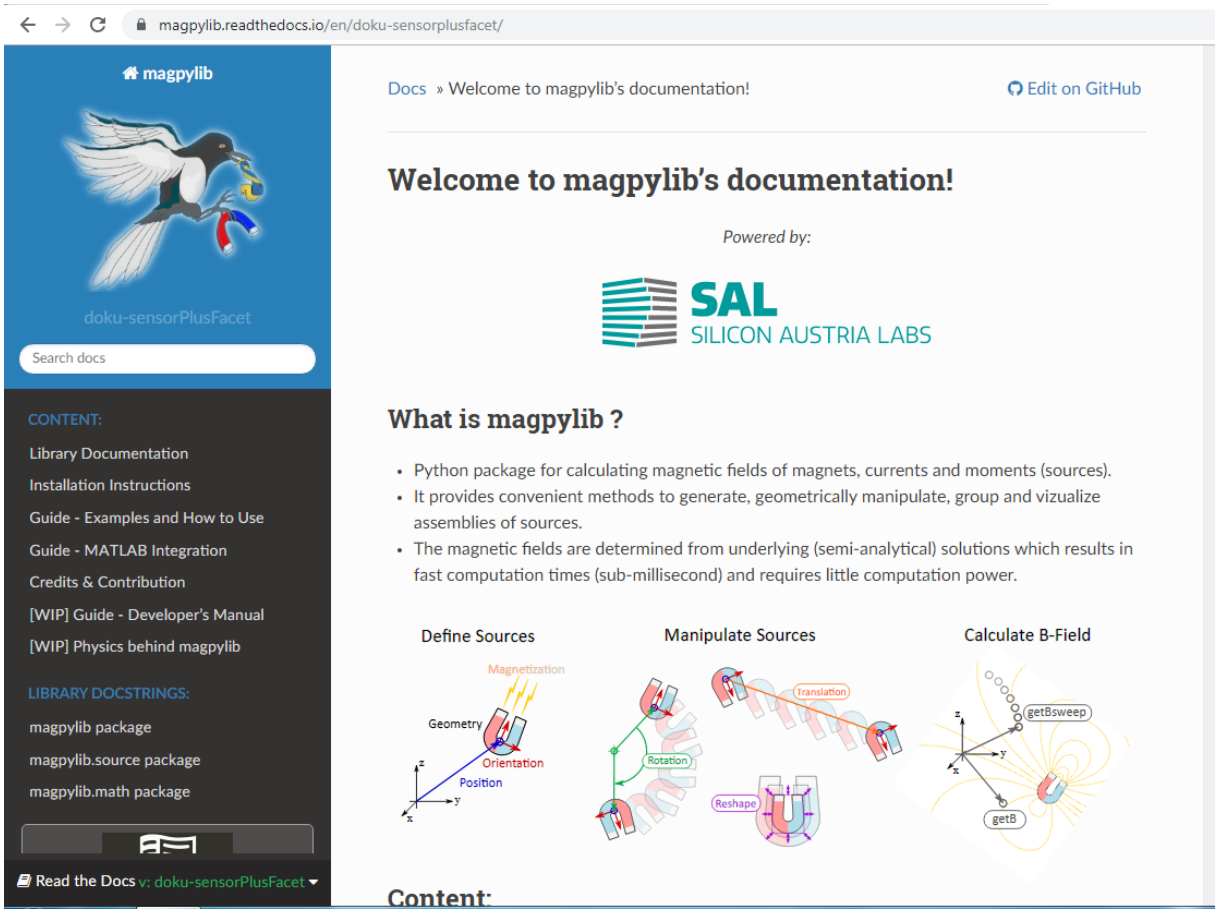
Figura 25 – Fluxo de atualização e entrega automática de Documentação.



Fonte: Autor

Inicialmente foi utilizada a versão privada do serviço *ReadTheDocs*, permitindo que a documentação fosse limitada apenas àqueles com acesso. A versão pública da página de documentação pode ser visualizada na Figura 26.

Figura 26 – Printscreen da documentação online gerada pelo Sphinx, servida pelo serviço ReadTheDocs.org .



Fonte: Autor

3.5 Do Licenciamento

Inicialmente, a ferramenta era de direitos reservados à empresa. Isto significava que a empresa tinha domínio e direito exclusivo ao uso e acesso ao código fonte da ferramenta. Isto fornecia os seguintes benefícios:

a) Todos os benefícios da ferramenta eram de uso exclusivo da empresa.

b) O uso da ferramenta fora da empresa seria dado somente sobre contrato.

c) Isso daria vantagem à empresa em condições de competição e em negociação de compra e venda do *software*.

Entretanto, isso também significava que:

a) A distribuição da ferramenta seria de responsabilidade da empresa;

b) Muitos recursos de desenvolvimento requerem contratos de serviço para manter sigilo, encarecendo a manutenção da ferramenta.;

c) Vender a ferramenta implicaria em necessidade de suporte, e sistema de verificação de licença.

Como a ferramenta era a implementação de informação encontrada em licenciatura e não possuia nenhum tipo de segredo, foi então proposto pela equipe de desenvolvimento que o *software* fosse licenciado como código aberto. Houve uma pesquisa interna sobre diversas licenças que poderiam ser usadas. A licença proposta foi *Affero GPL V3* (FSF, 2019), que fornece:

a) Retenção do *copyright* da empresa original.

b) Abertura obrigatória de códigos que utilizem o *software* ou partes dele.

c) Possibilidade de uso comercial.

d) Auxilio no desenvolvimento de novos recursos e funcionalidades por terceiros.

Estas condições significam que, caso uma empresa além da original adote o *software* em parte ou totalidade em algum projeto para venda, serão legalmente obrigados a fornecer o código fonte de todo projeto derivado, incluindo as partes originais. A Empresa ainda possui direito de criar uma versão da ferramenta para que possa ser vendido sem essa condição futuramente. Isso torna o *software* perfeito para utilização interna da própria empresa e também acadêmica, mas não para ambientes corporativos, que ainda pode ser negociado.

O código aberto também providenciou uma redução de custos no desenvolvimento, já que muitos recursos como *CircleCI*, *GitHub* e canais de distribuição *PyPi* são disponibilizados gratuitamente para projetos de *software* livre.

3.5.1 Distribuição do *Software*

Com o *software* desenvolvido, era necessário configurar um canal de distribuição para que usuários pudessem obter a ferramenta facilmente. Para instalar a ferramenta, o usuário poderia baixar a última versão do *VCS* e instalar a ferramenta com seu gerenciador de pacotes. Isso provou-se complicado e trabalhoso para usuários que não conhecem o sistema de versionamento, logo foi necessário pensar em disponibilizar a ferramenta em algum canal de downloads.

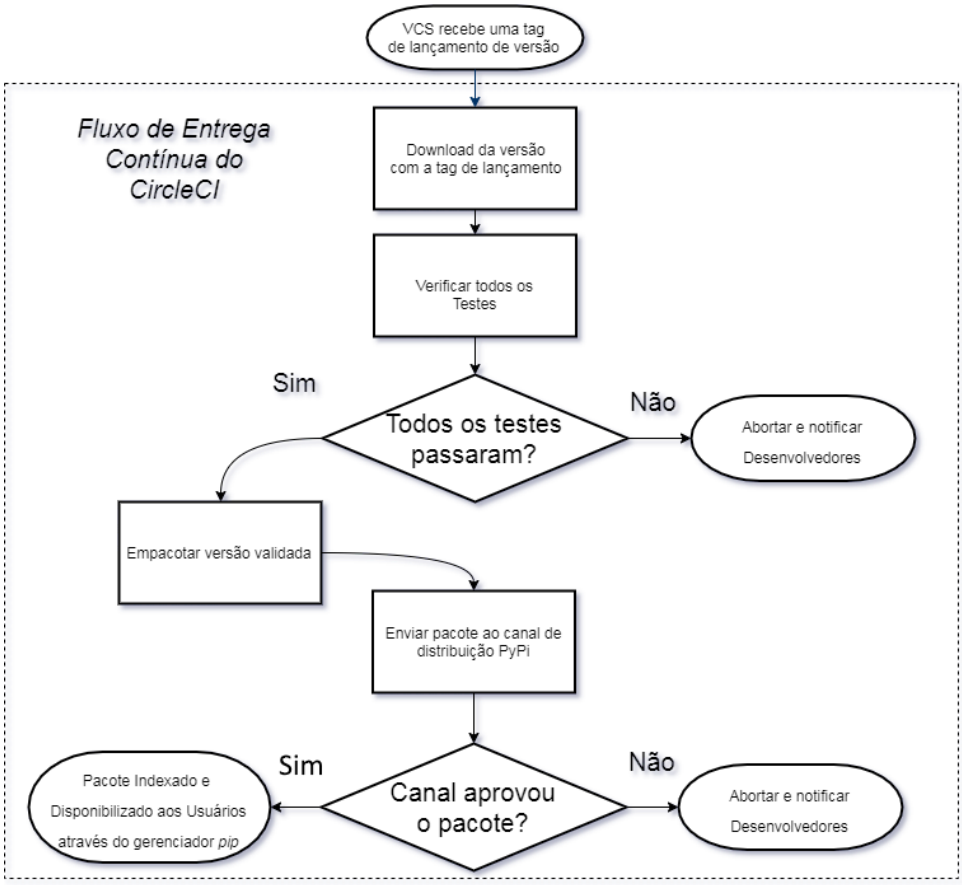
Felizmente, o módulo gerenciador de pacotes pip do *Python* não só permite que pacotes externos sejam instalados, mas também permite que pacotes sejam baixados e então instalados, providenciando apenas o nome do pacote, desde que o pacote esteja no canal de distribuição *PyPi* (*Python* Package Index). *PyPi* é o repositório oficial de pacotes para a linguagem de programação *Python*, fornecendo uma quantidade considerável de opções adicionais para usuários *Python* baixarem e utilizarem

rapidamente, além de todas as versões anteriores de cada pacote.

Para registrar um pacote no *PyPi*, é necessário acessar o website *pypi.org* e registrar uma conta de usuário mantenedor e o nome do pacote. Os pacotes devem seguir uma padronização do script de configuração setup.py para se enquadrar em seu sistema de categorização. A fonte do pacote é então empacotada da maneira proposta e enviado utilizando um comando *Python*, que é então disponibilizado como última versão no canal.

Como este processo definido para o projeto, percebeu-se que este deveria ser repetido para cada versão oficializada da ferramenta. Para evitar erros manuais, foi utilizado o conceito de Entrega Contínua, utilizando o serviço CircleCI. A implementação do sistema pode ser visualizada em formato de fluxograma na figura 27.

Figura 27 – Fluxo de Entrega Contínua do Projeto.



Fonte: Autor

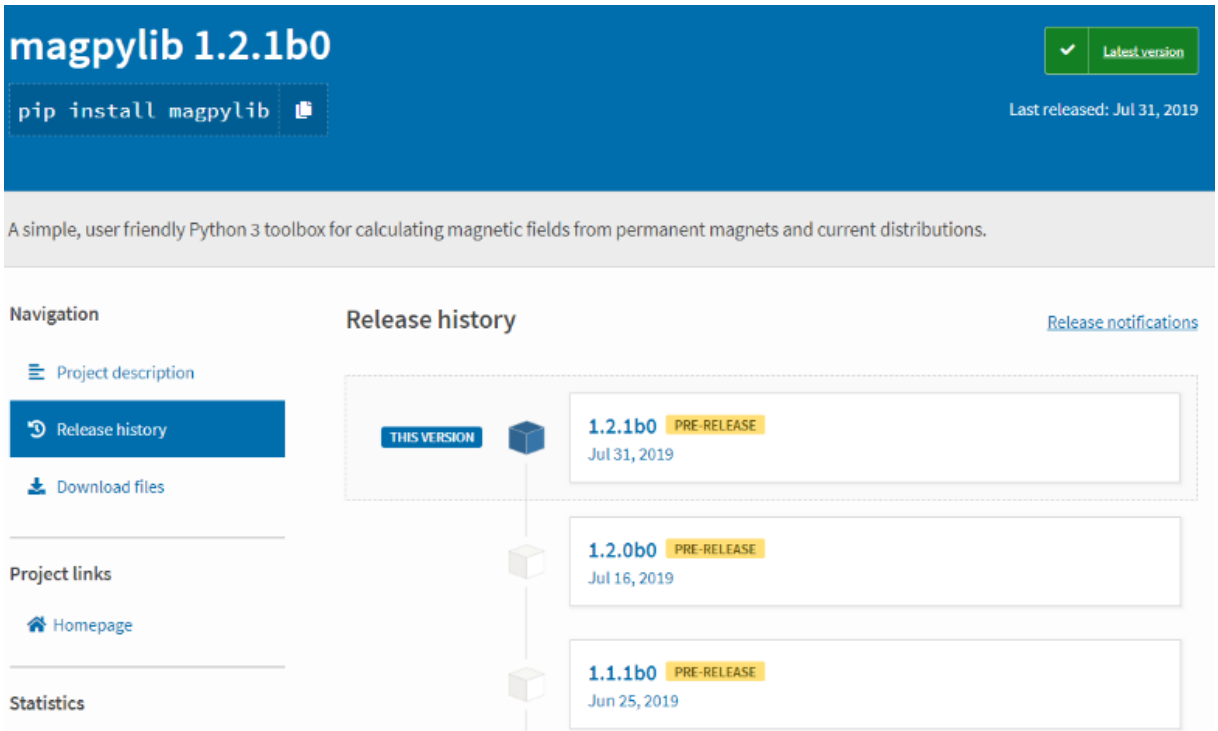
Através deste sistema, o serviço inicia uma nova rotina ao receber uma notificação de lançamento de versão do *VCS*. Os testes são executados novamente, e caso todos forem aprovados ocorrerá o empacotamento adequado do *software* como requisitado pelas instruções *PyPi*. Após isso, o pacote é enviado pelo sistema ao repositório oficial. O repositório responderá se a entrega foi bem sucedida, e o serviço de Entrega Contínua informará os Desenvolvedores envolvidos.

3.6 Resultados Finais

Com a implementação do *VCS* para desenvolvimento, sistemas de integração contínua, entrega contínua, e documentação automatizada, a ferramenta foi validada como estável pela equipe interna. A ferramenta foi então lançada como *software* aberto em estágio $1.0.0$ Beta em 29 de Maio de 2019, disponível para download gratuito no repositório oficial de pacotes *Python PyPi*. A página do projeto epode ser visualizada na Figura 28.

Novas versões foram lançadas com otimizações, recursos, e correções de pequenos *bugs*. As funcionalidades gráficas e aplicação do *software* receberam o prêmio de melhor apresentação na conferência de Sensoriamento Magnético “Magnetic Frontiers” em Lisboa, Portugal, no dia 27 de Junho de 2019 (CCSD, 2019).

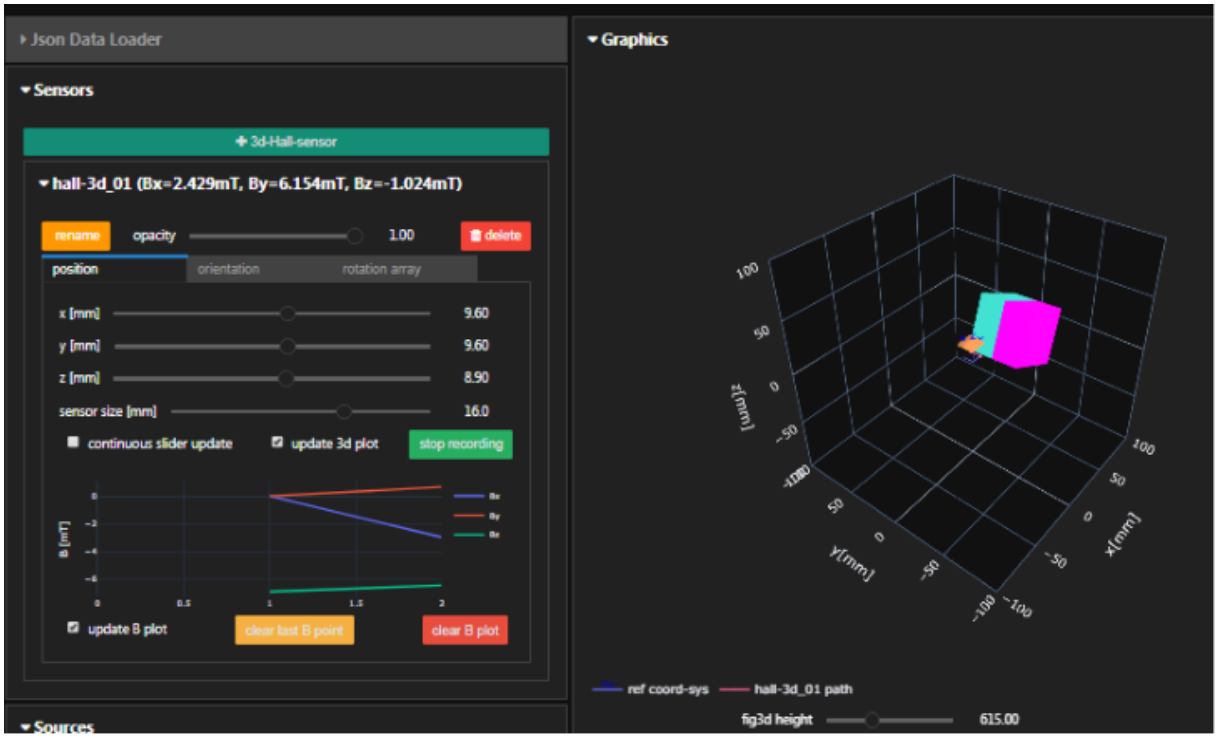
Figura 28 – Página de Versões da magpylib no repositório *PyPi* .



Fonte: Autor

Com a abertura do repositório remoto do *VCS* no *GitHub* e também da documentação, recursos da ferramenta foram estendidas pela comunidade. Usuários criaram extensões para utilizar o ambiente Jupyter Notebook e a biblioteca plotly para a realização do fluxo principal em tempo real da ferramenta com o auxílio de uma interface gráfica (BOISSELET, 2019). Esta interface pode ser visualizada na Figura 29.

Figura 29 – Interface gráfica alternativa realizada pela comunidade.



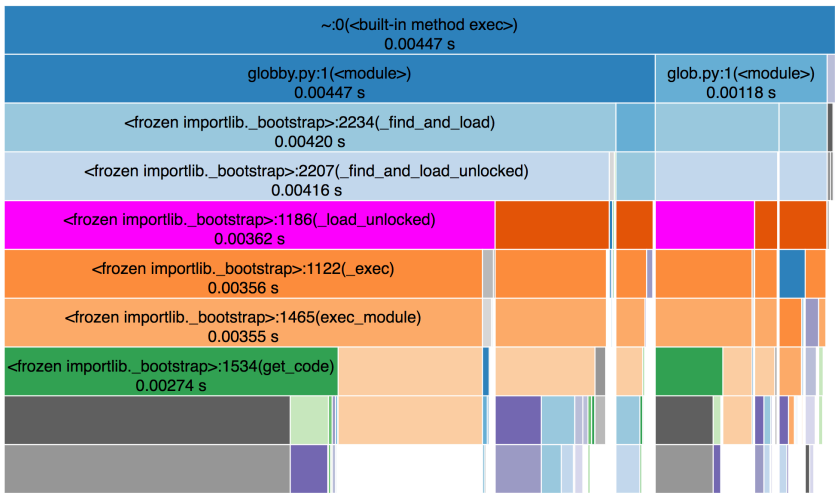
Fonte: (BOISSELET, 2019)

A ferramenta recebeu suporte oficial do autor até 31 de Julho de 2019, que a partir de então recebeu mais de 5000 *downloads* no total através dos servidores de distribuição, em 3 meses. (ANACONDA, 2019) (FLYNN, 2019)

3.6.1 Otimizações

Ao fim do projeto, foram realizadas análise de desempenho do *software*. Utilizando o módulo padrão do Python *cProfile* e o visualizador externo *SNAKEVIZ* (DAVIS, 2019), observou-se que a maior parte do tempo do *software* era desperdiçado por chamadas de funções supérfluas.

Figura 30 – Visualizador ICICLE do SNAKEVIZ.



Fonte: (DAVIS, 2019)

No visualizador representado pela Figura 30, funções são executadas em um fluxo de início ao fim, começando do topo. Funções chamadas por outras funções adicionam uma camada para baixo. A largura de cada bloco de função representa o tempo de execução. A abertura de qualquer função tem um tempo mínimo chamado *overhead*.

Foi interpretado que, devido ao *Python* 3 ter sua implementação oficial de maneira interpretada ao invés de compilada com o interpretador *CPython*, havia um grande desperdício de recurso ao utilizar funções e rotinas que se apresentavam como ferramentas de organização de código. Compiladores muitas vezes re-escrevem rotinas como código sequencial ao invés de ramificação para aumentar a performance ao evitar transições de contexto. Disto, foram removidas muitas destas funções de utilidade para diminuir a complexidade durante a execução, com um custo da organização do código fonte do projeto. Esta deficiência é muitas vezes contornada ao utilizar módulos criados com binários, compilados de código externo ao *Python*.

Com isto em mente, otimizações adicionais poderiam ser realizadas ao refatorar o código de modelos matemáticos em C ou C++. Estas e outras rotinas de cálculo poderiam ser transformadas em código compilado, transformado-os em módulos da ferramenta compreensíveis ao interpretador *CPython*. Estes módulos receberiam os parâmetros necessários, e retornariam os resultados esperados através da interface

existente em *Python*. Devido ao escopo de requisitos e tempo de execução do projeto, esta otimização não foi realizada.

4 CONCLUSÃO

Foi desenvolvida uma versão aprimorada e estável do *software magpylib* em código aberto para uso industrial ou acadêmico, e também foi desenvolvido um sistema de entrega de versões do programa em um canal de distribuição oficial de *Python*, *PyPi*. O sistema desenvolvido cumpriu os objetivos específicos organizacionais e de desenvolvimento ao incluir um controle de versão utilizando *git* e *GitHub*, detecção automática de defeitos e entrega contínua com o serviço *CircleCI*, além de geração e distribuição automática de documentação com *ReadTheDocs* e *Sphinx*.

O trabalho beneficiou não só os usuários da ferramenta como os futuros desenvolvedores, já que muito que foi introduzido é de cunho organizacional. Versões futuras do *software* podem ser facilmente concebidas usando *git*, e lançadas para os canais de distribuição. O licenciamento do *software* permite também que os gerentes da empresa possam tomar futuras decisões quanto ao suporte e uso comercial do produto.

Futuramente, os modelos de cálculos matemáticos da ferramenta podem ser otimizados para alcançar mais desempenho ao utilizar extensões *Python* feitas em C ou C++, ao invés de apenas *Python* com *numpy*. Como demonstrado pela comunidade, também pode ser de interesse a criação de uma interface gráfica para entrada e visualização de dados mais prática.

Foi proposta a possibilidade de transformar o software em um *app* de celular para facilitar a execução de medidas simples.

REFERÊNCIAS

AHO, A. V. **Compilers: Principles, Techniques, and Tools (2nd Edition)**. [S.l.]: Addison Wesley, 2006. ISBN 0321486811. Citado na página 20.

ANACONDA, I. **Anaconda Cloud - Conda Forge Magpylib**. 2019. <http://web.archive.org/web/20191203011433/https://anaconda.org/conda-forge/magpylib>. Online, Acessado: 2019-12-03. Citado na página 60.

BOISSELET, A. **A graphical user interface for the magpylib package in jupyter with ipywidgets**. 2019. <http://web.archive.org/web/20191204225420/https://github.com/Alexboiboi/MagpylibGui>. Online, Acessado: 2019-12-4. Citado na página 60.

BRANDL, G.; TEAM the S. **The Sphinx Documentation generator**. 2020. <http://web.archive.org/web/20200204200401/https://www.sphinx-doc.org/en/master/>. Acessado: 2020-03-13. Citado na página 52.

CALLISTER, W. D. **Fundamentals of Materials Science and Engineering: An Integrated Approach**. [S.l.]: Wiley, 2011. ISBN 1118061608. Citado 2 vezes nas páginas 27 e 28.

CCSD, C. pour la C. S. D. **P4-18 - Designing Airgap-Stable Magnetic Position Systems by Field Shaping**. 2019. <http://web.archive.org/web/20191202023958/https://mag-frontiers.sciencesconf.org/resource/page/id/20>. Online, Acessado: 2019-5-20. Citado na página 59.

CODECOV. **Code Coverage Done Right**. 2019. <https://codecov.io/>. Online, Acessado: 2019-12-02. Citado na página 48.

CONSERVANCY, S. F. **git scm**. 2019. <web.archive.org/web/20200212200058/https://git-scm.com/>. Online, Acessado: 2020-02-12. Citado na página 25.

DAVIS, M. **A browser based graphical viewer for the output of Python's cProfile module**. 2019. <http://web.archive.org/web/20190520124633/https://jiffyclub.github.io/snakeviz/>. Online, Acessado: 2019-05-20. Citado na página 61.

DOCS, I. Read the. **Technical documentation lives here**. 2020. <http://web.archive.org/web/20200204200401/https://readthedocs.org/>. Acessado: 2020-03-13. Citado na página 53.

FLYNN, C. **Magpylib Package Stats**. 2019. <https://pypistats.org/packages/magpylib>. Online, Acessado: 2019-11-02. Citado na página 60.

FSF, F. S. F. **GNU AFFERO GENERAL PUBLIC LICENSE**. 2019. <http://web.archive.org/web/20191128211149/https://www.gnu.org/licenses/agpl-3.0.en.html>. Online, Acessado: 2019-11-28. Citado na página 56.

GitHub. **No License**. 2019. <http://web.archive.org/web/20190513152912/https://choosealicense.com/no-permission/>. Online, Acessado 2019-05-13. Citado na página 26.

HUIZINGA, D. **Automated Defect Prevention: Best Practices in Software Management**. [S.l.]: Wiley-IEEE Computer Society Pr, 2007. ISBN 0470042125. Citado 3 vezes nas páginas 25, 32 e 48.

INC, A. **Anaconda Python/R Data Science Platform**. 2019. <http://web.archive.org/web/20191127031914/https://www.anaconda.com/>. Online, Acessado: 2019-11-27. Citado na página 35.

INC, A. **Conda Package Manager**. 2019. <http://web.archive.org/web/20191127031914/http://web.archive.org/web/20200212153533/https://conda.io/en/latest/>. Online, Acessado: 2020-02-12. Citado na página 23.

INC., C. I. S. **CircleCI: Continuous Integration and Delivery**. 2020. <http://web.archive.org/web/20200204200401/https://circleci.com/>. Acessado: 2020-03-13. Citado na página 48.

JACKSON, J. D. **Classical Electrodynamics, 2nd Edition**. [S.l.]: Wiley, 1975. ISBN 047143132X. Citado na página 43.

John Hunter. **a Python 2D plotting library**. 2019. <http://web.archive.org/web/20191128062349/https://matplotlib.org/>. Online, Acessado 2019-11-28. Citado na página 15.

John S. Loomis. **Assembly Language Programming**. 2019. <http://web.archive.org/web/20190430002625/http://www.johnloomis.org/ece314/notes/carch/node7.html>. Online, Acessado 2019-04-30. Citado na página 19.

LUBUNTU, C. **Lubuntu Linux System**. 2019. <https://web.archive.org/web/20191004021135/https://lubuntu.net/>. Acessado: 2019-10-04. Citado na página 34.

MARTINS, L. P. Tcc. In: **JOYSTICK MAGNÉTICO DE DOIS EIXOS E SOFTWARE DE DEMONSTRAÇÃO DE FUNCIONAMENTO**. [S.l.: s.n.], 2019. v. 1. Citado na página 15.

MICROSOFT. **Open-source source-code editor**. 2019. <http://web.archive.org/web/20191128024253/https://code.visualstudio.com/>. Acessado: 2019-11-28. Citado na página 35.

NUMPY, C. **Numpydoc docstring guide**. 2019. <http://web.archive.org/web/20191203011433/https://numpydoc.readthedocs.io/en/latest/format.html>. Online, Acessado: 2019-10-07. Citado na página 41.

Numpy Developers. **The fundamental package for scientific computing with Python**. 2019. <http://web.archive.org/web/20191203003917/https://www.numpy.org/>. Online, Acessado 2019-05-03. Citado 2 vezes nas páginas 15 e 22.

OPENSOURCE.ORG. **Licenses and Standards**. 2020. <http://web.archive.org/web/20200212031838/https://opensource.org/licenses>. Online, Acessado: 2020-1-12. Citado 2 vezes nas páginas 10 e 27.

ORACLE. **Free and open-source hosted hypervisor for x86 virtualization**. 2019. <http://web.archive.org/web/20191119233313/https://www.virtualbox.org/>. Acessado: 2019-11-19. Citado na página 34.

ROY, P. V. **Concepts, Techniques, and Models of Computer Programming (The MIT Press)**. [S.l.]: The MIT Press, 2004. ISBN 0262220695. Citado na página 21.

SILVA, A. H. da. Tcc. In: **IMPLEMENTAÇÃO DE UM JOYSTICK MAGNÉTICO DE TRÊS EIXOS E SOFTWARE DE DEMONSTRAÇÃO GRÁFICA**. [S.l.: s.n.], 2018. p. 1–55. Citado 2 vezes nas páginas 15 e 29.

SOMMERVILLE, I. **Software Engineering: (Update) (8th Edition)**. [S.l.]: Addison Wesley, 2006. ISBN 9780321313799. Citado na página 33.

SPYDER, C. **The Scientific Python Development Environment**. 2018. <http://web.archive.org/web/20190803110320/https://www.spyder-ide.org/>. Acessado: 2019-08-03. Citado na página 35.

Wikipedia contributors. **Computer programming in the punched card era — Wikipedia, The Free Encyclopedia**. 2019. <https://en.wikipedia.org/w/index.php?title=Computer_programming_in_the_punched_card_era&oldid=924989260>. [Online, Acessado 2019-11-03]. Citado na página 18.

ZOLKIFLI, N. N.; NGAH, A.; DERAMAN, A. Version control system: A review. **Procedia Computer Science**, Elsevier, v. 135, p. 408–415, 2018. Citado na página 24.