UNIVERSIDADE FEDERAL DE JATAÍ (UFJ)
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS (ICET)
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Ryan Francys Mendes dos Santos

Projeto e Implementação da Biblioteca Padrão de uma Linguagem de Programação Específica do Domínio da Robótica: Funções Matemáticas

Jataí-Goiás

2024

Ryan Francys Mendes dos Santos

Projeto e Implementação da Biblioteca Padrão de uma Linguagem de Programação Específica do Domínio da Robótica: Funções Matemáticas

Monografia apresentada ao curso de Ciência da Computação do Instituto de Ciências Exatas e Tecnológicas da Universidade Federal de Jataí (UFJ), como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Prof. Dr. Thiago Borges de Oliveira

Jataí-Goiás

2024

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFJ.

Santos, Ryan Francys Mendes
Projeto e Implementação da Biblioteca Padrão de uma Linguagem de Programação Específica do Domínio da Robótica: Funções Matemáticas / Ryan Francys Mendes Santos. - 2024.
XLVIII, 48 f.: il.

Orientador: Prof. Dr. Thiago Borges Oliveira.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Jataí, Instituto de Ciências Exatas e Tecnológicas, Ciência da Computação, Jataí, 2024.
Bibliografia.
Inclui siglas, abreviaturas, símbolos, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. Linguagem de programação. 2. Cordic. 3. Biblioteca padrão. 4. Robcmp. I. Oliveira, Thiago Borges, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE JATAÍ
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

DECLARAÇÃO DE APROVAÇÃO DA VERSÃO FINAL

Declaro que o(a) discente Ryan Francys Mendes dos Santos curso de Bacharelado em Ciência da Computação Trabalho de Conclusão de Curso (TCC) com o título final Projeto e Implementação da Biblioteca Padrão de uma Linguagem de Programação Específica do Domínio da Robótica: Funções Matemáticas na data de 11/12/2024 e efetuou todas as correções pertinentes sugeridas pela banca examinadora, composta pelo seguintes membros:

Orientador(a)	Thiago Borges de Oliveira
Membro 1	Ariadne de Andrade Costa
Membro 2	Franciny Medeiros Barreto

Declaro ainda que a versão final anexada a este processo está adequada para ser devidamente depositada em repositório institucional.

Observação

Esta declaração deve ser assinada pelo(a) orientador(a)

Documento assinado eletronicamente por **THIAGO BORGES DE OLIVEIRA**, **Professor do Magistério Superior**, em 18/12/2024, às 19:55, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufj.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0372169** e o código CRC **68F75B0C**.

Dedico este trabalho a todos que acreditaram e me apoiaram nesta jornada. À minha família, que sempre me apoiou e me sustentou nos momentos mais difíceis. E ao meu professor e orientador, Thiago Borges de Oliveira, que sempre me incentivou.

Agradecimentos

Gostaria de expressar minha sincera gratidão a todos que contribuíram de alguma forma para a realização deste trabalho. Agradeço à minha família pelo apoio incondicional, pela paciência e pelos sacrifícios ao longo desta jornada. Ao meu orientador, Thiago Borges de Oliveira, por sua orientação, incentivo e valiosas contribuições. Agradeço também aos meus amigos e colegas, que estiveram ao meu lado, oferecendo apoio e motivação. Sem cada um de vocês, este trabalho não seria possível. Muito obrigado!

'Quando se quer realmente uma coisa, todo o universo conspira para que você realize o seu desejo'.

(Paulo Coelho)

Resumo

As bibliotecas padrão são um conjunto de bibliotecas predefinidas que acompanham a maioria das linguagens de programação, fornecendo funcionalidades essenciais para facilitar o desenvolvimento de software. Elas oferecem implementações de funções e rotinas que atendem a necessidades comuns, evitando que os desenvolvedores precisem reescrever código básico e complexo, o que aumenta a produtividade e a confiabilidade dos programas. Dentro das bibliotecas padrão, uma das mais utilizadas é a biblioteca matemática, que contém funções essenciais para realizar cálculos, além de oferecer operações como trigonometria, exponenciação, logaritmos, entre outras, frequentemente utilizadas em diversas áreas da ciência. Neste trabalho, investigou-se alternativas para a implementação de funções matemáticas na Robotics Language, desenvolvida na Universidade Federal de Jataí. Foram implementadas e avaliadas seis funções matemáticas: seno, cosseno, tangente, exponencial, logaritmo e raiz quadrada, utilizando algoritmos alternativos, como o CORDIC para as operações trigonométricas. Avaliou-se o tempo de execução e o tamanho do programa objeto para as plataformas AVR e STM32. Os resultados iniciais indicaram que os algoritmos são funcionalmente adequados e precisos, mas não competitivos em termos de tempo de execução e tamanho do programa objeto gerado, considerando o estágio atual de desenvolvimento do Robcmp e dos backends AVR e ARM do LLVM.

Palavras-chaves: *Linguagem de programação; CORDIC; Biblioteca padrão; Robcmp.*

Abstract

The standard libraries are a set of predefined libraries that come with most programming languages, providing essential functionalities to facilitate software development. They offer implementations of functions and routines that address common needs, preventing developers from having to rewrite basic and complex code, thus increasing productivity and program reliability. Among the standard libraries, one of the most commonly used is the math library, which contains essential functions for performing calculations, in addition to offering operations such as trigonometry, exponentiation, logarithms, and others, frequently used in various areas of science. This work investigated alternatives for implementing mathematical functions in the Robotics Language, developed at the Federal University of Jataí. Six mathematical functions were implemented and evaluated: sine, cosine, tangent, exponential, logarithm, and square root, using alternative algorithms such as CORDIC for trigonometric operations. The execution time and object code size were evaluated for the AVR and STM32 platforms. The initial results indicated that the algorithms are functionally adequate and precise, but not competitive in terms of execution time and object code size, considering the current development stage of Robcmp and the AVR and ARM backends of LLVM.

Key-words: *Programming language; CORDIC; Standard library; Robcmp.*

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

RNL	Revisão Narrativa de Literatura
BCC	Bacharelado em Ciência da Computação
UFJ	Universidade Federal de Jataí
MCU	Microcontroller Unit
SO	Sistema Operacional
GCC	Gnu Compiler Collection

Sumário

1 Introdução

1.1 Motivação

A automatização de processos com robôs está cada vez mais sendo almejada, tanto por pessoas para melhorar o rendimento do seu próprio dia quanto na indústria para automatizar processos ou serviços que antes eram manufaturados. Hoje em dia, no ramo da saúde, até mesmo cirurgias são executadas por robôs assistidos (SANTANA et al., 2022). Porém, cada vez é mais difícil realizar a análise, o design, a implementação e o teste de *software* em sistemas integrados (AKDUR; GAROUSI; DEMIRÖRS, 2018), devido a junção de vários tipos distintos de componentes de *software* e *hardware*, que tornam únicas a comunicação entre os componentes e, consequentemente, a forma de os programar.

Parte central dos sistemas integrados, ou embarcados, são os microcontroladores (ou MCUs). Eles são usados em vários dispositivos da nossa vida, como: automóveis, receptores de TV, câmeras, impressoras, brinquedos e robôs (HUSSAIN et al., 2016). Uma particularidade relevante é o fato de possuírem uma menor memória e poder de processamento, comparados aos processadores de computadores portáteis ou de mesa. Por exemplo, microcontroladores da família STM32 (MICROELECTRONICS, 2023), frequentemente usados na indústria, possuem entre 2 e 620 KB de memória SRAM e frequência máxima de 600 Mhz. Por outro lado, as principais vantagens de seu uso são o baixo custo e a relativa simplicidade de programação, comparado aos microprocessadores, principalmente por não usarem sistemas operacionais e possuírem memória de armazenamento, memória de trabalho e CPU num só *chip* (HUSSAIN et al., 2016).

Cada modelo ou família de microcontrolador possui um conjunto próprio de registradores, periféricos e instruções da CPU. Para facilitar o uso de seus microcontroladores, cada fabricante disponibiliza uma biblioteca de rotinas e funções, geralmente nomeada como *Hardware Abstraction Layer* (HAL), escrita em uma linguagem de propósito geral, frequentemente a linguagem C. Devido à ausência de padrões e a própria heterogeneidade de periféricos, HALs de fabricantes distintos não são compatíveis entre si e é necessário, a cada modelo de MCU incluído em um projeto de sistema embarcado, que os programadores aprendam a respectiva interface de programação.

Com o objetivo de simplificar a programação dos microcontroladores, *frameworks* e bibliotecas para linguagens de programação de propósito geral foram propostos ao longo do tempo, provendo um conjunto de funções em *software* que fornecem uma camada adicional sobre o HAL do fabricante. Um exemplo em particular é o *framework* Arduino (ARDUINO, 2024), que fornece abstração a nível de periféricos – somente para os periféricos

mais comuns, para as plataformas AVR[1], STM32[2], ExpressIf32[3], dentre outras. Outros incluem o CMSIS[4], MBED[5], STM32CUBE[6] e LibOpenCM3[7], porém estes específicos para arquitetura ARM.

Portanto, o suporte para a programação dos microcontroladores é fornecido por várias peças de *software*: *i*) o suporte fundamental da biblioteca padrão da linguagem de programação escolhida, que inclui funções matemáticas, de conversão de tipos, ordenação, dentre outros (FOWLER; KORN; VO, 1995), *ii*) o suporte do HAL do fabricante que permite o acesso aos periféricos do microcontrolador, e *iii*) o suporte de algum *framework* que isole a heterogeneidade de periféricos e facilite a programação de MCUs distintas.

Não obstante, ainda é comum que projetos de *firmware* relevantes implementem suas próprias camadas de abstração de *hardware*, como foi o caso do projeto Marlin, um dos *firmwares* para impressoras 3D mais utilizado no mundo (Marlin Firmware, 2024). De acordo com os próprios desenvolvedores, a escolha se deve principalmente a necessidade de melhor controle do *hardware* para desempenho, acesso a periféricos específicos e economia de energia (Marlin Firmware, 2024).

Nesta direção, o projeto Robotics Language: Uma Linguagem de Programação de Propósito Específico para Microcontroladores, PI05974-2024 – e seu predecessor, Especificação e Construção de Protótipos Funcionais de Kits Robóticos de Baixo Custo para uso em Processos de Ensino-Aprendizagem (PI02361-2018), atuam no desenvolvimento de uma linguagem específica para o domínio da robótica e de microcontroladores, denominada The Robotics Language, e de seu compilador, Robcmp, na Universidade Federal de Jataí.

O objetivo da Robotics Language é isolar as especificidades dos microcontroladores, provendo uma camada de abstração de *hardware* dentro do próprio compilador e em sua biblioteca padrão, ao invés de depender de *frameworks* ou bibliotecas que implementem tal camada. Desta forma, os desenvolvedores podem escrever uma só vez o código de suas aplicações (o *firmware* para projetos de automação, equipamentos de tecnologia embarcada, dentre outros) sem a necessidade de desenvolver, como parte da própria aplicação, um HAL específico, ou ter partes condicionadas para cada um dos equipamentos distintos que a aplicação suporta. Uma vantagem que emerge deste suporte, a nível de compilador, é que as palavras reservadas e mecanismos específicos do domínio dos microcontroladores e robótica permitem que a análise semântica do compilador aponte erros de programação já durante a compilação, os quais, em linguagem de propósito geral, só poderiam ser encontrados em tempo de execução, durante o teste manual da aplicação no *hardware*

[1] https://registry.platformio.org/tools/platformio/framework-arduino-avr/compatibility
[2] https://registry.platformio.org/tools/platformio/framework-arduinoststm32
[3] https://registry.platformio.org/platforms/platformio/espressif32
[4] https://www.arm.com/technologies/cmsis
[5] https://registry.platformio.org/tools/platformio/framework-mbed
[6] https://registry.platformio.org/tools/platformio/framework-Stm32cube
[7] https://registry.platformio.org/tools/platformio/framework-libopencm3

embarcado (OLIVEIRA, 2024).

1.2 Objetivo do Trabalho

Dentre os vários desafios da implementação de uma linguagem de programação, e especificamente a Robotics Language, teve-se como objetivo, neste trabalho, projetar e implementar um conjunto de funções matemáticas e trigonométricas, na biblioteca padrão da linguagem de programação pertencente ao compilador Robcmp. Os objetivos específicos foram:

- Escolher os algoritmos mais adequados ao domínio e implementar as funções específicas para operações matemáticas (trigonométricas, radiciação, exponenciação e logaritmo); e
- Elaborar e executar testes unitários e experimentos para confirmar o funcionamento e desempenho da implementação.

1.3 Contribuição do Trabalho

A contribuição deste trabalho está na criação de uma biblioteca de funções matemáticas (trigonométricas, exponenciais, radiciais e logarítmicas), que proporcionam ao usuário a simplicidade presente em linguagens de propósito específico, além de permitir a expressividade e um desempenho próximo de uma linguagem de propósito geral.

2 Referencial Teórico

2.1 Introdução

Para a compreensão desse projeto de pesquisa, são apresentadas neste capítulo as definições utilizadas no decorrer do texto.

2.2 Compilador

Os compiladores estão cada vez mais presentes conforme a tecnologia avança, pois são eles que transformam sequências de instruções de alto nível, para uma linguagem na qual o processador consiga trabalhar (AHO RAVI SETHI, 2007). Conforme novas instruções são criadas em microcontroladores e CPUs, os compiladores devem se adaptar para utilizar essas novas instruções, a fim de melhorar seu desempenho e diminuir o tempo de execução das operações do software.

O compilador é um programa que recebe um programa fonte escrito em uma linguagem de programação e o traduz em um programa equivalente em outra linguagem, avisando o usuário e interrompendo a compilação quando erros são encontrados durante o processo de tradução (AHO RAVI SETHI, 2007). Um compilador é dividido internamente em fases, as quais contribuem para a compilação (FOLEISS et al., 2009). A Figura 1 exibe a sequência macro de passos de um compilador. Após receber um programa fonte, o compilador o passa por dois componentes principais: *i*) o *front end*, que realiza as etapas de análise lexica, sintática e semântica, produz uma árvore sintática e a transforma em código intermediário, e *ii*) o *back end*, onde serão realizadas etapas como otimização e geração de código de máquina, retornando um programa objeto e terminando o processo de compilação.

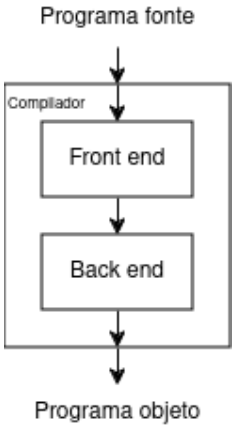


Figura 1 – Arquitetura macro de um compilador - adaptado de (AHO RAVI SETHI, 2007).

2.3 Front end

O *front end* de um compilador é a parte que converte código fonte em uma representação intermediária antes da geração do código alvo (LOPES; AULER, 2014). O código fonte passa consecutivamente pelas fases de análise léxica, sintática e semântica, transformando o programa fonte de uma representação para a outra (AHO RAVI SETHI, 2007). Na Figura 2 vemos as transformações no *front end*. Após receber uma entrada formada por uma sequência de caracteres (programa fonte), a analise léxica a transforma em uma sequência de *tokens*, que é entregue para a análise sintática, que por sua vez a transforma em uma árvore sintática, ou AST. A árvore sintática é passada para a análise semântica onde ocorrerão vários tipos de verificações por meio de visitação aos nós da árvore. Após a verificação semântica não detectar nenhum erro, a árvore sintática é repassada para o gerador de código intermediário a transformar em uma linguagem intermediária conhecida pelo *back end*.

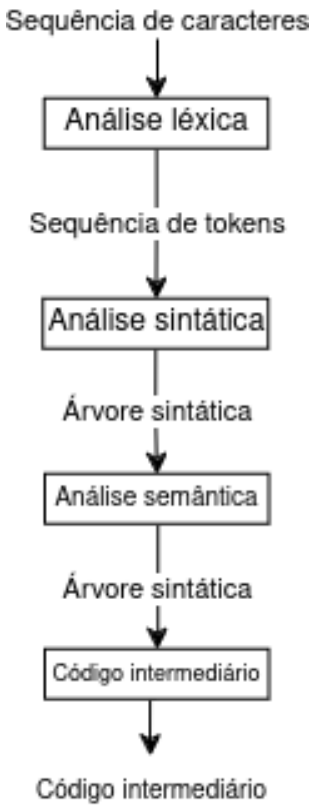


Figura 2 – Etapas do *front end* de um compilador - adaptado de (LOPES; AULER, 2014).

2.3.1 Analisador Léxico

Logo após o compilador receber uma entrada, o analisador léxico entra em ação lendo a sequência de caracteres, identificando e associando cada palavra com seus devidos

tokens, que serão utilizados na etapa de análise sintática (RICARTE, 2008). Também é possível realizar iteração com o usuário retornando mensagens de erros junto ao número de linha cujo erro foi encontrado (AHO RAVI SETHI, 2007).

O analisador léxico é subordinado do analisador sintático, entrando em ação quando lhe é requisitado um próximo *token* (FOLEISS et al., 2009), ou seja, o analisador léxico converte a sequência de caracteres em sequência de *tokens* aos poucos, em sintonia com o analisador sintático, de forma a reduzir o uso de memória durante o processo de compilação.

Para que os *tokens* sejam reconhecidos, utiliza-se uma linguagem regular (FEDOZZI, 2018). De acordo com Costa et al. (2010), uma linguagem regular é uma linguagem reconhecida por um autômato finito determinístico (AFD) e possui uma quantidade finita de estados. Ricarte (2008) define um automato finito como sendo uma máquina de estados finitos, que reconhece entradas pertencentes a linguagem regular, formada por uma quíntupla do tipo (K, Σ, δ, S, F), onde K é o conjunto de estados, Σ é o alfabeto de entrada finito, δ é o conjunto de transições, S representa o estado inicial e F o conjunto de estados finais.

Nos compiladores, frequentemente os analisadores léxicos são construídos usando expressões regulares, que são formas reduzidas de se especificar autômatos finitos não-determinísticos (AFN). Apesar de distinto, o AFN é um formalismo equivalente a um AFD (RICARTE, 2008). As expressões regulares possuem operadores e símbolos básicos que representam classes. Costa et al. (2010) os separa em classes de caracteres, quantificadores e operações de união e concatenação. O operador ponto (.) representa qualquer caractere. Colchetes ([,]) definem uma classe de caracteres. Acento circunflexo (^) nega uma classe de caracteres. Barra vertical (|) representa a união. Asterisco (*) indica que a cadeia pode ocorrer zero ou mais vezes. O sinal de mais (+) indica que a cadeia pode ocorrer uma ou mais vezes e por fim a interrogação (?) indica que uma cadeia pode ocorrer uma ou nenhuma vez. Portanto, uma expressão regular do tipo $[0\text{-}9]^+$ reconhece quaisquer números naturais diferentes de {}. Tal expressão pode ser utilizada para definir um *token* que reconhece um número do tipo inteiro pelo analisador léxico de uma linguagem de programação.

2.3.2 Analisador Sintático

O analisador sintático obtém uma cadeia de *tokens* geradas pelo analisador léxico, onde será verificada a sintaxe gerada e retornados os erros sintáticos encontrados (AHO RAVI SETHI, 2007). Com esses *tokens* é gerada a árvore sintática, que será utilizada e incrementada pelas demais fases da compilação (FOLEISS et al., 2009).

De acordo com Ricarte (2008), para construir a análise sintática de linguagens de

programação, utiliza-se uma gramática livre de contexto. Uma gramática livre de contexto é formada por um conjunto de regras de transformação, as quais contém terminais (*tokens*) e variáveis (ou não terminais). As regras são construídas de forma que uma variável pode ser transformada em outras variáveis e terminais (COSTA et al., 2010).

Para auxiliar as próximas etapas do compilador é criada uma estrutura auxiliar, nomeada de tabela de símbolos. De acordo com Ricarte (2008), a tabela de símbolos é uma estrutura criada para auxiliar as atividades de análise semântica do código. Nela são mantidas informações sobre as variáveis como: escopo de variáveis, variáveis declaradas e tipo de dado (int, string, float, ponteiro). Não há uma padronização para criar a tabela de símbolos, de modo que cada projetista adota sua própria maneira, dificultando a integração de módulos de diferentes compiladores.

Uma das dificuldades de se construir uma gramática é a garantia que ela não gere uma árvore sintática abstrata (AST) de derivação ambígua. Uma gramática é considerada ambígua quando a partir de uma mesma cadeia (fragmento do código fonte), seja possível obter mais de uma árvore de derivação. A Figura 3 exemplifica uma AST ambígua, onde para a expressão 5+3/2 pode ser construída mais de uma forma. O código gerado a partir de cada árvore teria resultados distintos, e o compilador não saberia qual a árvore correta a ser construída.

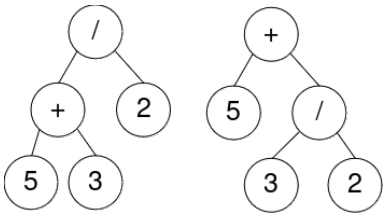


Figura 3 – Exemplo de árvore ambígua de derivação para a expressão 5+3/2.

2.3.3 Analisador Semântico

Segundo Costa et al. (2023), o analisador semântico é responsável por verificar erros semânticos, utilizando a tabela de símbolos para auxilio durante a análise, com a intenção de verificar se a AST gerada pela fase sintática faz sentido de acordo com a semântica da linguagem e assim gerar um programa seguro, sem erros e eficiente. Para realizar tal tarefa, de acordo com Foleiss et al. (2009), ocorre a visita em cada nó da AST, conferindo se sua estrutura é coerente e viável para a execução. Algumas das verificações mais comuns são: verificação de compatibilidade de parâmetros, compatibilidade de tipos em instruções de atribuição, determinação de constantes e verificação de declaração antes do uso de variáveis e funções.

Na Figura 4 temos um exemplo de árvore sintática contendo um erro semântico,

onde a árvore possui a expressão 51*3/"a". O analisador semântico deve detectar que os operandos são incompatíveis e gerar uma mensagem de erro sobre a operação, pois uma *string* não pode ser utilizada como operando de uma divisão.

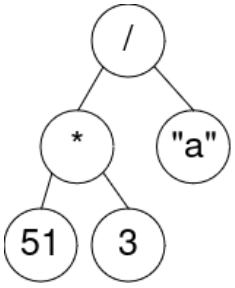


Figura 4 – Exemplo de erro semântico para a expressão 51*3/“a”

2.3.4 Geração de código intermediário

A geração de código possui o papel de gerar um código intermediário a partir da árvore sintática e da tabela de símbolos, cujo resultado seja próximo do código-alvo (FOLEISS et al., 2009). O código intermediário é utilizado como uma forma de se comunicar com o *back end* e é independente de arquitetura. Desta forma, um compilador pode emitir código-alvo para várias arquiteturas (FOLEISS et al., 2009). Esse suporte é implementado no *back end*, que de fato sabe como traduzir da linguagem intermediária para o código-alvo em cada arquitetura. O código-alvo pode assumir várias formas, como: linguagem de máquina, linguagem realocável de máquina ou linguagem de montagem (AHO RAVI SETHI, 2007).

No Robcmp, a saída da geração de código é um programa escrito em LLVM IR (LLVM Foundation, 2024). A LLVM IR é a linguagem suportada pelo conjunto de ferramentas do LLVM Project (LLVM Foundation, 2024), as quais também são usadas no Robcmp para implementar as demais etapas da compilação, como a geração de código-alvo.

A Figura 5 apresenta um fragmento do código LLVM IR gerado pelo Robcmp sobre o programa fonte da Figura 10 que se trata de uma implementação do algoritmo de ordenação *bubble sort* com verificação se os números realmente foram ordenados no final. No fragmento de código do LLVM IR podemos ver algumas marcações (*labels*) com nomes *while_cond5*, *while_body6*, *if_then* e *if_cont*, e operações como *load* e *add*. A LLVM IR é uma linguagem simples, do ponto de vista que ela não suporta orientação a objetos, nem outras características únicas das linguagens modernas. O código orientado a objeto precisa ser transformado em código procedural. Esse é o maior trabalho dos *front ends* hoje em dia.

```
while_cond5:                                      ; preds = %if_cont, %while_body
  %s.0 = phi i16 [ %binop4, %while_body ], [ %binop35, %if_cont ]
  %sext10 = sext i16 %s.0 to i32
  %cmpi11 = icmp slt i32 %sext10, %1
  br i1 %cmpi11, label %while_body6, label %while_cond

while_body6:                                      ; preds = %while_cond5
  %a13 = load i16, ptr %gep, align 2
  %gep16 = getelementptr [0 x i16], ptr %0, i16 0, i16 %s.0
  %a17 = load i16, ptr %gep16, align 2
  %cmpi18 = icmp sgt i16 %a13, %a17
  br i1 %cmpi18, label %if_then, label %if_cont

if_then:                                          ; preds = %while_body6
  store i16 %a17, ptr %gep, align 2
  store i16 %a13, ptr %gep16, align 2
  br label %if_cont

if_cont:                                          ; preds = %while_body6, %if_then
  %binop35 = add i16 %s.0, 1
  br label %while_cond5
}
```

Figura 5 – Fragmento de código LLVM IR gerado pelo compilador Robcmp perante o programa da Figura 10.

2.4 Back end

Após ser gerado pelo *front end*, o código intermediário não mais possui erros que impeçam a geração de código alvo. Portanto, pode ser convertido em código de máquina. De acordo com Lopes e Auler (2014), o *back end* tem como principal papel receber um código intermediário gerado pelo *front end* e retornar um código-alvo, ou código-objeto (geralmente escrito em *assembly*, específico para a arquitetura alvo).

O LLVM é um *framework* escrito em C/C++, que realiza a otimização e geração de código alvo para várias arquiteturas, a partir de sua linguagem intermediária, a LLVM IR. Com a linguagem intermediária, o LLVM consegue otimizar código e gerar um programa objeto simples (LATTNER; ADVE, 2004).

2.5 Microcontroladores

Microcontroladores são considerados computadores em um *chip*, pois eles possuem CPU, memória RAM, uma forma de ROM em memória FLASH, persistente, e portas de Entrada/Saída, assim como computadores, porém possuem tanto poder computacional quando tamanho reduzidos (HUSSAIN et al., 2016). Alguns exemplos de microcontroladores comuns são:

- AVR: Desenvolvido pela Microchip Technology, é uma família de microcontroladores focados em performance, baixo consumo energético e flexibilidade. Possuem modelos de 2kB a 16kB de memória RAM e *clock* de no máximo 32 MHz. A Figura 6 apresenta um exemplo de microcontrolador AVR fabricado pela Microchip Technology, sendo

este um AVR16DD14, que possui 14 pinos de entrada/saída, 24 MHz de frequência máxima, 2kB de memória SRAM e 16kB de memória FLASH (INCORPORATED, 2024).

- STM32: Desenvolvido pela ST Microelectronics, é uma família de microcontroladores de 32 bits baseado no núcleo ARM Cortex, que combinam alta performance, capacidades de tempo-real, processamento de sinal digital, baixo consumo e baixa voltagem de operação. Na Figura 7 vemos um exemplo de uma placa de circuito contendo um microcontrolador STM32, fabricado pela ST Microelectronics (o losango, na figura). O modelo da imagem é um STM32F103C8T6, com 72 MHz de frequência máxima, 20kB de memória SRAM, 128kB de memória FLASH e 37 portas de entrada/saída.

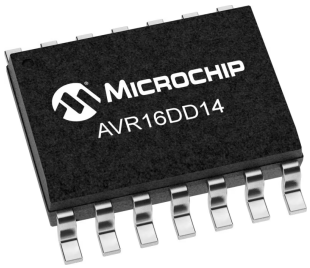


Figura 6 – Exemplo de microcontrolador AVR fabricado pela Microchip Technology, antiga Atmel. Este é um AVR16DD14 que possui 14 portas de entrada/saída, 24 MHz de frequência máxima, 2kB de memória SRAM e 16kB de memória FLASH (INCORPORATED, 2024).

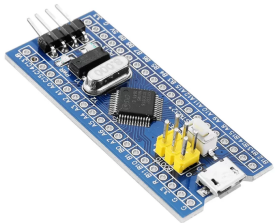


Figura 7 – Exemplo de placa contendo um microcontrolador STM32, fabricado pela ST Microelectronics. O modelo da imagem é um STM32F103C8T6, com 72 MHz de frequência máxima, 20kB de memória SRAM, 128kB de memória FLASH e 37 portas de entrada/saída. Os demais componentes na placa são necessários para o funcionamento do microcontrolador, incluindo o oscilador, a alimentação e o acesso as portas de entrada/saída pelos orifícios nas bordas laterais. (MICROELECTRONICS, 2023).

2.6 Linguagem Especifica de Domínio

Segundo Mernik, Heering e Sloane (2005) uma linguagem de domínio especifico (ou DSL) é uma linguagem feita com proposito de se encaixar em um problema existente, com maior coesão e facilidade de uso, comparados a linguagens de propósito geral. Uma DSL pode ser modelada de modo que durante sua criação sejam tratados problemas de uma determinada área, tornando até mesmo sua interpretação e escrita de código mais intuitiva.

Como exemplos de DSLs temos o SQL para consulta a banco de dados e R para estatística. A Figura 8 exemplifica uma consulta SQL, que é uma linguagem de domínio especifico para banco de dados, na qual a estrutura do código descreve como será realizada a consulta no banco de dados. Portanto, trata-se de uma linguagem mais clara, utilizando tanto palavras-chaves quanto uma lógica de escrita mais próxima do modo de funcionamento de seu domínio.

```
Select * from Mydatabase.School
where School.City like "São Paulo"
order by School.SchoolId asc;
```

Figura 8 – Exemplo de código SQL, realizando uma consulta em um banco de dados. A consulta apresentada seleciona as escolas da cidade de São Paulo e as ordena de forma descente de acordo com a variável SchoolId, mostrando no final todos os atributos da tabela School pertencente ao Banco de dados Mydatabase.

O oposto de uma linguagem de domínio especifico são as linguagens de propósito geral como C, Python e Java. A Figura 9 apresenta uma implementação em código da sequência de Fibonacci na linguagem Python. A função `Fibonacci` recebe como entrada o K-ésimo termo da sequência e determina seu valor de forma recursiva. Numa linguagem de propósito geral, não temos palavras específicas do domínio. Por exemplo, palavras destinadas a atuar com sensores, atuadores, motores, dentre outros, do domínio da robótica. A implementação destas funcionalidades fica a cargo de bibliotecas ou *frameworks* específicos.

2.7 Robcmp

Apesar da quantidade de linguagens de programação existentes, ainda hoje a programação para microcontroladores usados na robótica, IoT, impressoras 3D e outros equipamentos de fabricação digital, se dá em linguagens de programação de propósito geral como C, C++ e Micro Python. Essas linguagens, apesar de já serem consolidadas, por serem de propósito geral e não terem conhecimento sobre o hardware onde serão executadas, acabam por não detectar erros que ocorrerão apenas quando em execução no

```python
def Fibonacci(n):
    if n < 0:
        print("Inválido")
    elif n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

if __name__ == "__main__":
    value = Fibonacci(7)
    print(value)
```

Figura 9 – Exemplo de código para realizar o calculo da sequência de Fibonacci utilizando recursividade na linguagem de proposito geral Python.

hardware. Um exemplo disso é o *overflow* da soma ou da pilha, deixando assim com o programador a responsabilidade de conhecer o hardware e identificar possíveis erros que possam ocorrer durante a execução no microcontrolador.

Entre 2018 e 2024, o projeto Robotics Language: Uma Linguagem de Programação de Propósito Específico para Microcontroladores, PI05974-2024 – e seu predecessor, Especificação e Construção de Protótipos Funcionais de Kits Robóticos de Baixo Custo para uso em Processos de Ensino-Aprendizagem (PI02361-2018), avançaram na implementação de um compilador e uma linguagem especifica para microcontroladores e robótica. Trabalhos de iniciação cientifica e projetos finais de curso acrescentaram recursos como suporte a *arrays* e matrizes, tipos compostos, interfaces, injeção de dependência de hardware, depuração de código em simulador e ligação independente para microcontroladores AVR e STM (ALMEIDA, 2019; MARTINS, 2018; GOMES, 2018) .

O Robcmp é um compilador de uma linguagem de programação desenvolvido para microcontroladores usados na robótica (OLIVEIRA, 2024). A linguagem tem como objetivo isolar as especificidades dos microcontroladores, por meio de uma camada de abstração de hardware no próprio compilador e em sua biblioteca padrão assim sendo independente de *frameworks*. Além de aumentar o poder da análise semântica, essas especificidades evitam erros que ocorreriam apenas em tempo de execução, durante a própria compilação.

Com a intenção de auxiliar o programador a visualizar a estrutura de seu programa foi desenvolvida uma extensão para colorir a sintaxe de arquivos que serão compilados pelo Robcmp (com extensão .rob). A extensão possui o nome RobCmpSyntax e se integra ao Visual Code. Na Figura 10 podemos visualizar um exemplo de sintaxe colorida de um programa, que possui a coloração azul para palavras reservadas do sistema como *void*, *int16*, *while*, *if* e *return*, coloração verde para números e coloração preta para funções e nomes de variáveis. Esses detalhes corroboram para o desenvolvimento e intuitividade do programador. Também é possível depurar o programa, operação esta que ajuda o desenvolvedor a entender qual o estado atual de seu programa, verificar se o algoritmo está realizando o que se espera e até mesmo conferir o causador de algum erro.

```
1   void sort(int16[] a) {
2       t = int16(0);
3       while t < a.size - 1 {
4           s = t + 1;
5           while s < a.size {
6               if a[t] > a[s] {
7                   temp = a[t];
8                   a[t] = a[s];
9                   a[s] = temp;
10              }
11              s++;
12          }
13          t++;
14      }
15  }
16  int16 main() {
17      a = {int16(34), -5, 6, 0, 12,
18           100, 56, 22, 44, -3, -9,
19            12, 17, 22, 6, 11 };
20      r = {int16(-9), -5, -3, 0, 6,
21           6, 11, 12, 12, 17, 22, 22,
22            34, 44, 56, 100 };
23      sort(a);
24      i = 0;
25      while i < a.size {
26          if a[i] != r[i] {
27              return 1;
28          }
29          i++;
30      }
31      return 0;
32  }
```

Figura 10 – Exemplo de código escrito na linguagem Robcmp utilizando a extensão RobCmpSyntax v0.0.1 para deixar o código colorido.

2.8 Bibliotecas Padrão de Linguagens de Programação

Toda linguagem de programação traz consigo um conjunto de funções básicas para conversão entre tipos, algoritmos de ordenação, funções matemáticas, que é chamado de biblioteca padrão da linguagem. Por exemplo em linguagem C, as funções trigonométricas *sin* e *cos* que calculam os valores trigonométricos do seno e cosseno de um ângulo, *strcat* que concatena duas *strings*, *qsort* que utiliza o algoritmo de ordenação *quick-sort* para ordenar algum tipo de dado.

A biblioteca padrão é disponibilizada com a intenção de auxiliar o programador com códigos pré-selecionados e reutilizáveis, os quais o mesmo necessita usar em qualquer sistema que desenvolve, tornando assim viável a produção de sistemas complexos em menor tempo.

A fim de melhorar o desenvolvimento, Deitel e Deitel (1999) orienta o uso de recursos já implementados em uma linguagem de programação, aumentando assim a produtividade de código. Essa é, inclusive, uma técnica muito utilizada na orientação a objetos. Além do aumento da produtividade, recursos de bibliotecas padrões já possuem

otimizações e técnicas para realizar a função desejada, de forma a se utilizar o mínimo de recursos necessários, sendo esses recursos memória e tempo de processamento.

2.8.1 A biblioteca padrão C (LibC)

A linguagem C não provê de forma nativa (palavras chaves da linguagem, por exemplo) as funções para realizar operações de entrada/saída, gerenciamento de memória e manipulação de strings. Ao contrário, provê tais recursos mediante a implementação de uma biblioteca padrão, que durante a compilação é vinculada aos programas (LOOSEMORE et al., 2024). Essa biblioteca é chamada LibC. Nela, funções são disponibilizadas por meio de arquivos de cabeçalho, *headers*, como `math.h`, `stdio.h`, `stdlib.h`, além de outros.

As funções contidas na LibC são de diversos tipos. Plauger (1992) agrupa as funções da biblioteca padrão em seis grupos, sendo eles: matemática de inteiros, algorítmos (busca, ordenação, números aleatórios), conversão de textos, alocação de espaço na memória, interação com ambiente (*abort*, *exit*, *system*, *getenv*) e conversão de multi-bytes;

2.8.2 Biblioteca padrão em linguagens orientadas a objetos

A nominação dada à biblioteca padrão da linguagem orientada a objeto C++ é Standard Template Library. Nela, todos os componentes são *templates* de outro componente. A biblioteca em si é composta de vários componentes bem estruturados e sua essência está nos contêineres, iteradores e algoritmos, sendo um ótimo exemplo de programação genérica (JOSUTTIS, 2012). Tal conceito é baseado na separação entre dados e operações, onde os dados são gerenciados por contêineres e os operadores são definidos por algoritmos de configurações. Os iteradores são formados pela união entre o contêiner e os algoritmos.

Já na linguagem Java, a organização se dá em pacotes. Uma pequena biblioteca foi adicionada no pacote Java.util referenciado como *Java Colections*, ou JC. Da mesma forma, existe a *Generic Library for Java*, ou JGL. Esses dois pacotes fornecem a implementação de estruturas como mapas, dicionários e sequências (TAMASSIAYZ et al., 2001). A biblioteca padrão Java possui, além desses pacotes, uma serie outros pacotes que foram adicionados conforme o avanço da linguagem; como pacotes para operações matemáticas (*math*); datas, tempos, instantes ou contagem de tempo (*time*) e gerador de números aleatórios (*random*).

De forma similar, linguagens mais novas, como Julia[1] e Rust[2] também disponibilizam suas biblioteca padrão na forma de pacotes ou APIs. Na linguagem Julia há diversos módulos dentro de sua biblioteca padrão, como por exemplo, o módulo *dates* para

[1] https://julialang.org
[2] https://www.rust-lang.org

trabalhar com datas com precisão de até milissegundos de precisão, módulo *statistics* para trabalhar com operações de estatística (desvio padrão, variância, média, moda e mediana) (LANGUAGE, 2024). Na biblioteca padrão da linguagem Rust, os módulos existentes realizam operações como alocação de memória (*alloc*), contagem de tempo com diferentes unidades de medida (*time*) e *multithreading* para programação em paralelo (*threads*) (Rust Team, 2024).

2.8.3 Conjunto de funções comuns nas bibliotecas padrão

Uma lista não exaustiva das funções comuns nas bibliotecas padrão das linguagens é apresentada na Tabela 1, incluindo a sua descrição e exemplos. Um subconjunto desta lista, relacionado às funções matemáticas, foi implementado na biblioteca padrão da Robotics Language e compôs o escopo deste trabalho.

Operação	Descrição	Exemplos
Funções Matemáticas		
`cosf` e `cos`	As funções cos e cosf retornam o cosseno do seu argumento (a), informado em radianos.	C: cos(a), cosf(a) Java: Math.cos(a)
`sinf` e `sin`	As funções sin e sinf retorna o seno do seu argumento (a), informado em radianos.	C: sin(a), sinf(a) Java: Math.sin(a)
`pow`	A função pow retorna a potenciação de seu argumento a elevado seu argumento b (a,b), calculado em ponto flutuante.	C: pow(a,b) Java: Math.pow(a,b)
`sqrt`	A função sqrt retorna a raiz quadrada de seu argumento (a), calculado em ponto flutuante.	C: sqrt(a) Java: Math.sqrt(a)
`log`	A função log retorna o logaritmo de seu argumento (a) na base 10, calculado em ponto flutuante.	C: log(a) Java: Math.log(a)
`ln`	A função ln retorna o logaritmo de seu argumento (a) na base e, calculado em ponto flutuante.	C: ln(a) Java: Math.ln(a)
Ordenação		
`qsort` e `sort`	As funções qsort e sort retornam um vetor/lista do seu argumento (a) ordenado.	C: qsort(a) Java: Collections.sort(a)
Números aleatórios		
`rand`	A função rand retorna um valor aleatório menor que a(opcional).	C: rand()%a Java: new Random().nextInt(a);

Tabela 1 – Funções comuns na biblioteca padrão das linguagens de programação

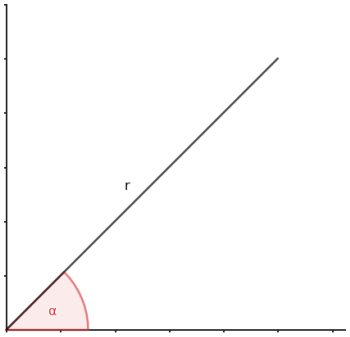
2.9 CORDIC

O *Cordinate Rotation Digital Computer* (VOLDER, 2000), conhecido por seu acrônimo CORDIC, foi primeiramente descrito em 1959 por Jack E. Volder, e é um algoritmo simples e eficiente para calcular funções hiperbólicas e trigonométricas, comumente

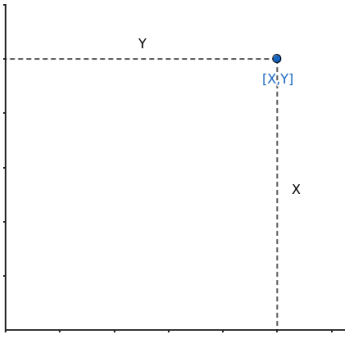
usado quando nenhum hardware multiplicador está disponível (o caso em particular dos microcontroladores), e devido a isso foi empregado neste trabalho para a implementação das funções trigonométricas.

O Funcionamento do CORDIC se dá pela rotação de um vetor $[x, y]$ (geralmente iniciado com $x = 1$ e $y = 0$) que é rotacionado por um ângulo, gerando assim um novo vetor $[x', y']$. O sentido dá rotação (horário ou anti-horário) define se o ângulo será adicionado ou subtraído do vetor atual. O vetor de rotação do CORDIC é composto por um vetor de i posições com cada posição valendo $arctan(2^i)$; quanto maior o vetor de rotação do CORDIC, maior será o número de iterações possíveis de se realizar, assim, aumentando a precisão das funções matemáticas calculadas.

O vetor de rotação armazena apenas o valor do raio do sistema de coordenadas polares, equivalente à coordenada no sistema de coordenadas retangulares. Ambos os planos conseguem representar um plano, porém com unidades diferentes. A Figura 11a ilustra uma representação de um ponto no plano polar que utiliza um raio de tamanho r e um ângulo α para representar a distância de um ponto desde a origem do plano e sua inclinação; A Figura 11b ilustra a representação de um ponto no plano retangular que a partir dos valores x e y geram uma posição única no plano (ponto), com as devidas coordenadas $[x, y]$.



(a) Plano polar



(b) Plano retangular.

Figura 11 – Dois sistemas de coordenadas diferentes, o Plano polar (a) e o Plano retangular (b).

Para o funcionamento do CORDIC, são definidos valores de inicialização $(x_0, y_0) = (1, 0)$ e ângulo $\theta_0 = 0$. Para cada iteração, a rotação é aplicada sobre o vetor de entrada (x_i, y_i) de forma que o vetor se ajuste ao ângulo i do vetor de rotação, de acordo com a seguinte formula de rotação:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} x_i - \sigma_i \cdot 2^{-i} \cdot y_i \\ y_i + \sigma_i \cdot 2^{-i} \cdot x_i \end{bmatrix}$$

Onde σ é um sinal que vale 1 quando o sentido da rotação é no sentido horário e -1 quando o sentido da rotação é anti-horário. Este sinal é escolhido de acordo com o ângulo desejado

para aproximar-se da direção correta.

No final de n iterações, os valores finais das coordenadas x_n e y_n são utilizados para calcular o valor da função desejada.

O pseudocódigo 1 representa como o algoritmo CORDIC funciona de forma geral. Nas primeiras 5 linhas são feitas a inicialização das variáveis, (x, y)=$(1, 0)$ que representam um ponto (x, y) no plano retangular, z que representa angulo inicial do ponto (x, y) e K sendo a correção de escala – sem essa variável, a cada iteração, a norma do vetor, isto é, o valor de (x, y) em coordenadas polares, deixaria de ter raio $r = 1$, pois com a rotação do vetor não é preservada sua norma. Na linha 8, preenchemos o vetor que armazenará os valores de rotação onde cada posição do vetor conterá $atan(2^{-i})$. Entre as linhas 10 e 21 é onde ocorre a rotação do vetor (x, y). Entre as linhas 11 e 14 ocorre a escolha do sentido da rotação (horário ou anti-horário). Entre as linhas 16 e 21 ocorre a rotação do vetor, atualização dos valores (x, y) e ângulo z. O valor de saída se dá nas linhas de 23 a 26, onde também é feito o ajuste de correção de escala.

Algorithm 1 Pseudocódigo CORDIC

1: **Função CORDIC**(ângulo desejado θ, número de iterações n)
2: $x \leftarrow 1.0$ ▷ Valor inicial de x
3: $y \leftarrow 0.0$ ▷ Valor inicial de y
4: $z \leftarrow \theta$ ▷ Ângulo inicial de (x, y)
5: $K \leftarrow 1.0$ ▷ Fator de ganho inicial
6:
7: **for** $i \leftarrow 0$ **to** $n - 1$
8: vetorRotacao$[i] \leftarrow \text{atan}(2^{-i})$
9:
10: **for** $i \leftarrow 0$ **to** $n - 1$
11: **if** $z < 0$**:**
12: $\sigma \leftarrow -1$ ▷ Rotação no sentido horário
13: **else**
14: $\sigma \leftarrow 1$ ▷ Rotação no sentido anti-horário
15:
16: $x_{\text{novo}} \leftarrow x - \sigma \cdot 2^{-i} \cdot y$
17: $y_{\text{novo}} \leftarrow y + \sigma \cdot 2^{-i} \cdot x$
18: $z_{\text{novo}} \leftarrow z - \sigma \cdot \text{vetorRotacao}[i]$ ▷ Atualiza o ângulo ou o valor
19: $x \leftarrow x_{\text{novo}}$
20: $y \leftarrow y_{\text{novo}}$
21: $z \leftarrow z_{\text{novo}}$
22:
23: **for** $i \leftarrow 0$ **to** $n - 1$
24: $K \leftarrow K \cdot \sqrt{1 + 2^{-2i}}$ ▷ Fator de ganho para correção de escala
25:
26: Return $(x \cdot K, y \cdot K)$ ▷ Retorno para funções trigonométricas

3 Trabalhos relacionados

3.1 Introdução

Neste capítulo, são apresentados os trabalhos relacionados à implementação da biblioteca padrão de uma linguagem de programação específica do domínio da robótica: funções matemáticas, encontradas a partir de uma revisão narrativa de literatura (RNL) que partiu do conjunto de monografias já defendidas no contexto do projeto de pesquisa ao qual este trabalho também se insere (ALMEIDA, 2019; MARTINS, 2018; GOMES, 2018), seguido pela busca de trabalhos correlatos no PlatformIO Registry (PlatformIO Labs OÜ, 2024).

O PlatformIO[1] consiste em uma plataforma de desenvolvimento de código aberto voltada para sistemas embarcados. Ele oferece uma ferramenta completa que simplifica o processo de criação, desenvolvimento e execução de projetos para microcontroladores. A plataforma é compatível com diversas placas e plataformas de *hardware*, permitindo aos desenvolvedores trabalhar de forma eficiente com diferentes dispositivos. Além disso, o PlatformIO possui integração com editores de código como o Visual Studio Code, fornecendo recursos como gerenciamento de bibliotecas, depuração e compilação automatizada, tornando-se uma escolha popular para engenheiros e desenvolvedores de *software* embarcado.

Dentro do site do PlatformIO há o PlatformIO Registry[2], um repositório online que fornece acesso a uma diversa coleção de bibliotecas, plataformas e ferramentas utilizadas no desenvolvimento de sistemas embarcados. Assim podemos encontrar as atuais tecnologias disponíveis para desenvolver software para MCUs. Por se tratar de um repositório extensivo de soluções para microcontroladores, este catálogo foi usado nas buscas da RNL usada neste trabalho.

Para selecionar e comparar os trabalhos encontrados, adotou-se os seguintes critérios:

- O compilador gera código alvo para microcontroladores (MCU)
- O domínio da linguagem do compilador é voltado para programação em microcontroladores (DOMÍNIO)
- A biblioteca, *framework* ou compilador é *open source* (OPEN)
- A biblioteca, *framework* ou compilador é matura (MAT)

[1] https://platformio.org/
[2] https://registry.platformio.org/

Para analisar o nível de maturidade foram analisados se a biblioteca, *framework* ou compilador possuem as características de: *i*) Histórico de versões e lançamentos, *ii*) Correção de *bugs*, *iii*) Documentação completa abordando exemplos, instalação e uso avançado, *iv*) Comunidade ativa com contribuições externas, *v*) Uso no mercado por empresas e projetos. Somente os trabalhos que possuem as cinco características foram aprovados.

3.2 Trabalhos analisados

Com base nos critérios apresentados na seção anterior, foram analisados 4 trabalhos com destaque em quatro mais relevantes encontrados a seguir.

3.2.1 Rust

Rust (Rust Team, 2024) é uma linguagem de programação cuja primeira versão foi lançada em 2015, com foco na segurança e desempenho. Assim como o Robcmp, a linguagem Rust possui como *back end* o LLVM. O maior destaque da linguagem Rust é seu controle de memória, evitando erros de acesso a memória e assim aumentando a segurança da aplicação.

Apesar do suporte nativo a microcontroladores não ser completo na linguagem, há projetos como o AVR-Rust (AVR-RUST, 2024), uma extensão do compilador Rust, projetada para suportar microcontroladores AVR, que pode ser utilizado em sistemas embarcados, como o ATmega328p. O principal objetivo é oferecer uma alternativa segura e eficiente ao tradicional uso de C/C++ nesses dispositivos.

3.2.1.1 AVR-GCC

O AVR-GCC é a *toolchain* mais utilizada para desenvolvimento de software para MCUs da Atmel AVR, sendo disponibilizada como ferramenta da própria fabricante (Microchip, antiga Atmel) (Microchip Technology Inc., 2024) e também utilizada na IDE do Arduino, que a utiliza para gerar e atualizar código nas placas Arduino.

O compilador da *toolchain*, avr-gcc, é um compilador para as linguagens C, C++ e Ada para código de máquina como alvo microcontroladores da família AVR. Ele utiliza parte do compilador de código aberto GNU Compiler Collection (GCC) como programa *driver* para outros programas, na qual todo o conjunto cria a saída final do compilador, o programa objeto na linguagem da MCU AVR. Entretanto, o AVR-GCC não utiliza

sua própria biblioteca padrão, a AVR-LibC, que fornece a maioria das funções contidas na biblioteca padrão do GCC – a gLibC, além de uma biblioteca adicional com funções especificas para uma MCU AVR.

3.2.1.2 ARM-GCC

Outra *toolchain* baseada no compilador GCC e específica para plataforma ARM é denominada arm-none-eabi-gcc. O significado dos acrônimos é: *arm* se refere a arquitetura ARM na qual o compilador irá gerar código alvo; *none* indica que o ambiente destino não possui um sistema operacional; *eabi* significa *Embedded Aplication Binary Interface*, que define como o código binário se comunica com o *hardware* e, por fim, o *gcc* se refere ao compilador que será utilizado para gerar código de máquina.

Sobre esta *toolchain*, existem alguns *frameworks*, listados no PlatformIO Registry, especificamente em PlatformIO (2024), que apoiam o desenvolvimento de software para MCUs STM32, como Mbed[3], CMSIS[4], LibOpenCM3[5] e Arduino[6]. Todos eles disponibilizam suporte a funções matemáticas a partir de uma implementação padrão da LibC, para plataforma ARM, por padrão disponibilizada pela *toolchain*, mas que pode ser substituída por outras implementações alternativas compatíveis. Alguns possuem classes (orientação a objeto) específicas para *string* e acesso a periféricos facilitados e ambas constituem alternativas maduras para o desenvolvimento para MCUs ARM, em geral.

3.2.2 Majestic: Uma Ampliação de uma Linguagem de Programação para Robótica Educacional

Com foco na robótica educacional, Almeida (2019) realizou seu trabalho implementando recursos úteis para uma linguagem de programação, abrindo portas para a criação de novos algoritmos simples e complexos além de adicionar um tipo de estrutura de dado na qual o usuário pode utilizar para desenvolver seus próprios códigos.

O software desenvolvido por Almeida (2019) utilizou o Robcmp como base para a implementação de três novos recursos: funções com parâmetros, vetores e matrizes. Com estas novas funcionalidades, foi-se criada uma nova linguagem, denominada pelo próprio autor de *Majestic Language*, e seu compilador. Futuramente, o Robcmp veio a adotar e implementar em seu próprio código essas novas funcionalidades.

Dentre os trabalhos futuros sugeridos (ALMEIDA, 2019) menciona a realização de

[3] https://registry.platformio.org/tools/platformio/framework-mbed
[4] https://registry.platformio.org/tools/platformio/framework-cmsis
[5] https://registry.platformio.org/tools/platformio/framework-libopencm3
[6] https://registry.platformio.org/tools/platformio/framework-arduinoststm32

experimentos comparativos com AVR-GCC e o *backend* AVR do LLVM – o que realizamos neste trabalho.

3.3 Resumo Comparativo

A Tabela 2 apresenta um resumo comparativo dos trabalhos analisados na seção anterior com os critérios construídos a partir da revisão de literatura. A classificação foi estabelecida na seguinte fundamentação: os trabalhos que apresentam a característica serão atribuídos "+" e, para os que não apresentarem o critério, será atribuído "-".

Como verificado na Tabela 2, os trabalhos AVR-GCC e ARM-GCC são os trabalhos que possuem um maior nível de maturidade, isto é, são frequentemente a escolha principal para o desenvolvimento nas suas respectivas arquiteturas e possuem uma documentação elaborada, facilitando assim para que outras pessoas consigam compreender o funcionamento e modificar o projeto. Uma ressalva para o trabalho Rust, que possui documentação, porém seus desenvolvedores assumem a existência de erros que necessitam de atenção.

Tabela 2 – Comparativo entre trabalhos relacionados

Trabalhos	Critérios			
	MCU	DOMÍNIO	OPEN	MAT
Rust	+	-	+	-
AVR-GCC	+	-	+	+
ARM-GCC	+	-	+	+
Majestic	+	+	+	-
Este trabalho	+	+	+	-

Todos os trabalhos relacionados contemplam o critério MCU que aborda se o compilador gera código alvo para MCUs. O critério DOMÍNIO, que aborda se o domínio da linguagem de cada compilador é voltado para microcontroladores, foi o critério menos contemplado, devido o custo de uma DSL ser muito elevado e um baixo retorno sobre, opta-se por utilizar recursos de uma linguagem bem matura e realizando adaptações para que assim se possa obter um compilador "adaptado" para compilar código para uma MCU.

Quanto ao critério OPEN, que aborda se o *framework* é *open-source*, foi contemplado por todos os trabalhos, na qual é possível verificar como os outros compiladores realizam seus próprios cálculos de funções, corrigir erros ou até mesmo realizar personalizações no código.

O critério MAT, que trata sobre o nível de maturidade do *framework*, foi o critério com maior divisão. Nele, os trabalhos AVR-GCC e ARM-GCC, ambos baseado na suíte de compiladores GCC, são considerados *toolchain*'s estáveis e as principais escolhas de

programação para as respectivas plataformas. Os trabalhos Rust (AVR-Rust) e Majestic, devido serem opções mais recentes, apresentam nível de maturidade inferior.

4 Implementação

4.1 Introdução

Neste capítulo apresenta-se alguns detalhes da implementação das funções no escopo deste trabalho. O código fonte da implementação pode ser encontrado no repositório online <https://github.com/RyanFrancys/robcmp/tree/ryan/lib/math>.

4.2 Algoritmo CORDIC

4.2.1 Funções trigonométricas (sin, cos e tan)

As funções trigonométricas foram implementadas no arquivo <https://github.com/RyanFrancys/robcmp/blob/ryan/lib/math/trigonometric.rob>. Descreve-se, a seguir, um exemplo explicativo de seu funcionamento.

Suponhamos que queremos calcular $\sin(45°)$ e $\cos(45°)$ utilizando o algoritmo CORDIC. O ângulo de 45 graus pode ser representado em radianos como $\theta = \frac{\pi}{4}$. O objetivo do CORDIC é iterativamente aproximar os valores de seno e cosseno usando as seguintes operações:

1. **Inicialização:** Começamos com as coordenadas $x_0 = 1$, $y_0 = 0$ (vetor inicial no eixo x) e $z_0 = \theta = 45°$. O valor de z_0 é o ângulo que desejamos calcular, e as coordenadas x_0 e y_0 são os valores iniciais do vetor de rotação.

2. **Primeira iteração:** Calculamos o valor da rotação usando um ângulo fixo, por exemplo, $\text{atan}(2^{-0}) = 45°$, e comparamos o sinal de z_0 com o valor dessa rotação. No caso de z_0 ser positivo, a rotação será no sentido anti-horário.

3. **Iteração seguinte:** Continuamos a aplicar rotações sucessivas (com ângulos menores) até que o valor de z se aproxime de zero (ou o erro seja suficientemente pequeno). Em cada iteração, as coordenadas x e y se aproximam dos valores de $\cos(45°)$ e $\sin(45°)$, respectivamente.

Após o número de iterações fornecido pelo usuário, as coordenadas finais x_n e y_n fornecerão os valores de $\cos(45°)$ e $\sin(45°)$ com certa precisão, a partir destes valores é possível obter o valor da tangente utilizando uma operação de divisão entre o seno e cosseno. Observou-se que a partir de 20 iterações eram-se obtidos uma precisão desejável de 5 casas decimais.

4.3 Algoritmos aproximativos

4.3.1 Função raiz quadrada

Diferentemente dos métodos anteriores, para o calculo da raiz quadrada, foi adotado uma metodologia de cálculo que realiza a aproximação por potência de dois. Esse método é interessante para situações em que a implementação eficiente e a rapidez no cálculo são mais importantes do que uma precisão infinita, sendo adequado para dispositivos com recursos computacionais limitados.

A função de raiz quadrada, sqrt, foi implementada no arquivo <https://github.com/RyanFrancys/robcmp/blob/ryan/lib/math/sqrt.rob>.

Consideremos um cenário em que precisamos calcular a raiz quadrada de um número $x = 20.0$, o código iterativamente calcula a raiz quadrada de x. O fluxo de execução seria o seguinte:

1. Como $x > 1$, o algoritmo começa com uma estimativa inicial de poweroftwo $= 1.0$.
2. O algoritmo então aumenta poweroftwo multiplicando por 2 até que o quadrado de poweroftwo seja maior que x.
3. O valor de y é ajustado, e o algoritmo entra em um loop iterativo, refinando a aproximação da raiz quadrada.
4. Após 22 iterações, o valor final de y será uma boa aproximação de $\sqrt{20.0}$.

A valor 22 foi escolhido a fim de maximizar a quantidade de casas decimais de precisão. Esse cálculo é realizado sem usar operações de multiplicação complexas, o que torna a função particularmente eficiente em hardware com limitações, como em dispositivos embarcados, onde o custo de operações aritméticas mais pesadas pode ser significativo.

4.3.2 Função logaritmo(ln)

A função logaritmo, ln, foi implementada no arquivo <https://github.com/RyanFrancys/robcmp/blob/ryan/lib/math/ln.rob>.

Utilizando de uma tabela auxiliar para se aproximar com uma maior precisão os resultados, a função ln calcula a aproximação do valor do logaritmo natural de um número x. Suponha que a função receba de entrada $x = 100.0$. O processo seria o seguinte:

1. O valor de x começa como 100.0, que é maior que 1, então não há necessidade de multiplicação ou divisão para normalizar x nesse caso.

2. O algoritmo utiliza a tabela de aproximações ‘a’ para reduzir o valor de x em etapas, dividindo sucessivamente por cada valor de ‘a[i]‘ até que x se aproxime de um valor simples.

3. Uma vez que o valor de x foi ajustado, o algoritmo aplica transformações adicionais para melhorar a precisão da aproximação.

4. Por fim, o valor de k, que foi acumulado durante as transformações, é somado ao resultado final.

5 Avaliação e Testes

5.1 Ambiente Experimental

Para realizar os testes foram utilizados como MCUs o simulador SimAVR, um simulador de microcontroladores da família AVR da microchip, simulando as MCUs atmega328p (até 16MHz, 2Kbytes de SRAM e 32kbytes de memória ROM)[1] e atmega1284p (até 20MHz, 128KBytes de ROM e 16KBytes de SRAM)[2]; quanto aos testes com o STM32, foram executados testes no STM32f103c8t6 (até 72MHz, 20kBytes de SRAM e 128KBytes ROM)[3] como MCU da STmicroelectronics.

Para a compilação dos códigos – *cross* compilação, foi utilizado o Sistema Operacional Ubuntu 22.04.5 LTS, num Processador Ryzen 5500U com 16GB de RAM. Para compilar o código em linguagem C fo utilizado o AVR-GCC versão 5.4.0 para MCUs da familia STM32. Para compilar os programas em linguagem C para MCUs STM32 foi utilizado o compilador *arm-none-eabi-gcc* versão 12.3.1.

As seguintes variáveis foram observadas, para análise de desempenho em diferentes linguagens:

- Tempo de execução do programa no simulador ou diretamente na MCU (STM32) – em segundos
- Tamanho do código alvo gerado – em bytes

Para a realização dos experimentos foram desenvolvidos 6 testes unitários, um para cada função implementada na biblioteca matemática do Robcmp. Cada algoritmo foi compilado uma vez utilizando os mesmos parâmetros e as mesmas *flags* de otimização '-Os'. Durante os experimentos com o simulador *SimAVR*, apenas o simulador estava sendo executando na máquina em conjunto com o SO. Para medir o tempo, usou-se o comando '*time*', fornecido pelo próprio SO.

Os algoritmos em linguagem Robcmp utilizados para estes experimentos estão disponíveis em repositório online, tanto os códigos em Robcmp <https://github.com/RyanFrancys/robcmp/tree/ryan/test/general> quanto os códigos em linguagem C <https://github.com/RyanFrancys/robcmp/tree/ryan/CodeTest>. Procurou-se, no máximo pos-

[1] <https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf>
[2] <https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42719-ATmega1284P_Datasheet.pdf>
[3] <https://www.st.com/en/microcontrollers-microprocessors/stm32f103c8.html>

sível, manter os algoritmos em Robcmp e linguagem C equivalentes em termos de funcionalidade e mínimos em termos de tamanho de código implementado.

5.1.1 Experimentos com tempo de execução do programa

O *SimAVR* é um simulador de microcontroladores AVR que opera sobre um Sistema Operacional (SO), utilizando os recursos e as funções do sistema para realizar a simulação do microcontrolador, assim o *SimAVR* interage com o ambiente do sistema, o que pode impactar o tempo de execução da simulação, uma vez que o SO gerencia processos, alocação de memória e a distribuição de ciclos de CPU.

Neste sentido, para se obter os valores de tempo de execução nas MCUs da família *AVR* (atmega1284p e atmega328p), no simulador *SimAVR*, foi utilizado o comando *time*, que realiza a medição de tempo de uma tarefa ou comando utilizando-se apenas do tempo de usuário – que representa ao tempo que o processador passou executando o código do programa.

Cada programa foi executado um total 10 vezes e retornando uma média de tempo. A primeira média de tempo foi ignorada devido o tempo gasto pelo SO para *cache* de disco na memória. A média do tempo das demais execuções foi calculada para este experimento.

Quanto aos testes de tempo com o STM32, foram executados os programas na própria MCU. Para isso, foi desenvolvido um programa na linguagem Robcmp, que executa num loop infinito a contabilização do tempo dentro de uma função chamada run_test. O valor do tempo foi escrito em uma porta serial para que se pudesse registrar.

Os valores obtidos estão dispostos na Tabela 3. Todos os valores estão representados em segundos. Devido ao limite de tempo para implementação dos códigos em C para o STM32, a comparação foi deixada para trabalhos futuros e não está disposta na tabela.

Algoritmo	Atmega328p		Atmega1284p		STM32
	Robcmp	AVR-GCC	Robcmp	AVR-GCC	Robcmp
Cosseno(Cos)	0.129	0.013	0.129	0.013	0.445
Seno(Sin)	0.129	0.012	0.129	0.013	0.443
Tangente(Tan)	0.125	0.014	0.127	0.015	0.474
Logaritmo(Ln)	0.069	0.013	0.071	0.013	0.266
Exponencial(Exp)	0.052	0.013	0.051	0.013	0.198
Raiz quadrada(Sqrt)	0.049	0.005	0.049	0.004	0.207

Tabela 3 – Comparação de tempo de execução entre as linguagens Robcmp e C. Tempos em segundos.

5.1.2 Experimentos com Tamanho do programa

Para cada compilador existem ferramentas que são específicas para fornecerem uma análise detalhada do tamanho das diferentes seções do programa, como o código executável (seção .text), dados (seção .data) e variáveis não inicializadas (seção .bss). Elas são mais adequadas do que outros comandos, como '*ls -l*', que simplesmente exibe o tamanho físico do arquivo, incluindo a sobrecarga do sistema de arquivos e quaisquer metadados, mas não reflete o uso real da memória durante a execução do programa, tornando assim o *avr-size* pertencente ao *avr-gcc* e *llvm-size* pertencente ao Robcmp mais precisos e adequados para esta análise. Para este experimento em ambas ferramentas foram utilizados os valores da seção *dec* que representa o tamanho total do programa utilizando representação decimal.

O conteúdo apresentado na Tabela 4 abrange todos os resultados que foram obtidos durante os experimentos, na qual todos os valores estão representados em *bytes*.

Algoritmo	Atmega328p		Atmega1284p		STM32
	Robcmp	AVR-GCC	Robcmp	AVR-GCC	Robcmp
Cosseno(Cos)	4912	3192	4932	3362	9376
Seno(Sin)	4806	3198	4826	3368	9344
Tangente(Tan)	4720	3238	4740	3408	9312
Logaritmo(Ln)	5974	3108	5674	3280	8912
Exponencial(Exp)	5460	3360	5994	3536	9648
Raiz quadrada(Sqrt)	5654	2612	5480	2784	9552

Tabela 4 – Comparação de tamanho de arquivo entre entre as linguagens Robcmp e C. Valores em bytes.

5.2 Análise dos Resultados Obtidos

No aspecto de tamanho de código para microcontroladores, quanto menor o código, mais vantajoso se torna, pois permite uma utilização mais eficiente da memória, liberando espaço para outras funcionalidades. Além disso, um código mais compacto pode resultar em uma execução mais rápida, já que há menos instruções a serem processadas.

No aspecto de tempo de execução de um programa no microcontrolador, quanto menor o tempo gasto mais atrativo se é, pois se consegue um menor tempo de resposta a partir de uma entrada.

Dos valores apresentados na Tabela 3, entre as MCUs da família AVR (Atmega328p e Atmega1284p), os tempos de execução dos códigos do Robcmp foram parecidos em ambos os microcontroladores, sendo a maior diferença de tempo decorrido nos algoritmos

Logaritmo e Tangente, ambos possuindo uma diferença de tempo de apenas 0.002 segundos.

Quando compara-se os tempos do código gerado com o Robcmp (usando CORDIC), com os tempos do AVR-GCC (que usa funções otimizadas, escritas em assembly, para o mesmo propósito), a diferença se mostrou, em geral, em uma ordem de grandeza. No melhor caso para o Robcmp, o algoritmo Exponencial no Atmega1284p, o código gerado pelo AVR-GCC utiliza 25% do tempo do código gerado pelo Robcmp; o pior dos casos ocorre na função Seno com o Atmega328p, na qual o código gerado pelo AVR-GCC utiliza $9,3\%$ do tempo do código equivalente gerado pelo Robcmp.

Como mencionado anteriormente, devido ao limite de tempo para implementação dos códigos em C para o STM32, a comparação foi deixada para trabalhos futuros.

A Tabela 4 apresenta os resultados dos tamanhos de cada um dos 6 programas comparados. A partir dela, é possível observar que o AVR-GCC gerou um código com aproximadamente 40% de seu equivalente em Robcmp, sendo a maior diferença na função raiz quadrada(sqrt), na MCU Atmega328p. Nela, o AVR-GCC gerou um código com aproximadamente $46,19\%$ do tamanho do código equivalente em robcmp; a menor diferença ocorreu com o algoritmo Tangente com o código alvo da MCU Atmega1284p na qual o AVR-GCC gerou um código com tamanho $71,9\%$ comparado ao Robcmp.

Em suma, um maior esforço pode ser dedicado no futuro para inspecionar as causas do comportamento observado e identificar, no código gerado, quais são as partes que mais consumiram espaço. Por exemplo, apesar do código CORDIC não utilizar operações de multiplicação e divisão em tempo de execução, o código do teste unitário utiliza, e isso pode ter prejudicado os experimentos de tamanho (a vantagem da utilização do CORDIC pode ter sido anulada pelo teste unitário). Ainda, é possível implementar as funções trigonométricas aproveitando o núcleo de código comum entre todas elas.

Quanto ao tempo de execução, também é possível investigar o código gerado com ferramentas adequadas de *profiling* para identificar o motivo do consumo excessivo de tempo e dedicar algum esforço na otimização da própria implementação. Por exemplo, o código pode ser alterado para utilizar apenas números inteiros e operações de deslocamento de bits (*shift*).

6 Conclusões e Trabalhos Futuros

6.1 Conclusões

Considerando a necessidade de uma biblioteca matemática funcional para uma linguagem de programação de domínio específico, esta pesquisa propôs a implementação de funções que utilizam o algoritmo CORDIC.

A implementação do algoritmo CORDIC visa otimizar o cálculo de funções trigonométricas, hiperbólicas e outras operações matemáticas em ambientes que exigem alta performance e precisão. Através da abordagem proposta, foi possível observar que, embora não tenham sido obtidos os melhores tempos de execução ou o menor tamanho de programas alvo, o algoritmo é capaz de calcular com a mesma precisão que as funções matemáticas da linguagem C. Dessa forma, a pesquisa contribui para o desenvolvimento de ferramentas mais robustas e acessíveis para a comunidade de desenvolvedores, aprimorando a funcionalidade matemática das linguagens de programação específicas para domínios. O estudo também abre caminho para futuras investigações, como a integração de outros algoritmos numéricos e a exploração de técnicas que possam potencializar ainda mais o desempenho em sistemas com restrições de recursos.

Através de experimentos, foi possível concluir que as funções comparadas da biblioteca padrão do AVR-GCC, a LibC, possuem um melhor desempenho e melhor uso de espaço do que as funções implementadas no Robcmp, realizadas por este trabalho. No entanto, a precisão dos experimentos foi limitada pelo tempo disponível para otimização da implementação. Deixamos esse passo para trabalhos futuros.

Ao final desta pesquisa, obteve-se uma biblioteca matemática que abrange as funções trigonométricas (cosseno, seno e tangente), a função logarítmica (ln), a função exponencial (exp) e a função raiz quadrada (sqrt), implementadas na biblioteca padrão de um compilador com linguagem de domínio específico para a robótica.

6.2 Trabalhos futuros

Como possíveis melhoras para a biblioteca padrão do compilador Robcmp, considera-se como trabalhos futuros a adição das seguintes funcionalidades:

- Investigar e otimizar os algoritmos elaborados usando *profilers*, para melhorar seu tempo de execução e de memória;
- Simplificar a implementação dos algoritmos CORDIC usando funções de deslocamento

de bits, para reduzir o tempo de execução;

- Funções hiperbólicas: Aumentar o no repertorio da biblioteca padrão do Robcmp as funções hiperbólicas (cosh, sinh, tanh); e
- Implementar funções de conversão entre tipos, ordenação, números aleatórios e outras, para completar a biblioteca padrão.

Referências

AHO RAVI SETHI, J. D. U. A. V. *Compiladores Principios, Técnicas e Ferramentas*. [S.l.]: Springer Science & Business Media, 2007. 1–343 p. Citado 5 vezes nas páginas 9, 17, 18, 19 e 21.

AKDUR, D.; GAROUSI, V.; DEMIRÖRS, O. A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture*, Elsevier, v. 91, p. 62–82, 2018. Citado na página 14.

ALMEIDA, V. S. *Majestic: Uma Ampliação de uma Linguagem de Programação para Robótica Educacional*. 57 p. Monografia (Monografia) — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2019. Citado 3 vezes nas páginas 25, 32 e 34.

ARDUINO. *LLVM Language Reference Manual*. 2024. Disponível em:<https://www.arduino.cc/>. Acessado em: 2024-07-11. Citado na página 14.

AVR-RUST. *Book AVR-Rust*. 2024. Https://github.com/avr-rust/book.avr-rust.com/tree/master. Acessado em: 2024-11-10. Citado na página 33.

COSTA, R. H. P. et al. Linguagens formais e autômatos. *Editora Cientifica*, 2010. Citado 2 vezes nas páginas 19 e 20.

COSTA, R. H. P. et al. Compiladores. *Londrina: Editora Cientifica*, 2023. Citado na página 20.

DEITEL, H. M.; DEITEL, P. J. *Como programar em C*. [S.l.]: LTC, 1999. Citado na página 26.

FEDOZZI, R. Compiladores. *Londrina: Editora e Distribuidora Educacional*, 2018. Citado na página 19.

FOLEISS, J. H. et al. Scc: Um compilador c como ferramenta de ensino de compiladores. In: *WEAC2009-Workshop Educação em Arquitetura de Computadores*. [S.l.: s.n.], 2009. p. 15–22. Citado 4 vezes nas páginas 17, 19, 20 e 21.

FOWLER, G. S.; KORN, D. G.; VO, K.-P. Principles for writing reusable libraries. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 20, n. SI, p. 150–159, 1995. Citado na página 15.

GOMES, W. S. *Avaliação Técnica de Componentes Eletrônicos e Microprocessadores para uso no Processo de Ensino-Aprendizagem de Ciências Exatas*. 55 p. Monografia (Monografia) — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2018. Citado 2 vezes nas páginas 25 e 32.

HUSSAIN, A. et al. Programming a microcontroller. *Int. J. Comput. Appl*, v. 155, n. 5, p. 21–26, 2016. Citado 2 vezes nas páginas 14 e 22.

INCORPORATED, M. T. *AVR16/32DD14/20 datasheet*. 2024. Disponível em:<https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/AVR32-16DD20-14-Complete-DataSheet-DS40002413.pdf>. Acessado em: 2024-06-09. Citado na página 23.

JOSUTTIS, N. M. The c++ standard library: a tutorial and reference. *Addison-Wesley*, Addison-Wesley Professional, 2012. Citado na página 27.

LANGUAGE, J. P. *The Julia Language*. 2024. Disponível em:<https://docs.julialang.org/en/v1/>. Acessado em: 2024-07-11. Citado na página 28.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California: [s.n.], 2004. Citado na página 22.

LLVM Foundation. *LLVM Language Reference Manual*. 2024. Disponível em:<https://llvm.org/docs/LangRef.html>. Acessado em: 2024-07-09. Citado na página 21.

LOOSEMORE, S. et al. *The GNU C Library Reference Manual*. 2024. Disponível em:<https://sourceware.org/glibc/manual/2.39/>. Acessado em: 2024-06-25. Citado na página 27.

LOPES, B. C.; AULER, R. *Getting started with LLVM core libraries*. [S.l.]: Packt Publishing Ltd, 2014. Citado 3 vezes nas páginas 9, 18 e 22.

Marlin Firmware. *Hardware Abstraction Layer*. 2024. Disponível em:<https://marlinfw.org/docs/development/hal.html>. Acessado em: 2024-07-09. Citado na página 15.

MARTINS, D. R. *Elaboração de uma Linguagem de Programação Específica para Robótica Educacional*. 65 p. Monografia (Monografia) — Universidade Federal de Goiás, Regional Jataí, Jataí, GO, Brasil, 2018. Citado 2 vezes nas páginas 25 e 32.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 37, n. 4, p. 316–344, 2005. Citado na página 24.

Microchip Technology Inc. *GCC Compilers for AVR® and Arm®-Based MCUs and MPUs*. 2024. Disponível em:<https://www.microchip.com/en-us/tools-resources/develop/microchip-studio/gcc-compilers>. Acessado em: 2024-06-28. Citado na página 33.

MICROELECTRONICS, S. *STM32F103x8 datasheet*. 2023. Disponível em:<https://www.st.com/en/microcontrollers-microprocessors/stm32f103c8.html>. Acessado em: 2024-06-06. Citado 2 vezes nas páginas 14 e 23.

OLIVEIRA, T. B. de. *Robcmp: The Robotics Compiler*. 2024. Disponível em:<https://github.com/thborges/robcmp/>. Acessado em: 2024-06-16. Citado 2 vezes nas páginas 16 e 25.

PLATFORMIO. *platformio/ststm32: The STM32 family of 32-bit Flash MCUs based on...* 2024. Disponível em:<https://registry.platformio.org/platforms/platformio/ststm32/frameworks/>. Acessado em: 2024-11-30. Citado na página 34.

PlatformIO Labs OÜ. *PlatformIO Registry*. 2024. Https://registry.platformio.org/. Acessado em: 2024-11-13. Citado na página 32.

PLAUGER, P. J. *The standard C library*. New Jersey: Prentice-Hall, Inc., 1992. Citado na página 27.

RICARTE, I. *Introdução à compilação*. [S.l.]: Elsevier, 2008. Citado 2 vezes nas páginas 19 e 20.

Rust Team. *The Rust Standard Library*. 2024. Disponível em:<https://doc.rust-lang.org/std/index.html>. Acessado em: 2024-07-11. Citado 2 vezes nas páginas 28 e 33.

SANTANA, B. R. d. et al. Robot surgery in brazil. *Research, Society and Development*, v. 11, n. 12, p. e138111233223, Sep. 2022. Disponível em: <https://rsdjournal.org/index.php/rsd/article/view/33223>. Citado na página 14.

TAMASSIAYZ, R. et al. Jdsl: The data structures library in java. *DOCTOR DOBBS JOURNAL*, M AND T PUBLISHING INC, v. 26, n. 4, p. 21–33, 2001. Citado na página 27.

VOLDER, J. E. The birth of cordic. *Journal of VLSI signal processing systems for signal, image and video technology*, Springer, v. 25, n. 2, p. 101–105, 2000. Citado na página 29.