
**Sistema autônomo de irrigação
inteligente**

Pedro Samuel de Melo Vidotti

São Carlos - SP

Sistema autômato de irrigação inteligente

Pedro Samuel de Melo Vidotti

Orientador: Eduardo do Valle Simões

Monografia referente ao projeto de conclusão de curso dentro do escopo da disciplina Projeto de Formatura I (SSC0670) do Departamento de Sistemas de Computação do Instituto de Ciências Matemáticas e de Computação – ICMC-USP para obtenção do título de Engenheiro de Computação.

Área de Concentração: Processamento de Imagens

**USP – São Carlos
Novembro 2017**

“Billy took his pecker out, there in the prision night, and peed and peed on the ground. Then he put it away again, more or less, and contemplated a new problem: Where had he come from, and where should he go now?”

Slaughterhouse Five, Kurt Vonnegut

Dedicatória

Dedico esse trabalho aos meus pais, Tárcio e Márcia Vidotti pelo empenho em fazer com que eu chegasse até aqui e ao Bentinho e Leia, por todo o amor e carinho. Serei sempre grato.

Agradecimentos

Em primeiro lugar, gostaria de agradecer a minha parceira durante toda a graduação, Isabella Zanin Vicente, sem seu incentivo nos momentos difíceis e sua amizade a trajetória até aqui talvez tivesse outro rumo.

Também gostaria de agradecer meus amigos Gabriel Conrado, Rodrigo Oliveira, Danilo Correa e Bruno Makishi por terem me acolhido quando precisei e Gustavo Okuda e Sérgio Baptista por terem me ensinado o verdadeiro significado da palavra lixo.

Agradeço aos meus amigos do intercâmbio Isabela Leme Cruz, Isabella Dall'Asta, Abner Cabral, Derek Chan e Rafael Rigaud por estarem presentes em um ano muito importante para mim pessoalmente e continuarem presentes até hoje.

Agradeço também a minha amiga Elisa Marcatto pelo companheirismo durante esses últimos dois anos e por ter sido tão compreensiva e acolhedora.

Por fim, gostaria de agradecer o professor Eduardo do Valle Simões por me orientar nesse projeto.

Resumo

O projeto desenvolvido para esse Trabalho de Conclusão de Curso consiste em um sistema regido por uma placa Raspberry Pi ligada a um Arduino Uno e sensores. Esse sistema é responsável por tomar a decisão sobre quais regiões de um gramado definido através de uma imagem devem ser irrigadas. A Raspberry é responsável por hospedar o *software* que faz o processamento de imagem e toma a decisão final, enquanto o Arduino é responsável por ler os valores dos sensores e transmiti-los à Raspberry. Essa comunicação é feita através de um script em Python que lê os valores presentes na porta USB da Raspberry. O *software* principal do programa foi desenvolvido em C++ e são utilizados alguns scripts escritos em shell para programar a execução do programa principal. Para a obtenção dos resultados para esse projeto foram feitos testes em um ambiente simulado onde capta-se os valores dos sensores e é gerada uma imagem final mapeando quais regiões devem ser regadas.

Sumário

Lista de Abreviaturas	vii
Lista de Tabelas	viii
Lista de Figuras	ix
CAPÍTULO 1: INTRODUÇÃO	1
1.1. Contextualização e Motivação	1
1.2. Objetivos.....	2
1.3. Organização do Trabalho.....	2
CAPÍTULO 2: REVISÃO BIBLIOGRÁFICA	2
2.1 Considerações Iniciais	3
2.2 Automação Residencial	3
2.3 Processamento de Imagens	5
2.3.1 OpenCV.....	5
2.3.2 Espaço de Cores	8
2.3.3 Segmentação de Imagem.....	11
2.3.4 Processamento Morfológico.....	12
2.4 Topologia de Rede.....	13
2.4.1 Topologia de Rede em Estrela	13
2.5 Internet das Coisas.....	14
2.6 Efeito Fotovoltaico	14
2.7 Considerações Finais	14
CAPÍTULO 3: DESENVOLVIMENTO DO TRABALHO	15
3.1 Considerações Iniciais	15
3.2 Projeto.....	15

3.2.1 Software	16
3.2.2 Hardware	25
3.3 Descrição das Atividades Realizadas	27
3.3.1 Software	27
3.3.2 Hardware	61
3.4. Resultados Obtidos	65
3.4.1 Simulação com Valores dos Sensores (Raspberry Pi)	66
3.4.2 Simulação com Valores Alterados e Imagem Original (Notebook)	67
3.4.2 Simulação com Valores Alterados e Imagem Modificada.....	70
3.5. Dificuldades e Limitações	72
3.6. Considerações Finais	73
CAPÍTULO 4: CONCLUSÃO.....	73
4.1. Contribuições.....	73
4.2. Relacionamento entre o Curso e o Projeto	74
4.3. Considerações sobre o Curso de Graduação.....	74
4.4. Trabalhos Futuros	75
Referências	75

Lista de Abreviaturas

OpenCV	Open Computer Vision
BSD	Berkeley Software Distribution
RANSAC	Random Sample Consensus
RGB	Red Green Blue
HSV	Hue Saturation Value
IOT	Internet of Things
BGR	Blue Green Red
SRAM	Static Random-Access Memory
EEPROM	Electrically-Erasable Programmable Read-Only Memory
LED	Light Emitting Diode
GPU	Graphics Processing Unit
CPU	Central Processing Unit

Lista de Tabelas

Tabela 1 - Especificações do Arduino Uno.....	62
Tabela 2 - Especificações do sensor YL 69.....	63
Tabela 3 - Especificações da Raspberry Pi Model B	64
Tabela 4 - Valores Utilizados em Simulação do Sistema.....	67
Tabela 5 - Quantidade de Água e decisão tomada para os blocos na Figura 81	68
Tabela 6 - Quantidade de Água e decisão tomada para os blocos na Figura 82.	69
Tabela 7 - Quantidade de Água e decisão tomada para os blocos na Figura 83.	70
Tabela 8 - Quantidade de Água e decisão tomada para os blocos na Figura 84.	72

Lista de Figuras

Figura 1- Exemplo da comunicação dos elementos básicos na automação residencial (CASADOMO, 2010).....	4
Figura 2 - Ilustração do processo de stitching	7
Figura 3- Imagem resultado após o stitching.	8
Figura 4 - Representação do modelo RGB com valores entre 0 e 1.....	9
Figura 5 - Representação do modelo RGB com valores entre 0 e 255.....	10
Figura 6 - Hexágono representando o modelo de cores HSV.	11
Figura 7 - Exemplo de uma imagem e o efeito de quatro valores diferente de trehsold.	12
Figura 8 - Representação de uma Topologia de Rede em Estrela.	14
Figura 9 - Fluxograma representando o programa principal da parte de software do projeto	17
Figura 10 - Fluxograma do módulo de Processamento de Imagens do projeto	20
Figura 11 - Fluxograma do módulo de Tomada de Decisão do projeto	22
Figura 12 - Fluxograma do módulo de Aspersores do projeto	24
Figura 13 - Esquemático do Projeto.	26
Figura 14 - Cabeçalho da função stitch.	27
Figura 15 - Pedaço de código da função stitch do módulo de Processamento de Imagens.....	28
Figura 16 - Imagens de entrada para o programa.	28
Figura 17 - Imagem resultado da função stitch.	29
Figura 18 - Cabeçalho da função image_processing.....	29
Figura 19 - Representação da struct map_block.....	30
Figura 20 - Valores dos intervalos de cores utilizados para o processamento de imagens.	31
Figura 21 - Definição de variáveis utilizadas na divisão da imagem em blocos.....	31
Figura 22 - Chamada da função blur.	31
Figura 23 - Imagem resultado após aplicação da função blur.	32
Figura 24 - Chamada da função cvtColor.	32
Figura 25 - Imagem resultado após conversão para espaço de cor HSV.	33
Figura 26 - Chamada da função inRange.	33

Figura 27 - Chamada das funções responsáveis pelo processamento morfológico.....	33
Figura 28 - Imagem resultado do processo de segmentação.	34
Figura 29 - Imagem resultado do processamento morfológico.	34
Figura 30 - Chamada da função calculateAvgPxlColor.	35
Figura 31 - Imagem quadriculada, resultado da função calculateAvgPxlColor.	35
Figura 32 - Chamda da função apply_mask.	36
Figura 33 - Imagem utilizada como máscara pelo módulo de Processamento de Imagens.	36
Figura 34 - Chamada da função mapUnhealthyGrass.	36
Figura 35 - Cabeçalho da função calculateAvgPxlColor.	37
Figura 36 - Laço principal da função calculateAvgPxlColor.	38
Figura 37 - Cálculo do número de linhas e colunas presentes na imagem.	38
Figura 38 - Chamada da função rectangle.	39
Figura 39 - Função responsável por gerar a imagem máscara.	39
Figura 40 - Cabeçalho da função mapUnhealthyGrass.	40
Figura 41 - Laço principal da função mapUnhealthyGrass.	41
Figura 42 - Função calc_dif_cor.....	42
Figura 43 - Cabeçalho da função parse_tempo.	42
Figura 44 - Chamada do sistema para obter a previsão do tempo.	42
Figura 45 - Resultado do commando wttr.in/'Campinas'....	43
Figura 46 - Exemplo de como os coeficientes do tempo são atribuídos.	44
Figura 47 - Exemplo de como é obtida a porcentagem de chuva para um dia.	44
Figura 48 - Cabeçalho da função calcula_coeficiente.	44
Figura 49 - Fórmula do cálculo do coeficiente de chuva.	45
Figura 50 - Cabeçalho da função find_wind.	45
Figura 51 - Chamada do sistema para obter a velocidade do vento.	45
Figura 52 - Expressão Regular utilizada para encontrar a velocidade do vento.	45
Figura 53 - Cabeçalho da função get_vento.	46
Figura 54 - Decisão do valor atribuído à variável que será utilizada na tomada de decisão.	47
Figura 55 - Funções get_umidade e get_insolacao.....	48
Figura 56 - Função get_cor.....	49

Figura 57 - Função formula.....	49
Figura 58 - Máquina de Estados responsável por tomar a decisão de regar o bloco.....	51
Figura 59 - Funções responsáveis por ler e armazenar os resultados obtidos.....	52
Figura 60 - Cabeçalho da função read_sprinklers.....	53
Figura 61 - Struct que representa um sprinkler.....	53
Figura 62 - Determinação de direção do sprinkler.....	54
Figura 63 - Cabeçalho da função get_jato.....	54
Figura 64 - Struct que representa o bloco a ser regado e a quantidade de água.....	55
Figura 65 - Struct que representa um jato.....	55
Figura 66 - Processo de formação do jato.....	56
Figura 67 - Cabeçalho da função distância.....	56
Figura 68 - Cabeçalho da função angulo.....	57
Figura 69 - Fórmula para obter o ângulo.....	57
Figura 70 - Chamadas para obter os valores utilizados na tomada de decisão.....	58
Figura 71 - Chamadas das funções referentes ao Processamento de Imagens.....	58
Figura 72 - Determinação o resultado quando há previsão de muita chuva no dia.....	59
Figura 73 - Laço principal do programa principal.....	59
Figura 74 - Processo para salvar os resultados anteriores.....	60
Figura 75 - Programação de execução do programa principal.....	60
Figura 76 - Ilustração da placa Arduino UNO.....	61
Figura 77 - Ilustração do sensor YL-69.....	62
Figura 78 - Raspberry Pi Model B.....	64
Figura 79 - Painel Solar Fotovoltaico utilizado no Projeto.....	65
Figura 80 - Imagem Resultado do Programa Principal.....	66
Figura 81 - Porção selecionada para demonstrar a simulação sem resultados anteriores.....	67
Figura 82 - Porção selecionada para demonstrar a simulação com resultados anteriores.....	70
Figura 83 - Porção selecionada modificada para demonstrar a simulação sem resultados anteriores.....	70
Figura 84 - Porção selecionada modificada para demonstrar a simulação com resultados anteriores.....	71

CAPÍTULO 1: INTRODUÇÃO

1.1. Contextualização e Motivação

Nos últimos anos, com o surgimento e solidificação de placas de desenvolvimento de aplicações com microprocessadores, como a Raspberry Pi, e microcontroladores, como o Arduino, se tornou muito acessível projetar aplicações de diversas naturezas, entre elas, aplicações de automação residencial. Nessas aplicações, o conceito principal é utilizar a conexão de objetos à internet para captar informações ou então para controlá-los a distância. Esse conceito, segundo (Kopetz, 2011) caracteriza a Internet das Coisas.

Considera-se que no Brasil a cada 100 litros de água tratada, somente 63 são consumidos, que o consumo médio brasileiro é de 166,3 litros por habitante/dia, 51% acima dos 110 litros por habitante/dia recomendados pela ONU¹ e que 72% do consumo de água no Brasil vem da agricultura (Relatório CRHB, 2012), percebe-se que é necessário um controle maior sobre o quanto de água é gasto em atividades de irrigação.

Tendo como principal motivação essa busca por um consumo eficiente que diminua o desperdício de água, surgiu a ideia desse projeto. Com um sistema de irrigação controlado por componentes como a Raspberry Pi, é possível reduzir o excesso de consumo d'água utilizando métricas como insolação, velocidade do vento e umidade do solo. Isso é possível pois a Raspberry é capaz de, com as métricas citadas, calcular a quantidade necessária de água para a área selecionada e comandar um Arduino que seria responsável por guiar o aspersor, eliminando a subjetividade da quantidade de água usada para irrigação numa situação na qual um humano toma a decisão.

¹ <http://www.eosconsultores.com.br/consumo-e-desperdicio-de-agua/>

1.2. Objetivos

Esse projeto busca a criação de um sistema de irrigação inteligente *open source* que possa ser implementado em cenários onde se tenha uma área que deva ser irrigada evitando o desperdício de água. Visando atingir o maior número de pessoas possível, esse projeto foi pensado utilizando componentes de baixo custo e de fácil acesso, além de possuir um manuseio simplificado. Outra ideia importante do projeto é utilizar sempre que possível informações da internet, a fim de obter mais robustez nos resultados, levando em conta o maior número de informação possível.

A decisão de tornar esse projeto *open source* é que o maior número de pessoas o utilize em suas casas ou qualquer outro estabelecimento em que haja uma área a ser irrigada diminuindo a quantidade de água desperdiçada.

Por meio de uma imagem da região a ser irrigada que será utilizada para analisar a saúde do gramado e dos valores de insolação, velocidade do vento, quantidade de água despendida no dia anterior e umidade do solo, espera-se que a quantidade de água gasta na irrigação decidida pela lógica aplicadas não seja insuficiente e que também não caracterize desperdício.

1.3. Organização do Trabalho

Nesse capítulo foi apresentado o objetivo do trabalho e sua motivação. No capítulo 2 encontra-se uma revisão bibliográfica cobrindo todos os conceitos apresentados nesse trabalho assim como a teoria por trás das soluções utilizadas. O capítulo 3 descreve cada módulo do projeto desde sua arquitetura até a sua implementação e apresenta os resultados obtidos. Por fim, no Capítulo 4 encontra-se as conclusões e considerações para trabalhos futuros.

CAPÍTULO 2: REVISÃO BIBLIOGRÁFICA

2.1 Considerações Iniciais

Nesse capítulo, encontram-se os principais conceitos utilizados para o desenvolvimento do projeto, assim como a bibliografia necessária para compreender os problemas que surgiram no decorrer do projeto.

2.2 Automação Residencial

A automação residencial, que também é conhecida como Domótica ou *Smart Home*, é uma área que vem ganhando muita visibilidade nos últimos anos. Segundo (Bolzani, 2004), automação residencial é “um conjunto de tecnologias que ajudam na gestão e execução de tarefas domésticas cotidianas”, já para (WORTMEYER; FREITAS; CARDOSO, 2005), a automação residencial “representa o emprego de tecnologias ao ambiente doméstico (incluindo residências, condomínios, hotéis), com o objetivo de propiciar conforto, praticidade, produtividade, economia, eficiência e rentabilidade, com valorização da imagem do empreendimento e de seus usuários”.

De acordo com (Accardi; Dodonov, 2012), os elementos básicos da automação residencial são: Controladores, Sensores, Atuadores, Barramentos e Interfaces, como pode ser visto na Figura 1.

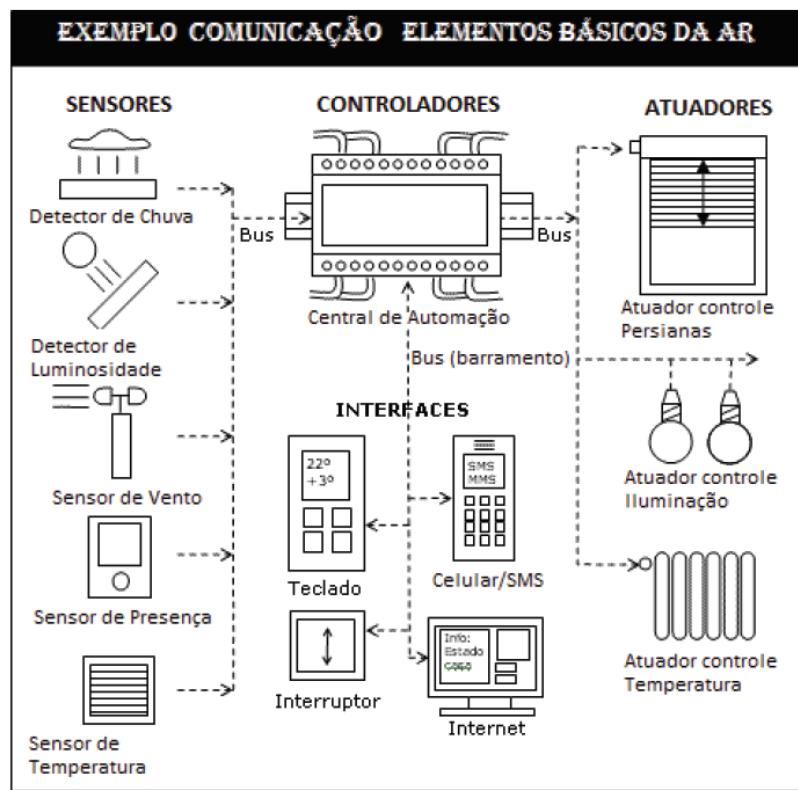


Figura 1- Exemplo da comunicação dos elementos básicos na automação residencial (CASADOMO, 2010)

Controladores, como o Arduino, são os responsáveis por controlar os sensores e atuadores, monitorando as informações dos sensores e enviando comandos para os atuadores. Os sensores são responsáveis por detectar estímulos e medir grandezas físicas e eventos tais como umidade, insolação e temperatura e transformá-los em valores que possam ser manipulados por computadores. Já os atuadores são dispositivos eletromecânicos que vão receber comandos do controlador e ativar equipamentos automatizados (Accardi; Dodonov, 2012).

Os Barramentos são meios físicos por onde a informação será transmitida, por exemplo, módulos *Bluetooth* (Accardi; Dodonov, 2012).

Por fim, as Interfaces são responsáveis por permitir ao usuário visualizar as informações do sistema e interagir com o mesmo (Casadomo, 2010).

2.3 Processamento de Imagens

Processamento de imagem é qualquer tipo de processamento onde a entrada consiste em uma imagem que tem como resultado uma outra imagem ou algum tipo de informação. Essa área vem de Processamento de Sinais, sendo um sinal, assim como uma imagem, um suporte físico que carrega no seu interior uma informação (Albuquerque; Albuquerque, 2001).

Portanto, segundo (Albuquerque; Albuquerque, 2001), processar uma imagem consiste em transformá-la sucessivamente a fim de extrair sua informação mais facilmente.

2.3.1 OpenCV

A ferramenta OpenCV (Open Source Computer Vision Library) é uma biblioteca lançada sob uma licença BSD, o que a torna livre para uso comercial e acadêmico. Ela possui interfaces para as linguagens C, C++, Java e Python e suporta os sistemas operacionais *Windows*, *Linux*, *MacOS*, *iOS* e *Android*.

As principais aplicações da biblioteca OpenCV são na área de Processamento de Imagem e Visão Computacional. Possuindo mais de 2500 algoritmos otimizados², essa biblioteca pode ser utilizada para detectar e reconhecer faces, identificar objetos, juntar imagens em uma panorâmica de alta resolução, modificar imagens e muitas outras aplicações.

2.3.1.1 Stitching

O verbo to stitch, do inglês, em tradução literal significa coser, dar pontos. Neste projeto, o conceito de stitching é unir imagens que possuam um grau de *overlap* (nesse contexto, *overlap* significa características compartilhadas pelas imagens) em uma só. O algoritmo utilizado é implementado na biblioteca OpenCV, versão 3.2.

Ao receber as imagens como entrada, o algoritmo vai extrair as características SIFT (Scale-invariant feature transform), encontrar os k vizinhos mais próximos pra cada característica extraída

² <http://opencv.org/about.html>

usando uma árvore k-d e então, para cada imagem: selecionar m imagens candidatas que possuem o maior número de característica que combinam com a imagem; encontrar combinações de característica que sejam geometricamente consistentes e utilizar RANSAC (método iterativo para estimar parâmetros de um modelo matemático de um conjunto de dados observados que possuam *outliers*) para resolver a homografia entre um par de imagens; encontrar componentes conectados de imagens que combinam e para cada componente conectado: Executar *bundle adjustment* para resolver as rotações $\theta_1, \theta_2, \theta_3$ e comprimento focal f de todas as câmeras e renderizar a panorama usando *multi-band blending* (Brown; Lowe, 2007). A Figura 2 e Figura 3 ilustram o algoritmo apresentado nessa seção.



a) Imagem 1



b) Imagem 2



c) Combinações SIFT 1



d) Combinações SIFT 2



e) RANSAC inliers 1



f) RANSAC inliers 2

Figura 2 - Ilustração do processo de stitching.

Fonte <http://matthewalunbrown.com/papers/ijcv2007.pdf>



g) Imagens alinhadas de acordo com a homografia

Figura 3- Imagem resultado após o stitching.

Fonte <http://matthewalunbrown.com/papers/ijcv2007.pdf>

2.3.2 Espaço de Cores

O espaço de cores é um espaço geométrico de três dimensões com eixos definidos de tal forma que possa representar todas as cores que são percebidas por humanos ou outros animais (Kuehni, 2003). Nesse espaço, cada cor é representada por um ponto.

A razão do espaço de cores ser representado em três dimensões é o fato da cor ser uma sensação criada em resposta a excitação do nosso sistema visual pela luz quando ela incide sobre a retina dos nossos olhos. Como a retina humana possui três tipos de células fotorreceptoras, chamadas de cones, a cor pode ser representada por um vetor de três componentes (Plataniotis; Venetsanopoulos, 2000).

2.3.2.1 Espaço RGB

O espaço de cor RGB é um modelo de cor aditivo baseado nas três cores primárias: Red (Vermelho), Green (Verde) e Blue (Azul). Ele é o modelo mais utilizado em meios digitais pois a combinações de suas cores primárias reproduz a grande maioria do espaço de cores humano.

A representação de uma cor é dada por um número que indica o quanto de cada cor primária está presente. Na Figura 4, os valores de R, G e B estão normalizados e variam entre 0 e 1:

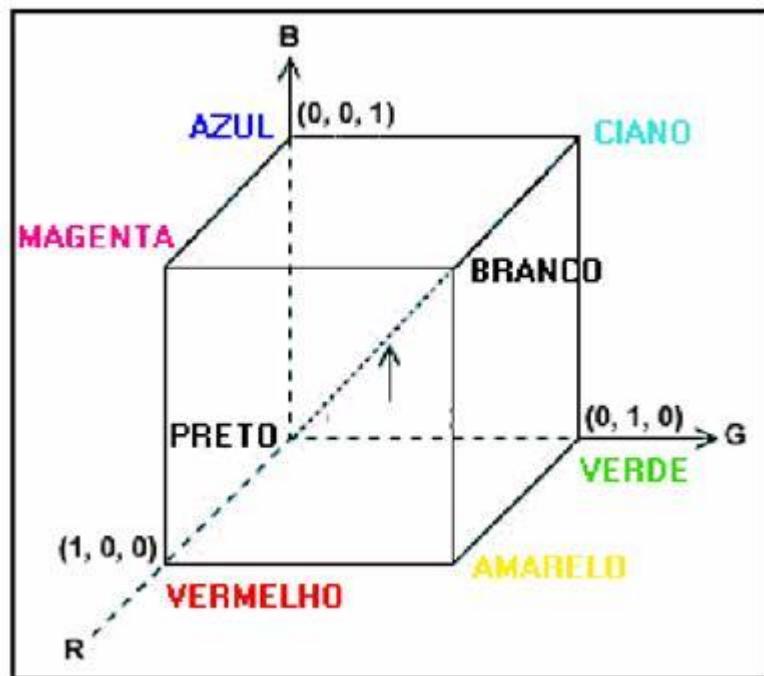


Figura 4 - Representação do modelo RGB com valores entre 0 e 1.

Fonte: <http://www.ufrrgs.br/engcart/PDASR/formcor.html>

Quando pensamos na representação de uma cor no modelo RGB em computadores, o valor de cada cor varia entre 0 e 255, sendo 255 o valor máximo que um byte (8 bits) pode armazenar. Na Figura 5 temos os valores R, G e B variando entre 0 e 255.

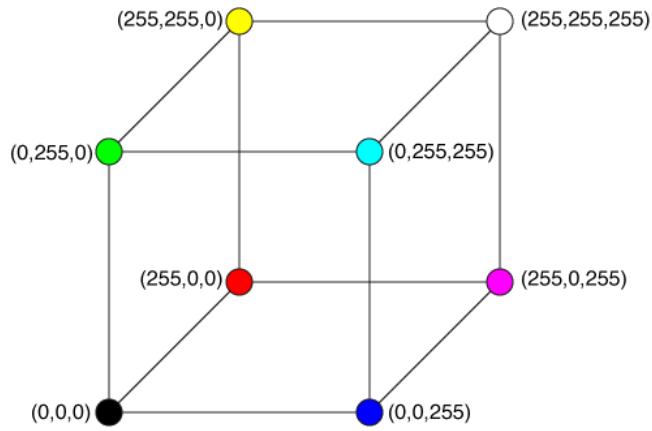


Figura 5 - Representação do modelo RGB com valores entre 0 e 255.

Fonte: <http://www.ufrgs.br/engcart/PDASR/formcor.html>

2.3.2.1 Espaço HSV

Embora o espaço RGB seja o mais utilizado para representações digitais de cores, ele não representa de maneira exata a maneira que as cores são percebidas pelo olho humano. A sigla HSV significa Hue (tonalidade), Saturation (Saturação) e Value (Valor). Uma das vantagens desse modelo é separar informação cromática (Tonalidade e Saturação) da informação acromática (Valor) (Loesdau; Chabrier; Gabillon, 2014).

A Tonalidade é responsável por definir a cor do objeto, ela representa a medida do comprimento de onda médio da luz que ele reflete. Seus valores vão de 0 a 360° ³.

A Saturação representa a “pureza” da cor. Quanto menor o valor, mais acinzentada fica a imagem. E por fim, o Valor representa o brilho da cor. Ambos os valores vão de 0 a 1. A Figura 6 contém representações do espaço apresentado nessa seção.

³ <http://www.ufrgs.br/engcart/PDASR/formcor.html>

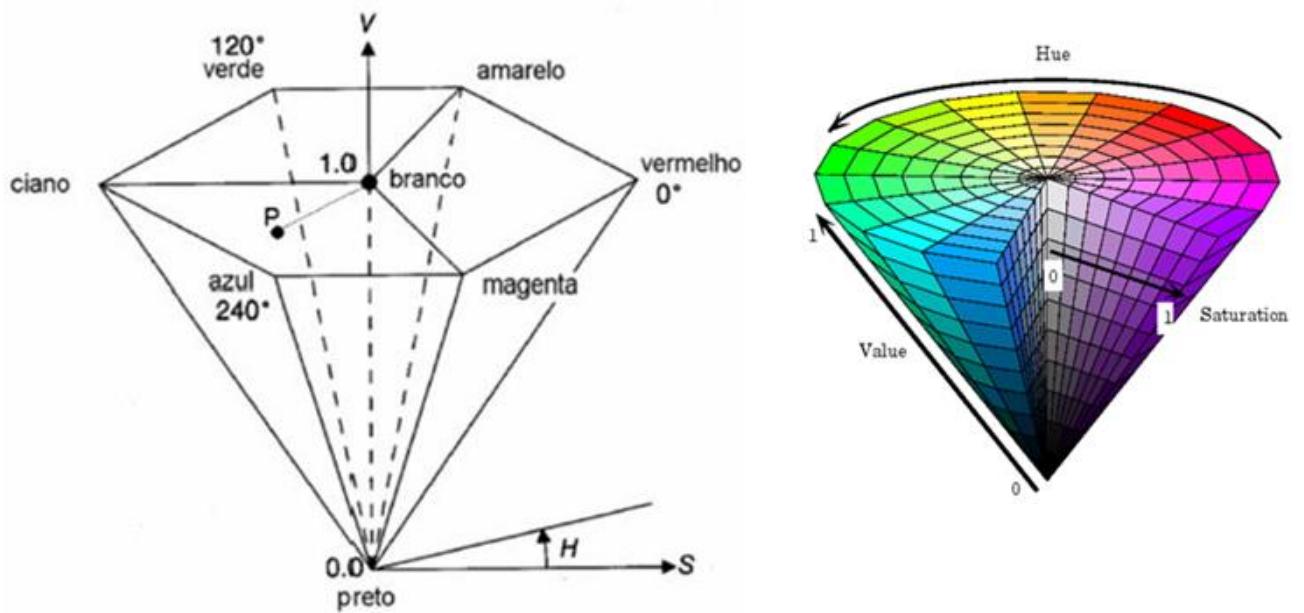


Figura 6 - Hexágono representando o modelo de cores HSV.

Fonte: ENVI (2000) e <https://edoras.sdsu.edu/doc/matlab/toolbox/images/color11.html>

2.3.3 Segmentação de Imagem

Segmentar uma imagem significa particioná-la em regiões distintas onde cada pixel contém atributos similares. O passo de segmentação é muito importante numa análise de imagem.

A forma mais simples de segmentação é o *tresholding*. O *tresholding* consiste em transformar uma imagem colorida (ou em escalas de cinza) em uma imagem binária que pode ser usada como um mapa. Essa imagem possui duas regiões: uma com pixels abaixo do valor de *treshold* e outra com valores iguais ou acima do valor de *treshold*. A maior dificuldade dessa técnica é encontrar o valor ideal de *treshold*. A Figura 7 ilustra o efeito do *tresholding* explicado nessa seção.

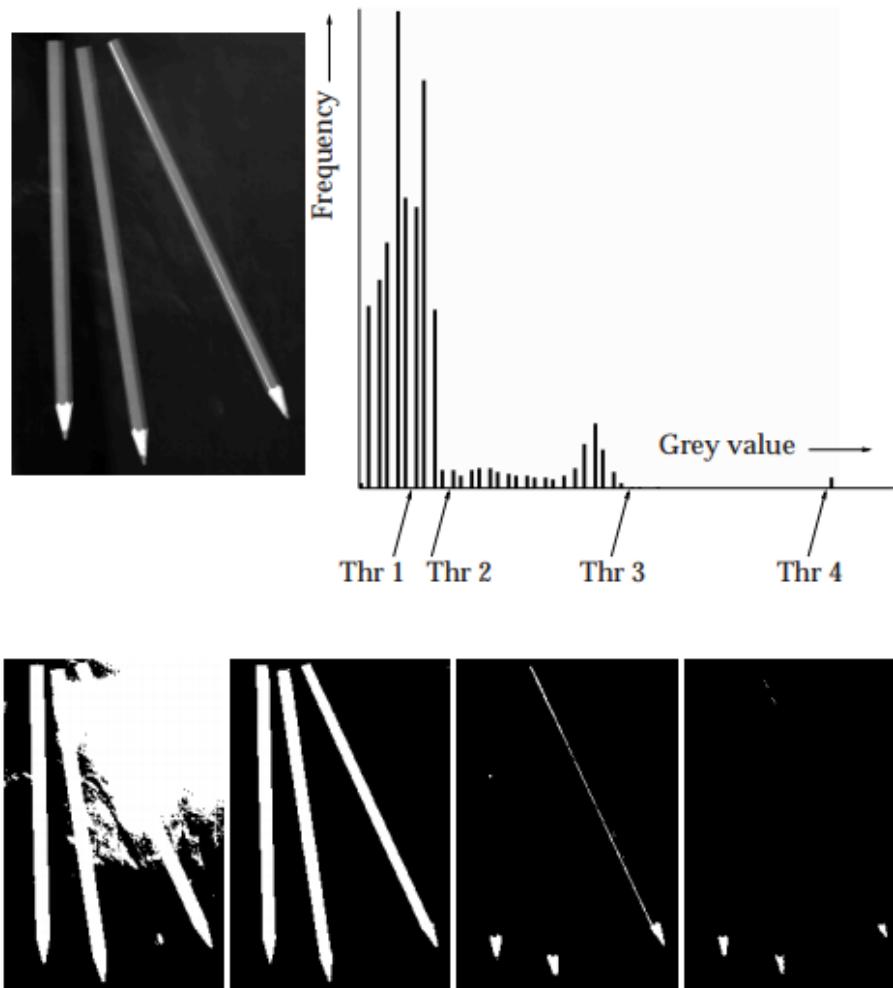


Figura 7 - Exemplo de uma imagem e o efeito de quatro valores diferentes de trehsold.

Fonte: <http://www.cs.uu.nl/docs/vakken/ibv/reader/chapter10.pdf>

2.3.4 Processamento Morfológico

Imagens, ou, como foi definido acima, mapas binários, gerados através do *thresholding* normalmente possuem imperfeições, distorções causadas por ruídos. O processamento morfológico tem como objetivo remover essas imperfeições.

O processamento morfológico em si é um conjunto de operações não-lineares relacionadas à morfologia das características de uma imagem. Segundo (Mathematical Morphology, 2017), as

operações morfológicas baseiam-se na ordem relativa dos pixels e não em seus valores numéricos, o que faz com que sejam ideais para processamento de imagens binárias⁴.

2.4 Topologia de Rede

Topologia de rede é a tecnologia de arranjo de vários elementos como *links*, nós e derivados. Uma topologia de rede representa a estrutura topológica de uma rede de computadores (Santra; Acharjya 2013). Ainda segundo (Santra; Acharjya 2013), hoje existem dois tipos de topologia: Topologia Física e Topologia Lógica. A topologia física foca no *hardware* associado ao sistema como terminais remotos, servidores e o cabeamento entre eles, já a topologia lógica representa o fluxo de dados entre os nós.

2.4.1 Topologia de Rede em Estrela

Na topologia de rede em estrela, há um nó central por onde toda a informação deve passar obrigatoriamente. Esse nó central está conectado a todos os outros nós presentes na rede e é responsável por gerenciar o fluxo de informação de toda a rede. Portanto, se um nó quer se comunicar com outro nó, ele deve primeiro mandar a informação ao nó central que irá repassar para o nó destino a informação.

As principais vantagens de uma topologia de rede em estrela são a facilidade de se inserir um novo componente e a imunidade da rede como um todo a um problema em um dos nós periféricos. Caso um desses nós apresente um problema, o funcionamento da rede não é comprometido.

Já as desvantagens dessa topologia é a limitação para redes pequenas e a dependência do nó central para o funcionamento. A Figura 8 representa uma Topologia de Rede em Estrela:

⁴ <https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm>

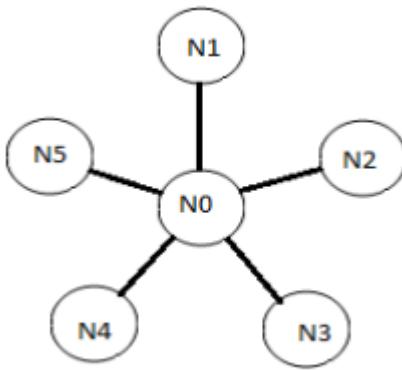


Figura 8 - Representação de uma Topologia de Rede em Estrela.

Fonte: (Santra; Acharjya 2013)

2.5 Internet das Coisas

A Internet das Coisas (IoT) mostra um novo paradigma que conecta uma variedade heterogênea de coisas ao nosso redor à internet e entre elas mesmas (Maalel et al., 2013). Com o surgimento da IoT, veio uma nova geração de objetos com capacidade de se conectar à rede e com habilidades como comunicação e sensoriamento que permitem aplicações como automação residencial, monitoramento de transportações e até na área de saúde (Maalel et al., 2013).

2.6 Efeito Fotovoltaico

A energia solar fotovoltaica é definida como a energia gerada através da conversão direta da radiação solar em eletricidade (Almeida et al., 2016), o efeito fotovoltaico é gerado através da absorção da luz solar, que ocasiona uma diferença de potencial na estrutura do material semicondutor.

2.7 Considerações Finais

Nesse Capítulo foram introduzidos todos os conceitos e teorias necessários para o desenvolvimento desse projeto. No próximo Capítulo será abordado o desenvolvimento do projeto e cada um de seus módulos detalhadamente.

CAPÍTULO 3: DESENVOLVIMENTO DO TRABALHO

3.1 Considerações Iniciais

Nesse Capítulo encontram-se as implementações presentes no projeto tanto de *hardware* quanto de *software*. Na parte de *software*, é explicado o funcionamento dos módulos como um todo e por fim, o funcionamento detalhado de cada função utilizada. Na parte de *hardware* encontra-se o esquema do projeto e as especificações de cada componente. Também estão presentes os resultados obtidos, análises dos resultados e uma análise do projeto como um todo, levando em conta as dificuldades e limitações que surgiram durante o processo.

3.2 Projeto

O objetivo do projeto é desenvolver um sistema de irrigação autônomo que, através de imagens da área a ser irrigada e dados coletados da internet e de sensores, decida quais regiões precisam ser irrigadas e o quanto de água cada uma dessas regiões necessita.

O sistema de irrigação autônomo possui duas frentes igualmente importantes: O *software* que é o responsável por controlar o sistema e tomar a decisão de regar ou não a área e o *hardware*, que é onde o *software* irá ficar hospedado e também responsável pelas ações do sistema.

O *software* é dividido em duas partes, a parte de tomada de decisão, que foi desenvolvida na linguagem C++ e a parte responsável por ler os sensores e transmitir os valores obtidos, que foi desenvolvida na linguagem do Arduino, que é baseada na linguagem C.

O *hardware* consiste na Raspberry Pi, hospedeira do programa responsável pela tomada de decisão, um Arduino, responsável por ler os valores dos sensores e controlar o servo encarregado de girar o aspersor e a válvula d'água.

3.2.1 Software

O programa principal, o que recebe os dados e a imagem e gera a decisão para cada bloco a ser irrigado foi dividido em quatro módulos: Núcleo, onde se encontra a função *main*, que comanda todo o programa; Processamento de Imagem, que é onde a imagem recebida pelo programa é processada a fim de extrair as informações necessárias; Tomada de Decisão, onde se encontram as funções responsáveis por obter os dados da *internet* e dos sensores, assim como a função que salva os resultados em um arquivo e Aspersor, módulo que se encarrega de moldar o jato que vai irrigar o bloco selecionado.

3.2.1.1 Núcleo

No núcleo, todos os outros módulos são chamados através de suas funções. Seu fluxo principal está representado no fluxograma abaixo, na Figura 9:

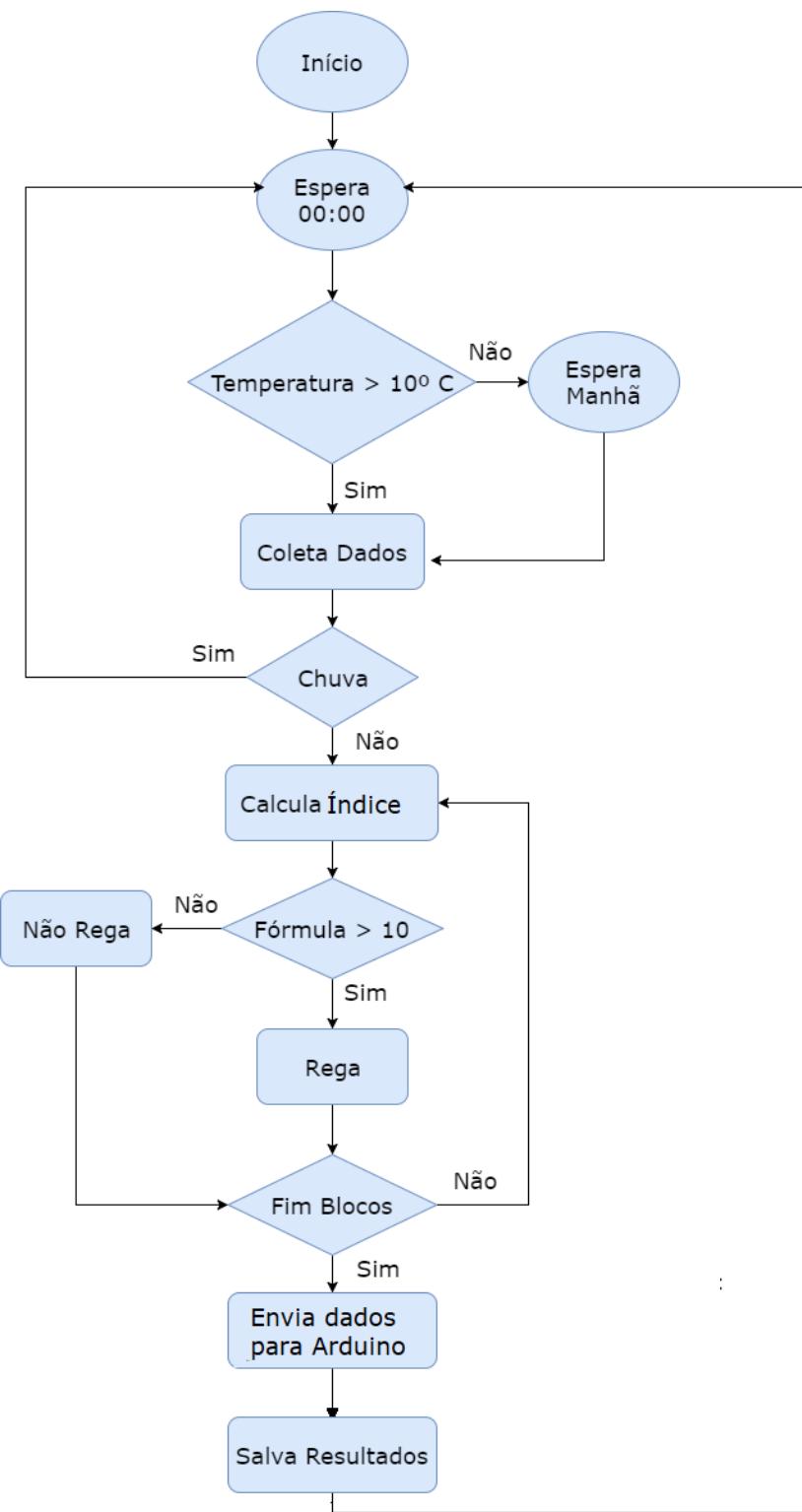


Figura 9 - Fluxograma representando o programa principal da parte de software do projeto

O programa foi projetado como um laço que espera sempre o horário de regar as plantas, que nesse projeto foi definido como meia noite (00:00). Esse horário foi escolhido pois se o jardim for irrigado já com sol, a água seca mais rápido, diminuindo seu aproveitamento. Segundo esse pensamento, o horário escolhido poderia ter sido logo após ao pôr do sol, mas o início da noite foi escolhido pois a incidência de pessoas andando no jardim é menor.

Caso a temperatura esteja muito baixa (abaixo dos 10º C), opta-se por regar as plantas de manhã, evitando que a água congele.

Definido o horário em que as plantas serão regadas, chama-se, então, as funções responsáveis por coletar os dados. Desses valores coletados, o primeiro a ser utilizado é o coeficiente de chuva. Ele representa a probabilidade de chuva contabilizando a previsão do tempo para o dia atual e os próximos dois (ensolarado, nublado, chuvoso, tempo claro e nevando) e quanto maior o seu valor, menor a necessidade de se regar as plantas. Se o número for muito baixo, significa que não há necessidade e então o laço volta para o começo, esperando pelo horário da próxima irrigação.

Após a verificação da possibilidade de chuva, vem o procedimento principal do programa, que é outro laço onde se faz o cálculo da fórmula que diz se o bloco (a imagem com a área total a ser irrigada é dividida em blocos, sendo que para cada um é calculada a fórmula para decidir se eles serão regados ou não) deve ser irrigado ou não. A fórmula leva em conta os valores coletados anteriormente, que são: Insolação, Velocidade do Vento, Umidade do Solo, Coeficiente de Chuva, Cor da Área e o Resultado do dia Anterior. Assim como o coeficiente da chuva, esse cálculo resulta num valor numérico entre 0 e 100 onde 100 é a necessidade máxima de água e de 10 até 0 significa que não é necessário regar aquele bloco.

Feito o cálculo do índice e decidido se o bloco deve ser irrigado ou não, faz-se o mesmo para os próximos até o fim dos blocos. Ao terminar o laço interior, os resultados são salvos num arquivo para serem utilizados no dia seguinte, os dados de controle são enviados para o Arduino e então o programa volta a esperar pela meia noite.

3.2.1.2 Processamento de Imagens

Para a criação desse módulo, foi utilizado como referência o código criado por (Tapparo, 2016). O módulo do programa responsável pelo processamento de imagens é extremamente dependente da biblioteca OpenCV, portanto algumas etapas de seu fluxo são funções implementadas pela biblioteca. O fluxo completo deste módulo encontra-se no fluxograma abaixo, na Figura 10:

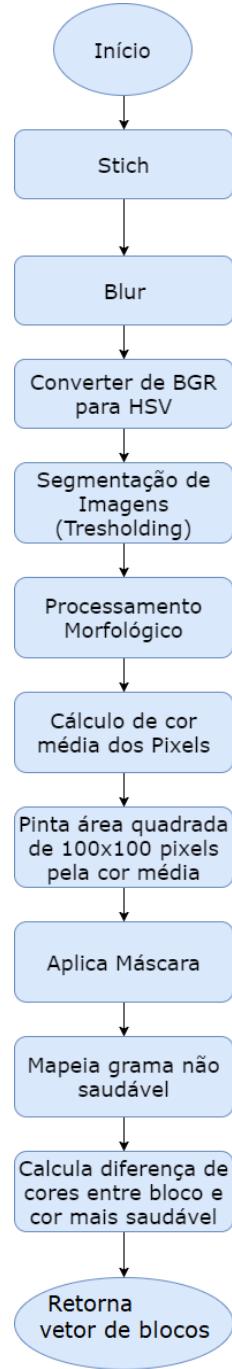


Figura 10 - Fluxograma do módulo de Processamento de Imagens do projeto

O processamento de imagens nesse projeto é uma sucessão de ações em cima das imagens iniciais, resultando em um vetor de blocos que representa as áreas que serão avaliadas para irrigação. Seu fluxo é simples e não possui nenhum laço de repetição ou rota alternativa, essa sequência de ações é chamada no início do programa e só é executada uma vez.

As imagens que chegam como parâmetro para o programa são juntadas através do método de *stitching* e a partir desse resultado são feitas as operações necessárias para extrair a informação desejada da imagem. O *blur*, conversão do espaço BGR (RGB na notação do OpenCV) para HSV, segmentação da imagem e processamento morfológico são todos métodos implementados pela biblioteca do OpenCV e são necessários para manter na imagem somente o que é desejado, que no caso desse projeto é a grama. A partir da imagem gerada por essas operações, são calculados a cor média dos pixels de bloco em bloco, sendo cada bloco um quadrado de lado de 100 *pixels*. Após esse cálculo, todos os pixels dentro da área onde foi calculada a média são “pintados” na cor média. Após essa transformação, é aplicada uma máscara para determinar quais blocos são considerados saudáveis e quais não são. Os blocos considerados não saudáveis, ou blocos que necessitam ser regados, são salvos num vetor que será mandado ao núcleo do programa onde será feito o processo de tomada de decisão.

3.2.1.3 Tomada de Decisão

Assim como no processamento de imagens, o fluxo de tomada de decisão é uma sequência de operações com a diferença que no final existem duas possibilidades: regar ou não regar o bloco. O fluxograma da tomada de decisão pode ser visto na Figura 11:

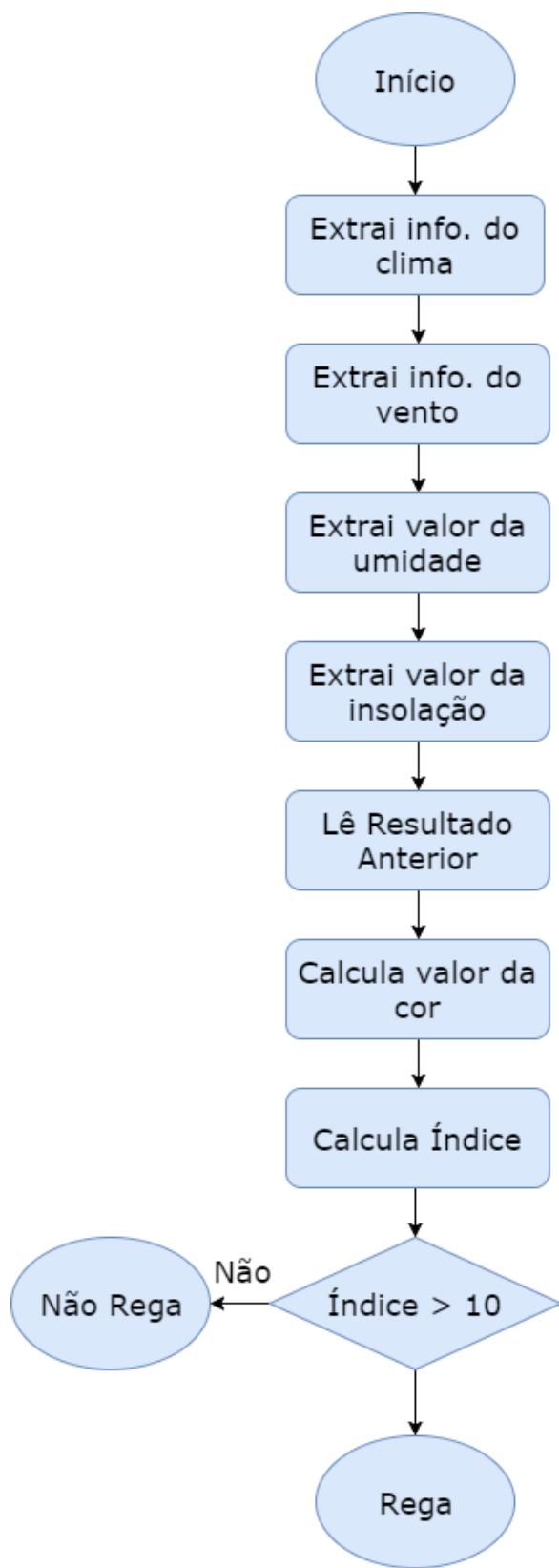


Figura 11 - Fluxograma do módulo de Tomada de Decisão do projeto

As informações do clima e do vento são retiradas da internet através de uma linha de comando, já as informações da umidade do solo e insolação são obtidas através dos respectivos sensores. O resultado anterior é lido de um arquivo onde todos os resultados são salvos, como mostrado no fluxograma do núcleo do programa. Por fim, o valor da cor é o módulo do vetor diferença entre a cor do bloco e a cor que é o valor inferior no intervalo que define se a grama precisa ser regada, ou seja, é o valor da grama menos saudável.

Após a obtenção de todos os valores necessários, faz-se o cálculo da fórmula para decidir se o bloco vai ser regado ou não. A partir da leitura do resultado anterior até o final, o fluxo é o mesmo que o fluxo do *loop* interior encontrado no Núcleo. Isso acontece pois para a tomada de decisão, não existe uma função que englobe todas as outras do mesmo módulo como acontece com o processamento de imagens, portanto, as funções são chamadas individualmente pelo núcleo.

3.2.1.4 Aspersores

O módulo responsável pelos aspersores é o mais simples, sendo dois métodos os mais importantes: O método responsável por ler os aspersores de um arquivo e o método que define os jatos. Esse último detecta o aspersor adequado para o bloco que foi passado como parâmetro e chama todas as funções restantes do módulo: a do cálculo da distância entre aspersor e módulo e a do cálculo do ângulo entre aspersor e módulo. A orientação é determinada na detecção de qual aspersor será utilizado. Seu fluxograma pode ser visto na Figura 12:

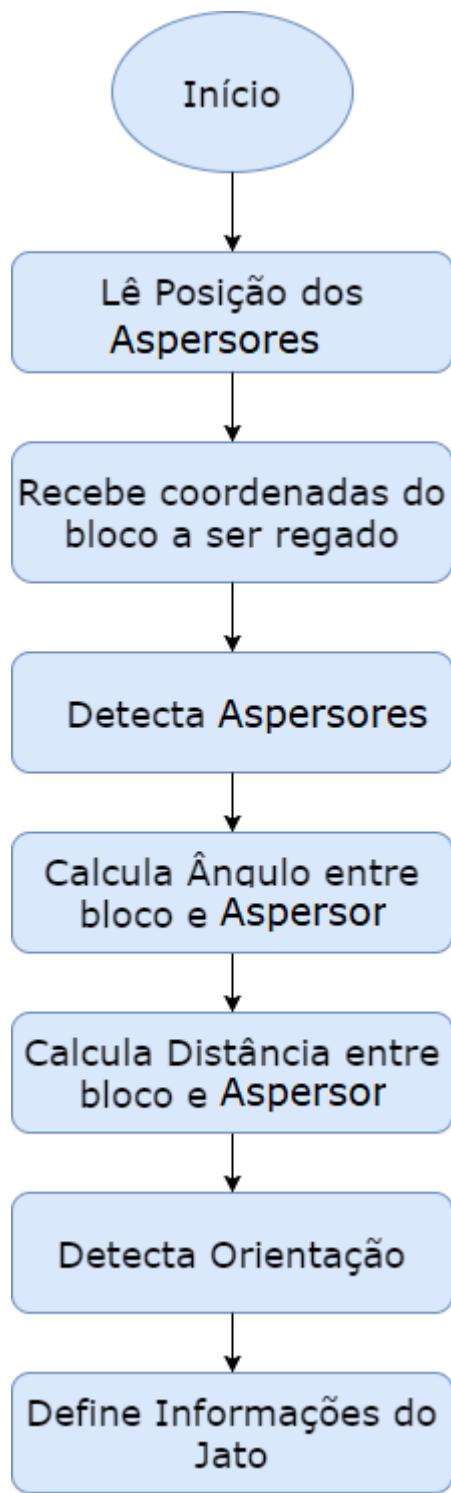
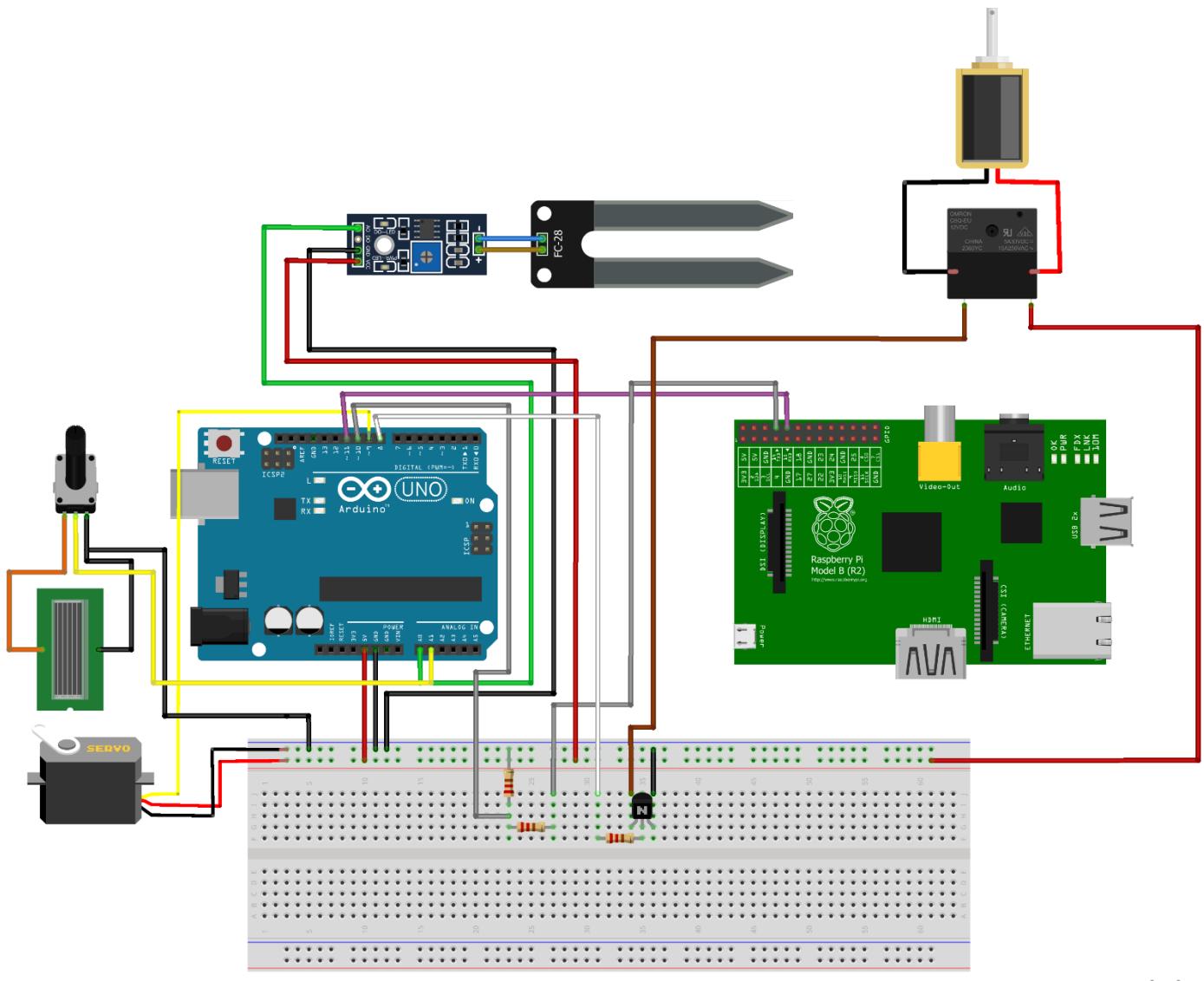


Figura 12 - Fluxograma do módulo de Aspersores do projeto

3.2.2 Hardware

O Hardware proposto para este projeto é um circuito que consiste em uma placa Arduino ligada diretamente numa Raspberry Pi. Todos os sensores utilizados para coletar valores e todos os componentes controlados pelo circuito estão ligados ao Arduino. O esquemático do projeto encontra-se na Figura 13.

Na Figura 13 podemos ver o sensor de umidade e o painel solar ligados ao Arduino nos pinos A0 e A1 respectivamente. O potenciômetro ligado ao painel solar é utilizado para evitar que uma tensão maior que 5 V seja lida pelo Arduino, o que levaria a placa a queimar. O servo e o módulo relé responsável por controlar a válvula solenoide estão ligados nos pinos 9 e 8, respectivamente. O servo é o responsável pela movimentação do aspersor, sendo controlado pelo ângulo enviado ao Arduino e a válvula solenoide, controlada pelo módulo de relé já mencionado, controla a quantidade de água gasta.



fritzing

Figura 13 - Esquemático do Projeto.

3.3 Descrição das Atividades Realizadas

Como foi dito na seção anterior, o projeto possui duas partes: o *software* e o *hardware*. Nesta sessão encontra-se a explicação para as implementações de cada módulo e descrição de cada componente que foi utilizado na implementação final do circuito do projeto.

3.3.1 Software

O software produzido para esse projeto é *open source* e encontra-se em <https://github.com/pedrosmv/TCC/>. O código referente à tomada de decisão, feito na linguagem C++, foi dividido em cinco arquivos, cada um relativo a um módulo, três headers, um para cada módulo menos o núcleo, que não possui função além da *main* e um programa que monitora a temperatura para programar a execução do programa principal. Além dos códigos em C++ responsáveis pelo processo de decisão, são utilizados dois scripts em Python para ler e enviar dados via porta serial entre a Raspberry Pi e o Arduino

3.3.1.1 Processamento de Imagem

3.3.1.1.1 Stitch

A primeira função do módulo de processamento de imagens é *stitch*, como podemos ver na Figura 14:

```
Mat stitch(int argc, char** argv)
```

Figura 14 - Cabeçalho da função *stitch*.

Os seus argumentos, *argc* e *argv* são os mesmos recebidos pela *main* passados via linha de comando. O inteiro *argc* é um valor que indica a quantidade de argumentos que foram passados na chamada do programa. Já *argv* é um vetor do tipo *char* que contém os argumentos. É necessário atentar-se para o fato que *argv[0]* é o nome do programa, portanto os argumentos encontram-se nas posições posteriores.

O tipo de retorno da função, Mat, é um tipo definido pela biblioteca OpenCV, que representa um *array* de canal simples ou multicanal que pode armazenar imagens. No caso da função aqui citada o retorno é uma imagem, união das imagens passadas por argumento.

As imagens presentes no vetor *argv* são lidas e colocadas num vetor de imagens do tipo Mat chamado “*imgs*” e esse vetor será utilizado pelo método *stitch* do objeto *sitcher*, como é possível observar na Figura 15:

```
Mat pano;  
Stitcher stitcher = Stitcher::createDefault(false);  
stitcher.setPanoConfidenceThresh(0.6);  
Stitcher::Status status = stitcher.stitch(imgs, pano);
```

Figura 15 - Pedaço de código da função *stitch* do módulo de Processamento de Imagens.

A imagem resultado, presente em *pano*, é o retorno da função.

Nesse projeto, durante o desenvolvimento, as imagens de entrada foram as imagens presente na Figura 16:



Lado Esquerdo



Lado Direito

Figura 16 - Imagens de entrada para o programa.

Após a chamada da função, o resultado foi a Figura 17:



Figura 17 - Imagem resultado da função stitch.

3.3.1.1.2 Image Processing

Essa função é o método principal do módulo de processamento de imagens. É ela que vai chamar todas as outras funções, exceto a `stitch()`, e extrair o vetor de blocos que será utilizado pelo núcleo do programa. Seu cabeçalho está representado na Figura 18:

```
vector<map_block> image_processing(Mat field, int &max_col, int &max_linha)
```

Figura 18 - Cabeçalho da função `image_processing`.

Esse vetor de blocos, que é o tipo do retorno da função é um vetor da *struct* map_block, determinada no *header* do módulo, imageprocessing.h, sua implementação pode ser vista na Figura 19:

```
struct map_block {  
    int x;  
    int y;  
    bool regado = false;  
    float dif_cor;  
};
```

Figura 19 - Representação da struct map_block.

Nessa struct estão agrupados os valores: x, posição X do bloco no mapa de pixels; y, posição Y do bloco no mapa de pixels; regado, uma variável booleana que diz se o bloco foi regado ou não e dif_cor, o valor da diferença entre a cor do bloco e a cor que é o valor inferior no intervalo que define se a grama precisa ser regada, ou seja, é o valor da grama menos saudável.

Como argumentos, image_processing recebe a imagem gerada pela função stitch() e o endereço de dois ponteiros, max_col representando o número máximo de colunas e max_linha representando o número máximo de linhas. Essas variáveis são utilizadas no módulo do aspersor e, portanto, seus valores são atribuídos no módulo de processamento de imagens, único local onde esse valor é acessível.

A primeira parte dessa função é definir o intervalo de cores que define a área de interesse da imagem, no nosso caso, o gramado (field_range) e o intervalo em que o gramado necessita de água (limites_rgb). Os valores usados nesse projeto foram definidos em (Tapparo, 2016). Na Figura 20 encontra-se os valores utilizados.

```

field_range.min = Scalar(12, 50, 30);
field_range.max = Scalar(80,255,200);

limites_rgb.min = Scalar(13, 80, 60);
limites_rgb.max = Scalar(64, 134, 105);

```

Figura 20 - Valores dos intervalos de cores utilizados para o processamento de imagens.

Após a definição dos intervalos, são definidas algumas variáveis que são utilizadas entre todas as funções, como pode ser visto na Figura 21:

```

quad_dim = 100;
black_pixel_maximum = quad_dim*quad_dim*0.6;
quad_linha = field.rows/quad_dim;
quad_col = field.cols/quad_dim;

```

Figura 21 - Definição de variáveis utilizadas na divisão da imagem em blocos.

Essas variáveis estão relacionadas com a divisão da imagem em blocos, quad_dim é a quantidade de pixels que formam um lado do bloco, black_pixel_maximum define a quantidade máxima de pixels que podem ser pretos dentro de um bloco e quad_linha e quad_col representam o número de blocos no eixo Y e X da imagem respectivamente.

O próximo passo da função é aplicar as operações necessárias para poder extrair da imagem as informações desejadas. A primeira parte utiliza apenas funções implementadas pelo OpenCV, resultando em uma imagem em preto e branco onde o branco é gramado e o preto é o que não deve ser considerado nas próximas operações.

A primeira dessas funções é a blur, usada para suavizar a imagem. Sua chamada pode ser vista na Figura 22:

```
blur(field, field, Size(2,2));
```

Figura 22 - Chamada da função blur.

O resultado pode ser visto na Figura 23:



Figura 23 - Imagem resultado após aplicação da função blur.

Após suavizar a imagem, é feita a conversão do espaço de cores BGR (RGB na notação do OpenCV) para HSV. Essa conversão é feita chamando a função cvtColor, como na Figura 24:

```
cvtColor(field, hsv_field, CV_BGR2HSV);
```

Figura 24 - Chamada da função cvtColor.

O resultado pode ser visto na Figura 25:

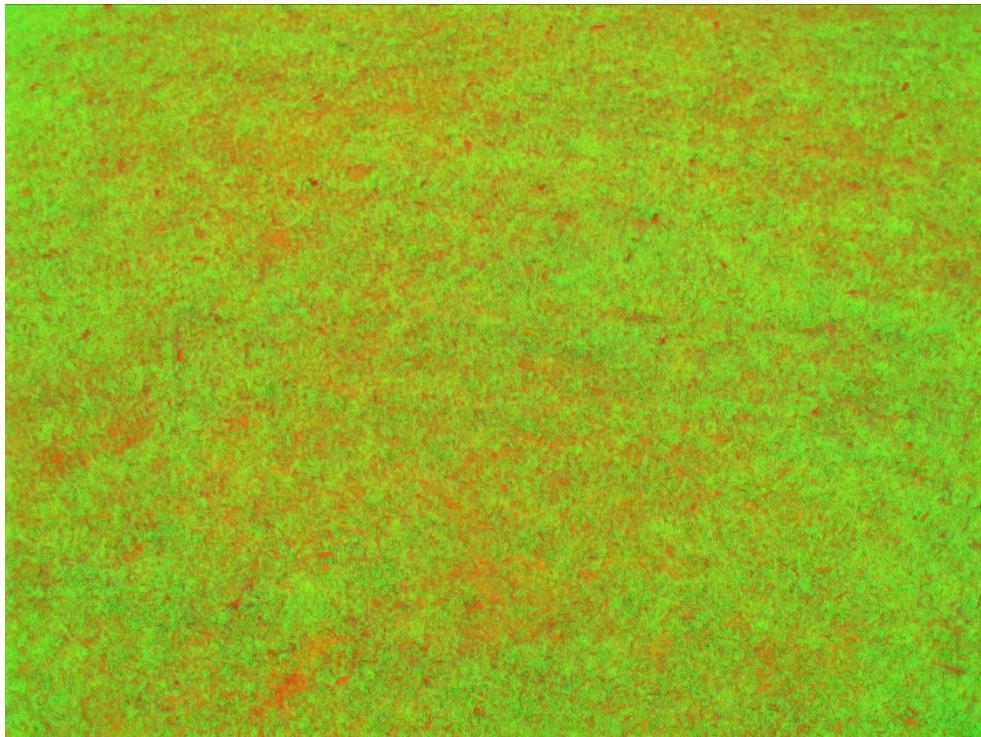


Figura 25 - Imagem resultado após conversão para espaço de cor HSV.

Com a imagem no espaço de cor HSV, é feita a segmentação na imagem, usando como intervalo de *threshold* o intervalo *field_range*, que vai delimitar o que é gramado e o que não é. A segmentação é feita através da função *inRange*, como pode ser visto na Figura 26:

```
inRange(hsv_field, field_range.min, field_range.max, field_threshold);
```

Figura 26 - Chamada da função *inRange*.

Como na imagem utilizada para teste não há nada além de gramado, o resultado, que pode ser visto na Figura 28, não apresenta uma região em preto, apenas pontos, que serão corrigidos pelo processo morfológico. O processamento morfológico é chamado como na Figura 27:

```
element = getStructuringElement(MORPH_RECT, Size(9, 9), Point(4,4));
morphologyEx(field_threshold, processed_field, MORPH_CLOSE, element);
```

Figura 27 - Chamada das funções responsáveis pelo processamento morfológico.

O processo morfológico encarrega-se de eliminar as imperfeições do *thresholding*, resultando na Figura 29.



Figura 28 - Imagem resultado do processo de segmentação.

Figura 29 - Imagem resultado do processamento morfológico.

Após o processamento morfológico, a imagem está pronta para ser processada a fim de fornecer as informações desejadas. A partir da Figura 28, é chamada a função squared_field, como na Figura 30:

```
squared_field = calculateAvgPxlColor(final_field, quad_dim, quad_linha, quad_col, black_pixel_maximum);
```

Figura 30 - Chamada da função calculateAvgPxlColor.

Essa função vai calcular a média das cores dos pixels numa área de 100x100 pixels e pintar todos os pixels dentro dessa área da cor média. O resultado é uma imagem quadriculada, como na Figura 31:



Figura 31 - Imagem quadriculada, resultado da função calculateAvgPxlColor.

Nessa imagem, utilizando o intervalo determinado em limites_rgb, será feita outra segmentação, a fim de gerar uma máscara que separará os blocos que devem ser regados daqueles que não devem ser regados.

A máscara é gerada pela função mask_field, que é chamada como na Figura 32:

```
mask_field = apply_mask(squared_field, limites_rgb);
```

Figura 32 - Chamada da função apply_mask.

O resultado é uma imagem onde os blocos que devem ser regados estão em preto e os blocos considerados saudáveis estão em branco. Após copiar os pixels brancos para uma outra imagem, o resultado é a Figura 33:



Figura 33 - Imagem utilizada como máscara pelo módulo de Processamento de Imagens.

Por fim, a função que gera o vetor de blocos que será retornado pela image_processing é a mapUnhealthyGrass, que tem a chamada representada na Figura 34:

```
mapUnhealthyGrass(squared_field, mask_field, quad_dim, quad_linha, quad_col, mapBlock, limites_rgb);
```

Figura 34 - Chamada da função mapUnhealthyGrass.

Essa função utiliza a Figura 31 e Figura 33 para dizer se o bloco está saudável ou não. Se o bloco é considerado não saudável, é criado um bloco do tipo map_block que é salvo no vetor de blocos. Terminada a execução dessa função, termina a execução da image_processing e o vetor de blocos é retornado.

3.3.1.1.3 Cálculo da Média de Cores do Pixel

A função calculateAvgPxlColor é responsável por transformar a imagem que passou pelo processamento morfológico na imagem quadriculada vista na figura 30. Seu cabeçalho está representado na Figura 35:

```
Mat calculateAvgPxlColor(Mat final_field, int quad_dim, int quad_linha, int quad_col, int black_pixel_maximum);
```

Figura 35 - Cabeçalho da função calculateAvgPxlColor.

Seus argumentos são: final_field, imagem que passou pelo processamento morfológico; quad_dim, dimensão do lado do quadrado que vai ser pintado com a cor média dos pixels dentro de sua área; quad_linha representa o número de blocos no eixo Y da imagem; quad_col representa o número de blocos no eixo X da imagem e black_pixel_maximum é o máximo de pixels que podem ser pretos dentro de um bloco. O retorno da função é um objeto do tipo Mat, classe do OpenCV usada para representar uma imagem.

O núcleo dessa função é um laço que itera por todos os blocos da imagem. Dentro desse laço principal, para percorrer a imagem passada por parâmetro, são usados dois laços *for* encadeados, como pode-se ver na Figura 36:

```

for(bgr_array_linha = (quad_dim*num_quad_linha); bgr_array_linha < (quad_dim*(num_quad_linha+1)); bgr_array_linha++) {
    for(bgr_array_col = (quad_dim*num_quad_col); bgr_array_col < (quad_dim*(num_quad_col+1)); bgr_array_col++) {
        if( final_field.at<Vec3b>(bgr_array_linha, bgr_array_col) != Vec3b(0,0,0) &&
            final_field.at<Vec3b>(bgr_array_linha, bgr_array_col) != Vec3b(255,255,255)) {
            blue_color += final_field.at<Vec3b>(bgr_array_linha, bgr_array_col)[0];
            green_color += final_field.at<Vec3b>(bgr_array_linha, bgr_array_col)[1];
            red_color += final_field.at<Vec3b>(bgr_array_linha, bgr_array_col)[2];
            avg_color++;
            not_black++;
        }
    }
}

```

Figura 36 - Laço principal da função calculateAvgPxlColor.

Nesses dois laços, as variáveis de iteração bgr_array_linha e bgr_array_col vão iterar sobre os pixels de um bloco. Começando em 0, num_quad_linha e num_quad_col são alteradas respectivamente fora desses laços interiores, mas dentro do laço principal, após a execução da função presente na Figura 38, como podemos ver na Figura 37:

```

    num_quad++;
    num_quad_col = num_quad%quad_col;
    if(num_quad_col == 0) {
        num_quad_linha++;
    }
}

```

Figura 37 - Cálculo do número de linhas e colunas presentes na imagem.

Com isso, a cada iteração do laço principal, os laços interiores percorrem 100 pixels no eixo x e 100 pixels no eixo y.

Dentro dos dois laços interiores, é verificado se o pixel não é branco nem preto, e então soma-se o valor de cada cor (verde, vermelho e azul) em sua respectiva variável. As variáveis avg_color e not_black servem para contar quantos pixels foram contabilizados e se eles não eram pretos, respectivamente.

Saindo dos laços interiores, são feitas duas verificações antes de pintar o bloco com a cor média: se o número de pixels não pretos é menor ou igual que o máximo de pixels pretos permitidos ou se ele é zero e se o número de pixels contabilizados foi 0. Essas verificações são feitas para evitar erros na hora de calcular a cor média.

Por fim, a modificação na imagem é feita através da função rectangle da biblioteca OpenCV, como podemos ver na Figura 38:

```
rectangle(squared_field, Point(0+(quad_dim*num_quad_col),(0+(quad_dim*num_quad_linha))),Point((quad_dim*(num_quad_col+1))-1,(quad_dim*(num_quad_linha+1))-1),  
Scalar(blue_color/avg_color,green_color/avg_color,red_color/avg_color),-1,8);
```

Figura 38 - Chamada da função rectangle.

3.3.1.1.4 Aplicação da Máscara

A função responsável por aplicar a máscara que irá separar os blocos saudáveis dos não saudáveis é um encapsulamento da função inRange da biblioteca OpenCV. Como pode-se ver na Figura 39:

```
Mat apply_mask(Mat squared_field, range limites_rgb){  
    Mat int_mask, final_mask;  
  
    inRange(squared_field, limites_rgb.min, limites_rgb.max, int_mask);  
    squared_field.copyTo(final_mask, int_mask);  
  
    return final_mask;  
}
```

Figura 39 - Função responsável por gerar a imagem máscara.

A função apply_mask recebe como parâmetro a imagem quadriculada e o intervalo que delimita as cores que representam um bloco saudável e retorna uma imagem segmentada onde os blocos que precisam ser irrigados estão em preto, como pode ser visto na Figura 33.

3.3.1.1.5 Mapeamento de Gramado Não Saudável

A função responsável por mapear o gramado não saudável recebe como parâmetro a imagem do jardim já transformada em uma imagem quadriculada, a imagem da máscara, o tamanho do lado de cada bloco, o número de linhas e de colunas da imagem, o endereço do vetor de blocos que será retornado pela função `image_processing` e o intervalo que representa uma cor saudável. Sua chamada pode ser vista na Figura 40:

```
void mapUnhealthyGrass(Mat field, Mat field_mask, int quad_dim, int quad_linha, int quad_col, vector<map_block> &mapBlock, range limites_rgb)
```

Figura 40 - Cabeçalho da função `mapUnhealthyGrass`.

Assim como a função que calcula a cor média dos pixels, `mapUnhealthyGrass` é composta por um laço principal e dois laços interiores encadeados. O laço principal percorre cada bloco da imagem, já os dois laços interiores percorrem os eixos X e Y da imagem. Para cada bloco, é verificado na imagem quadriculada e na máscara se o bloco precisa ser irrigado ou não. Caso seja necessário, salva-se as informações do bloco em um objeto do vetor de blocos, como pode ser visto na Figura 41 (as variáveis x e y da função representam as coordenadas do centro de cada bloco):

```

while(num_quad < quad_linha*quad_col) {
    for(row = 0; row < quad_linha; row++) {
        for(col = 0; col < quad_col; col++) {
            if(field.at<Vec3b>(y, x) != Vec3b(0,0,0)) {
                if(field_mask.at<Vec3b>(y, x) == Vec3b(0,0,0)) {
                    mapBlock.push_back(map_block());
                    mapBlock[i].x = x;
                    mapBlock[i].y = y;
                    mapBlock[i].regado = true;
                    b = field.at<Vec3b>(y, x)[0];
                    g = field.at<Vec3b>(y, x)[1];
                    r = field.at<Vec3b>(y, x)[2];
                    mapBlock[i].dif_cor = calc_dif_cor(b, g, r, limites_rgb);
                    i++;
                }
            }
            num_quad++;
            x += quad_dim;
        }
        x = (quad_dim/2);
        y += quad_dim;
    }
}

```

Figura 41 - Laço principal da função mapUnhealthyGrass.

Como o argumento da função é o endereço do vetor de blocos e não uma cópia dele, não é necessário retornar nada, portanto, a função é do tipo *void*. Ao fim do laço principal, o vetor estará preenchido e o módulo de processamento de imagem se encerra.

3.3.1.1.6 Cálculo da Diferença de Cor

A função *calc_dif_cor*, vista na Figura 41, é responsável por calcular a diferença entre a cor de cada bloco e o ponto mínimo do intervalo que define as cores de um gramado saudável. O ponto mínimo foi escolhido pois para essa função ser chamada, a cor do bloco está dentro do intervalo e ela está abaixo da cor que é considerada saudável, portanto, optou-se por usar a cor menos saudável como referência.

O cálculo é uma conta de diferença de vetores, sendo o resultado o módulo do vetor diferença, como visto na Figura 42:

```

float calc_dif_cor(int blue, int green, int red, range limites_rgb){
    float vetor_dif[3];
    float dif;
    vetor_dif[0] = limites_rgb.min[0] - blue;
    vetor_dif[1] = limites_rgb.min[1] - green;
    vetor_dif[2] = limites_rgb.min[2] - red;

    return dif = sqrt(pow(vetor_dif[0], 2) + pow(vetor_dif[1], 2) + pow(vetor_dif[2], 2));
}

```

Figura 42 - Função calc_dif_cor.

3.3.1.2 Tomada de Decisão

3.3.1.2.1 Captação da Previsão do Tempo

Um dos valores utilizado para calcular a quantidade de água necessária para um bloco é o coeficiente de chuva. Esse coeficiente, que tem valor entre 0 e 1, é calculado usando valores captados do site wttr.in referentes à previsão do tempo e probabilidade de chuva do dia atual e os dois seguintes. A chamada da função pode ser vista na Figura 43:

```
float parse_tempo();
```

Figura 43 - Cabeçalho da função parse_tempo.

Esses valores são obtidos através de uma chamada do sistema que faz uma requisição ao site via linha de comando, como pode ser visto na Figura 44:

```
system("curl wttr.in/'Campinas' > tempo.txt");
```

Figura 44 - Chamada do sistema para obter a previsão do tempo.

O resultado, salvo no arquivo, segue o modelo da Figura 45:

<p>Parcialmente nublado 24-26 °C ~ 13 km/h 19 km 0.0 mm</p>			
Manhã	Meio-dia	Sáb 07 Out	Tarde
<p>Possibilidade de chuva 23-24 °C ~ 12-18 km/h 19 km 0.0 mm 20%</p>	<p>Possibilidade de chuva 25-26 °C ~ 15-18 km/h 17 km 0.0 mm 7%</p>	<p>Parcialmente nublado 24-26 °C ~ 13-18 km/h 19 km 0.0 mm 0%</p>	<p>Parcialmente nublado 22-25 °C ~ 15-23 km/h 19 km 0.0 mm 0%</p>
Manhã	Meio-dia	Dom 08 Out	Noite
<p>Parcialmente nublado 22-23 °C ~ 17-23 km/h 20 km 0.0 mm 0%</p>	<p>Sol 28-29 °C ~ 16-22 km/h 20 km 1.3 mm 0%</p>	<p>Parcialmente nublado 29 °C ~ 7-11 km/h 20 km 0.0 mm 0%</p>	<p>Parcialmente nublado 26-27 °C ~ 7-13 km/h 19 km 0.0 mm 0%</p>
Manhã	Meio-dia	Seg 09 Out	Noite
<p>Sol 23-24 °C ~ 20-26 km/h 19 km 0.0 mm 0%</p>	<p>Sol 29 °C ~ 17-19 km/h 20 km 0.0 mm 0%</p>	<p>Sol 30-31 °C ~ 5-7 km/h 20 km 0.0 mm 0%</p>	<p>Céu limpo 27 °C ~ 6-12 km/h 20 km 0.0 mm 0%</p>

Figura 45 - Resultado do comando `wttr.in/'Campinas'`.

Para gerar os valores que serão utilizados na função do cálculo do coeficiente de chuva, é necessário analisar o arquivo de texto onde o resultado está salvo utilizando o método `find` da classe `string`, própria da linguagem C++.

Essa análise é feita num laço que itera pelo arquivo doze vezes, procurando pelas palavras *Sunny*, *Clear*, *Cloudy*, *Overcast*, *Rain* e *Snow*. No programa as palavras estão em inglês em decorrência da linguagem padrão do sistema operacional onde o projeto foi desenvolvido. Cada uma dessas palavras possui um valor que é inserido no vetor de coeficientes do clima. Esses valores são:

- *Sunny* (Ensolarado) = 3
- *Clear* (Tempo Claro) = 2
- *Cloudy/Overcast* (Nublado) = 1
- *Rain* (Chuva) = 0
- *Snow* (Neve) = 0

Esses valores irão formar o vetor de coeficientes do clima que serão utilizados no cálculo do coeficiente de chuva. Por fim, para cada parte do dia é salvo em um vetor a probabilidade de chuva, que também será utilizada no cálculo do coeficiente de chuva.

Nas Figura 46 e Figura 47 abaixo, encontra-se um exemplo de como é atribuído o valor do coeficiente do clima ao vetor e como é formado o vetor de probabilidade de chuva:

```
size_t sunny = stream.find("Sunny", 0);
if(sunny != string::npos)
{
    tempo_coeficiente[j] = 3;
    j++;
}
```

Figura 46 - Exemplo de como os coeficientes do tempo são atribuídos.

```
size_t found = stream.find('%');
if(found != string::npos)
{
    float aux;
    aux = stof(stream);
    chuva_porcentagem[i+1] = aux;
    i++;
}
```

Figura 47 - Exemplo de como é obtida a porcentagem de chuva para um dia.

3.3.1.2.2 Cálculo do Coeficiente de Chuva

O cálculo do coeficiente de chuva é feito utilizando os valores referentes à previsão do tempo e probabilidade de chuva. Como podemos ver na Figura 48:

```
float calcula_coeficiente(int tempo[], float chuva[])
```

Figura 48 - Cabeçalho da função calcula_coeficiente.

A função recebe como parâmetros dois vetores, um referente à previsão do tempo e outro à probabilidade de chuva. Utilizando os valores presentes nesses vetores, calcula-se um coeficiente com valor entre 0 e 1 que mede a probabilidade de ser necessário regar as plantas. Quanto maior o valor do coeficiente, menos chuva está previsto.

A fórmula para o cálculo do coeficiente está presente na Figura 49:

```
for (i=0; i<12; i++) {  
    coeficiente = coeficiente + (tempo[i] * (1 - (chuva[i]/100)));  
}
```

Figura 49 - Fórmula do cálculo do coeficiente de chuva.

3.3.1.2.3 Captação da Velocidade do Vento

A função responsável por captar a velocidade do vento é uma função sem parâmetros que retorna um valor do tipo *float*, como podemos ver na Figura 50:

```
float find_wind()
```

Figura 50 - Cabeçalho da função *find_wind*.

Assim como as informações do clima, a velocidade do vento é obtida através de uma chamada do sistema que executa o programa *ansiweather*, como pode-se ver na Figura 51:

```
system("ansiweather -l Campinas > vento.txt");
```

Figura 51 - Chamada do sistema para obter a velocidade do vento.

O resultado é salvo num arquivo que vai ser lido pelo programa e atribuído a uma string. Nessa string será feita uma busca utilizando expressão regular para achar a velocidade do vento. A expressão regular utilizada pode ser vista na Figura 52:

```
"[0-9]?[0-9]?(\.[0-9][0-9]?)? m/s"
```

Figura 52 - Expressão Regular utilizada para encontrar a velocidade do vento.

Essa expressão regular busca por números decimais ou inteiros que sejam seguidos por m/s. O valor captado pela busca é então convertido para um número do tipo *float*, multiplicado por 3.6 para obter o valor em km/h e então é retornado pela função.

3.3.1.2.4 Atribuição do Valor do Vento

Para cada elemento utilizado no cálculo do índice de necessidade de água, existe uma função responsável por captar e atribuir os seus valores. No caso do vento, essa função se chama `get_vento`, como é visto na Figura 53:

```
int get_vento()
```

Figura 53 - Cabeçalho da função `get_vento`.

A função `get_vento` chama a função `find_wind` e de acordo com a velocidade do vento retornada pela função atribui-se um valor a resultado que varia entre 0 e 5, sendo que 0 significa um vento muito forte, situação em que não compensa regar o jardim, e 5 representa a velocidade máxima aceitável para se regar o jardim. Esse valor presente em resultado será o valor utilizado no cálculo do índice de necessidade de água. A divisão é feita seguindo o modelo da Figura 54:

```

int get_vento(){

    int vento;
    float resultado;

    resultado = find_wind();

    if (resultado <= 5) {
        vento = 1;
    }
    else if (resultado > 5 && resultado <= 10) {
        vento = 2;
    }
    else if (resultado > 10 && resultado <= 15) {
        vento = 3;
    }
    else if (resultado > 15 && resultado <= 20) {
        vento = 4;
    }
    else if (resultado > 20 && resultado <= 25) {
        vento = 5;
    }
    else{
        vento = 0;
    }

    return vento;
}

```

Figura 54 - Decisão do valor atribuído à variável que será utilizada na tomada de decisão.

Essa divisão foi feita baseada no que diz (Kobiyama; Chaffe): “O vento modifica a camada de ar vizinho a superfície, substituindo uma camada muitas vezes saturada por uma com menor teor de vapor da água. Portanto, quanto maior a intensidade do vento, maior a intensidade de evaporação.”. Portanto, quanto maior a velocidade do vento, maior é a necessidade de água. Essa relação pode ser vista na Figura 57.

3.3.1.2.5 Atribuição do Valor da Umidade e Insolação

Para obter os valores de umidade e insolação é utilizado um script em Python que lê os valores dos sensores conectados ao Arduino. O script consiste em uma leitura da porta serial e em

salvar os valores lidos em um arquivo, chamado sensordata.txt. Esse arquivo será lido pelas funções get_umidade e get_insolacao no programa principal. Na Figura 55 encontra-se a implementação das funções.

```
int get_umidade(){

    string stream;
    ifstream file;
    string aux;
    int umidade_valor;

    file.open("sensordata.txt");

    while (getline (file, stream)) {

        stringstream line(stream);
        size_t found = stream.find("Umidade", 0);
        if(found != string::npos)
        {
            line >> aux >> umidade_valor;
        }

    }
    return ((umidade_valor*4)/1023);
}

float get_insolacao(){

    string stream;
    ifstream file;
    string aux;
    float insolacao_valor;

    file.open("sensordata.txt");

    while (getline (file, stream)) {

        stringstream line(stream);
        size_t found = stream.find("Insolacao", 0);
        if(found != string::npos)
        {
            line >> aux >> insolacao_valor;
        }

    }
    return insolacao_valor;
}
```

Figura 55 - Funções get_umidade e get_insolacao.

3.3.1.2.6 Atribuição do Valor da Cor

O valor numérico atribuído à cor do bloco é um cálculo feito utilizando o resultado da função calc_dif_cor, vista na figura M. Esse resultado é dividido pela diferença máxima do intervalo determinado na função image_processing (resultado da diferença entre o ponto mínimo do intervalo e o ponto máximo), gerando um coeficiente que é multiplicado pelo valor 80. Esse valor foi escolhido para complementar o resultado da equação entre a umidade do solo, insolação e velocidade do vento, que tem como valor máximo 20. Caso o bloco não tenha necessidade de ser regado, não é feito o cálculo e o retorno é 0. A implementação da função pode ser vista na Figura 56:

```
int get_cor(bool regado, float dif_cor){

    if(regado) {
        return 80*((dif_cor/142));
    }

    return 0;
}
```

Figura 56 - Função get_cor.

3.3.1.2.7 Cálculo do Índice de Necessidade de Água

Para o cálculo do índice de necessidade de água, a função formula recebe os parâmetros do dia num objeto da *struct* parameters e aplica os valores na fórmula, como pode ser visto na Figura 57:

```
float formula(parameters p ){
    float resultado;

    resultado = p.coeficienteChuva*((p.corGram + ((p.vento*p.insolacao)/p.umidade)) - p.resultadoAnterior);
    if (resultado < 0) {
        resultado = 0;
    }
    return resultado;
}
```

Figura 57 - Função formula.

Essa fórmula foi elaborada considerando que a velocidade do vento e a insolação, segundo (Kobiyama; Chaffe), aumentam a taxa de evaporação da água no solo, portanto, seus valores são

diretamente proporcionais a necessidade de água. Já a umidade do solo, quanto maior for, menos água é necessário, por isso sua contribuição é inversamente proporcional à necessidade de água. A cor da grama, calculada na função `get_cor`, representa a saúde do gramado e, portanto, seu valor tem grande influência no resultado da fórmula. Por fim, o resultado anterior é utilizado para evitar que caso um bloco seja regado com muita água em um dia, no próximo seja utilizado mais água do que o necessário. Caso o resultado anterior seja maior que o resultado da primeira equação, o resultado final é considerado como 0.

3.3.1.2.8 Tomada de Decisão sobre Bloco

A tomada de decisão sobre um bloco ser irrigado ou não é feita através de uma máquina de estados simples, que possui quatro estados: INICIAL, estado onde é feito o cálculo do índice da necessidade de água; NAOREGA, estado de decisão que diz para o bloco não ser regado; CALC, estado onde é tomada a decisão. Caso o resultado do índice seja maior do que 10, o bloco será irrigado, caso contrário, não; REGA, estado de decisão que diz para o bloco ser regado e define a quantidade de água a ser utilizada; FIM, estado final.

A máquina de estados pode ser vista na Figura 58:

```

int state_machine(parameters param_dia){
    enum estado estado_atual = INICIAL;
    float decisao;
    int qtd_agua;

    while(estado_atual != FIM) {

        switch(estado_atual) {
        case INICIAL:
            decisao = formula(param_dia);
            estado_atual = CALC;
        case CALC:
            if(decisao > 10) {
                estado_atual = REGA;
            }
            else {
                estado_atual = NAOREGA;
            }
            break;

        case REGA:
            qtd_agua = decisao;
            estado_atual = FIM;
            break;

        case NAOREGA:
            qtd_agua = 0;
            estado_atual = FIM;
            break;

        case FIM:
            break;
        }
    }
    return qtd_agua;
}

```

Figura 58 - Máquina de Estados responsável por tomar a decisão de regar o bloco.

3.3.1.2.9 Leitura e Armazenamento do Resultado Anterior em Arquivos

As últimas funções referentes ao módulo de tomada de decisão são as funções responsáveis por ler e salvar os resultados em um arquivo. O arquivo, chamado de resAnterior possui em cada linha as coordenadas X e Y do bloco e a quantidade de água utilizada para regá-lo. O processo de leitura e armazenamento dos dados é feito linha a linha, como pode-se ver na Figura 59:

```
void save_resAnterior(vector<block_result> resultados, ofstream &output){  
    vector<block_result>::iterator itr;  
  
    output.open("resAnterior", ios::app);  
    for (itr = resultados.begin(); itr != resultados.end(); itr++) {  
        output << (*itr).x << " " << (*itr).y << " " << (*itr).qtd_agua << endl;  
    }  
    output << " ";  
    output.close();  
}  
  
int get_resAnterior(int x, int y, ifstream &input){  
    int res_x, res_y, qtd_agua;  
    input.open("resAnterior");  
    string line;  
    while ( getline (input,line) )  
    {  
        istringstream resultado(line);  
        resultado >> res_x >> res_y >> qtd_agua;  
        if(res_x == x && res_y == y) {  
            input.close();  
            return qtd_agua;  
        }  
    }  
    input.close();  
    return 0;  
}
```

Figura 59 - Funções responsáveis por ler e armazenar os resultados obtidos.

3.3.1.3 Aspersores

3.3.1.3.1 Leitura de Posição dos Aspersores

A posição dos aspersores no jardim é definida manualmente, portanto, para o programa saber onde estão localizados os aspersores, é necessário inserir manualmente em um arquivo suas posições, na ordem coordenada X coordenada Y. Cada linha representa um aspersor.

Como pode-se ver na Figura 60, a função responsável pela leitura da posição dos aspersores retorna um vetor com todos os aspersores e suas coordenadas além da orientação:

```
vector<sprinkler> read_sprinklers()
```

Figura 60 - Cabeçalho da função `read_sprinklers`.

A *struct* `sprinkler` (aspersor em inglês), possui três campos: coordenada x, coordenada y e orientação, como pode ser visto na Figura 61:

```
struct sprinkler {
    float x;
    float y;
    enum orientacao orientacao;
};
```

Figura 61 - Struct que representa um `sprinkler`.

Existem quatro orientações possíveis: CIMA, BAIXO, ESQUERDA e DIREITA. A orientação do aspersor é definida na hora da leitura do arquivo, de acordo com sua posição no jardim. A Figura 62 abaixo ilustra a decisão para a Figura 17:

```

while (getline (input,line))
{
    sprinklers.push_back(sprinkler());
    istringstream resultado(line);
    resultado >> sprinklers[i].x >> sprinklers[i].y;
    if(sprinklers[i].x < 5 ) {
        sprinklers[i].orientacao = ESQUERDA;
    }
    else if((sprinklers[i].x >= 5 && sprinklers[i].x < 20) && sprinklers[i].y < 10) {
        sprinklers[i].orientacao = CIMA;
    }
    else if(sprinklers[i].x > 15) {
        sprinklers[i].orientacao = DIREITA;
    }
    else if((sprinklers[i].x >= 5 && sprinklers[i].x < 20) && sprinklers[i].y >= 10) {
        sprinklers[i].orientacao = BAIXO;
    }
    i++;
}

```

Figura 62 - Determinação de direção do sprinkler.

3.3.1.3.2 Modelagem do Jato D'água

Para molhar cada bloco do jardim, o microcontrolador responsável por controlar o aspersor precisa saber o ângulo entre os dois e a quantidade de água a ser utilizada. A função responsável por agrupar as informações é a `get_jato`, como na Figura 63:

```
jato get_jato(vector<sprinkler> sprinklers, block_result grass_block);
```

Figura 63 - Cabeçalho da função `get_jato`.

As coordenadas do bloco são passadas por parâmetro através de um objeto da *struct block_result*, que pode ser vista na Figura 64:

```
struct block_result {  
    int x;  
    int y;  
    int qtd_agua;  
};
```

Figura 64 - Struct que representa o bloco a ser regado e a quantidade de água.

E o resultado, um objeto da *struct* jato, contendo todas as informações, tem sua estrutura representada na Figura 65:

```
struct jato {  
    float distancia;  
    int angle;  
    int qtd_agua;  
    enum orientacao orientacao;  
};
```

Figura 65 - Struct que representa um jato.

A detecção de qual aspersor deverá ser utilizado é feita através das coordenadas do bloco, como pode ser visto na Figura 66:

```

if(grass_block.x < 10 ) {
    jato.distancia = distancia(sprinklers[2].x, grass_block.x, sprinklers[2].y, grass_block.y);
    jato.angle = angulo(sprinklers[2].x, grass_block.x, sprinklers[2].y, grass_block.y);
    jato.orientacao = sprinklers[2].orientacao;
}
else if((grass_block.x >= 5 && grass_block.x < 20) && grass_block.y < 10) {
    jato.distancia = distancia(sprinklers[0].x, grass_block.x, sprinklers[0].y, grass_block.y);
    jato.angle = angulo(sprinklers[0].x, grass_block.x, sprinklers[0].y, grass_block.y);
    jato.orientacao = sprinklers[0].orientacao;
}
else if(grass_block.x > 15) {
    jato.distancia = distancia(sprinklers[3].x, grass_block.x, sprinklers[2].y, grass_block.y);
    jato.angle = angulo(sprinklers[3].x, (grass_block.x)/100, sprinklers[3].y, grass_block.y);
    jato.orientacao = sprinklers[3].orientacao;
}
else if((grass_block.x >= 5 && grass_block.x <= 20) && grass_block.y >= 10 ) {
    jato.distancia = distancia(sprinklers[1].x, grass_block.x, sprinklers[1].y, grass_block.y);
    jato.angle = angulo(sprinklers[1].x, grass_block.x, sprinklers[1].y, grass_block.y);
    jato.orientacao = sprinklers[1].orientacao;
}

```

Figura 66 - Processo de formação do jato.

A lógica para selecionar o aspersor é a mesma utilizada para determinar a orientação do aspersor lido do arquivo, como pode ser visto na Figura 62.

3.3.1.3.3 Cálculo da Distância entre Aspersor e Bloco

A distância entre o aspersor e o bloco é calculada através de uma equação de distância entre dois pontos em uma reta. A função, que recebe as coordenadas como parâmetro, como pode ser visto na Figura 67:

```
float distancia (float xa, float xb, float ya, float yb)
```

Figura 67 - Cabeçalho da função distância.

A implementação da função é uma transcrição da equação abaixo:

$$distância = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$$

3.3.1.3.4 Cálculo do Ângulo entre Aspersor e Bloco

Para o cálculo do ângulo entre o aspersor e o bloco, a função ângulo recebe as coordenadas assim como a função para calcular a distância entre os dois, como está ilustrado na Figura 68:

```
int angulo (float xa, float xb, float ya, float yb)
```

Figura 68 - Cabeçalho da função angulo.

A implementação da função é baseada na equação abaixo:

$$\theta = \cos^{-1} \cos \theta$$

Onde:

$$\cos \theta = \frac{x_a * x_b + y_a * y_b}{\sqrt{x_a^2 + y_a^2} * \sqrt{x_b^2 + y_b^2}}$$

Para obter o valor em graus ao invés de radianos é feita uma conversão antes de retornar o resultado, como pode ser visto na Figura 69:

```
angulo = acos (cos0) * 180.0 / PI;
```

Figura 69 - Fórmula para obter o ângulo

3.3.1.3.4 Armazenamento dos Ângulos

Para enviar os ângulos dos blocos a serem regados, é utilizado um arquivo onde os ângulos são salvos pelo programa principal para serem lidos pelo script send.py. Devido a semelhança com a função save_resAnterior presente na Figura 59, optou-se por não colocar uma imagem ilustrando a função save_Angulos.

3.3.1.4 Núcleo

A função principal do programa é a *main* presente no arquivo core.cpp. Ela é responsável por chamar todos os módulos e interliga-los. A primeira etapa é coletar os dados dos sensores, através de uma chamada ao sistema, executa-se o script read.py e então os valores de insolação e umidade são salvos no arquivo sensordata.txt. Após isso, são coletados os dados do dia, como na Figura 70:

```
coeficiente = parse_tempo();
vento = get_vento();
umidade = get_umidade();
insolacao = get_insolacao();
```

Figura 70 - Chamadas para obter os valores utilizados na tomada de decisão.

Essas operações são feitas fora do laço principal pois elas só precisam ser feitas uma vez ao dia, já que o valor é o mesmo para o jardim todo.

Após a coleção de dados, é chamado o fluxo do processamento de imagens, responsável por formar a panorâmica do jardim e retornar um vetor contendo as informações de todos os blocos do jardim. Essa chamada é feita através das funções stitch e image_processing, como pode ser visto na Figura 71:

```
field = stitch(argc, argv);
grass_blocks = image_processing(field, max_col, max_linha);
final_field = imread("final.jpg");
```

Figura 71 - Chamadas das funções referentes ao Processamento de Imagens.

Por fim são lidos os aspersores do arquivo sprinklerpos.txt antes do laço principal.

Antes do laço principal, é feita uma verificação, caso o coeficiente de chuva seja menor que 10%, significa que irá chover e, portanto, não é necessário regar o jardim. É feito então uma iteração sobre os blocos presentes em grass_blocks para obter suas coordenadas e então é definido o vetor de resultados, como na Figura 72:

```

    else {
        i = 0;
        for(it = grass_blocks.begin(); it != grass_blocks.end(); it++){
            resultados.push_back(block_result());
            resultados[i].x = (*it).x/100;
            resultados[i].y = (*it).y/100;
            resultados[i].qtd_agua = 0;
            i++;
        }
    }

```

Figura 72 - Determinação o resultado quando há previsão de muita chuva no dia.

Caso o coeficiente de chuva seja maior que 10%, o laço principal itera sobre o vetor grass_blocks, criando um objeto do tipo parameters temporário, que é usado para a tomada de decisão. A tomada de decisão retorna a quantidade de água a ser utilizada e então essa quantidade de água é salva no vetor de resultados que será utilizado para definir a direção do jato d'água. A Figura 73 mostra o fluxo do laço principal:

```

for(it = grass_blocks.begin(); it != grass_blocks.end(); it++) {
    parameters p;
    p.coeficienteChuva = coeficiente;
    p.umidade = umidade;
    p.vento = vento;
    p.insolacao = insolacao;
    p.resultadoAnterior = get_resAnterior((*it).x/100, (*it).y/100, input);
    p.corGramado = get_cor((*it).regado, (*it).dif_cor);
    qtd_agua = state_machine(p);

    resultados.push_back(block_result());
    resultados[i].x = (*it).x/100;
    resultados[i].y = (*it).y/100;
    resultados[i].qtd_agua = qtd_agua;
    jatos.push_back(get_jato(sprinklers, resultados[i]));
    i++;
}

```

Figura 73 - Laço principal do programa principal.

Ao final da execução do laço, remove-se o arquivo de resultados anteriores para salvar o novo, os ângulos calculados pela função get_jato são salvos no arquivo ângulos e é chamado o script para enviar os dados para o Arduino, o send.py, como pode ser visto na Figura 74:

```

remove("resAnterior");
save_Angulos(jatos);
system("python send.py");
save_resAnterior(resultados, output);

```

Figura 74 - Processo para salvar os resultados anteriores.

E então termina a execução do programa.

3.3.1.4.1 Execução Diária do Programa

Para a execução diária do programa, optou-se por utilizar shell scripts que podem ser programados para execução automática através do programa Crontab. Para ativar a execução automática executa-se o script *start.sh* que irá chamar o script responsável por executar o programa *start.cpp*. Esse programa verifica a temperatura do dia e programa a execução do programa principal de acordo com o resultado, como pode ser visto na Figura 75. É necessário ajustar o caminho absoluto onde está armazenado o programa no computador para que o script funcione.

```

system("crontab -r");
if (temp > 10) {
    system("(crontab -l ; echo \"00 00 * * * /home/pedrosmv/Documents/TCC/Project/Core/runhomecontrol.sh $1 >/dev/null 2>&1\") | crontab -");
}
else
    system("(crontab -l ; echo \"00 06 * * * /home/pedrosmv/Documents/TCC/Project/Core/runhomecontrol.sh $1 >/dev/null 2>&1\") | crontab -");

```

Figura 75 - Programação de execução do programa principal.

3.3.2 Hardware

3.3.2.1 Arduino Uno R3

O Arduino UNO é uma placa com um microcontrolador baseado no microcontrolador ATmega328P que pode ser programada através de uma linguagem baseada em C ou até mesmo em C. Ele possui 14 pinos digitais de entrada e saída, 6 pinos de entrada analógicos e um cristal de quartzo 16 MHz. O modelo utilizado no projeto é similar ao da Figura 76. Suas especificações encontram-se na Tabela 1:



Figura 76 - Ilustração da placa Arduino UNO.

Fonte: <https://store.arduino.cc/usa/arduino-uno-rev3>

A grande vantagem do Arduino é seu baixo custo, o que faz com que o preço total do projeto continue dentro da abordagem adotada inicialmente de fazer um projeto acessível e de fácil implementação. Além do custo, as placas Arduino são *open-source*, possuindo uma extensa documentação disponível gratuitamente na internet, o que facilita seu uso.

Tabela 1 - Especificações do Arduino Uno.

Microcontrolador	ATmega328P
Voltagem de Operação	5 V
Voltagem de Entrada (Recomendado)	7-12 V
Limite de Voltagem de Entrada	6-20 V
Pinos de Entrada/Saída Digitais	14 (Desses, 6 podem prover saída PWM)
Pinos de Entrada Digitais	6
Corrente DC por Pino de Entrada/Saída	20 mA
Corrente DC no Pino de 3,3 V	50 mA
Memória Flash	32 KB
Memória SRAM	2 KB
Memória EEPROM	1 KB
Clock	16 MHz
Pino onde o LED <i>on-board</i> está conectado	13
Comprimento	68.6 mm
Largura	53.4 mm
Peso	25g

3.3.2.2 Sensor de Umidade do Solo YL-69

O sensor de umidade utilizado nesse projeto é composto por duas partes, como pode ser visto na Figura 77, uma placa eletrônica (YL-38) e um sensor com duas sondas (YL-69) que são responsáveis por detectar o conteúdo da água.



Figura 77 - Ilustração do sensor YL-69.

Fonte: <https://randomnerdtutorials.com/guide-for-soil-moisture-sensor-yl-69-or-hl-69-with-the-arduino/>

As sondas são responsáveis por passar corrente pelo solo e ler a resistência entre elas para obter o nível de umidade. A presença de mais água faz com que o solo conduza eletricidade mais fácil, o que indica uma baixa resistência, enquanto o solo seco dificulta a condução de eletricidade, indicando uma resistência alta. Vale ressaltar que esses valores são aproximações e que uma simples chacoalhada no solo pode mudar drasticamente o valor lido, ainda assim, pelo baixo custo do sensor, foi escolhido utilizá-lo ainda assim. Suas especificações encontram-se na Tabela 2.

Tabela 2 - Especificações do sensor YL 69

Voltagem de Operação	3,3 V – 5 V
Voltagem de Saída	0 V – 4,2 V
Corrente	35 mA
Dimensões YL-69	60mm x 20mm
Dimensões YL-38	30mm x 16mm

3.3.2.3 Raspberry Pi

A Raspberry Pi é um computador de placa única que possui microprocessadores, memória e entrada/saída embutidos em uma placa só. Uma Raspberry é capaz de executar as principais funções de um computador como acessar a internet, compilar códigos via terminal e editar arquivos. O modelo utilizado no projeto foi a Raspberry Pi Model B, que pode ser vista na Figura 78. Suas especificações encontram-se na Tabela 3.

O grande atrativo da Raspberry Pi, que levou a escolhê-la ao invés de um computador convencional é o seu baixo custo. O seu modelo mais atual, Raspberry Pi 3 Model B, pode ser encontrado na internet por U\$ 35, 00. Com o custo adicional de um cartão de memória, a Raspberry se torna apta a rodar o programa principal do projeto sem maiores problemas.

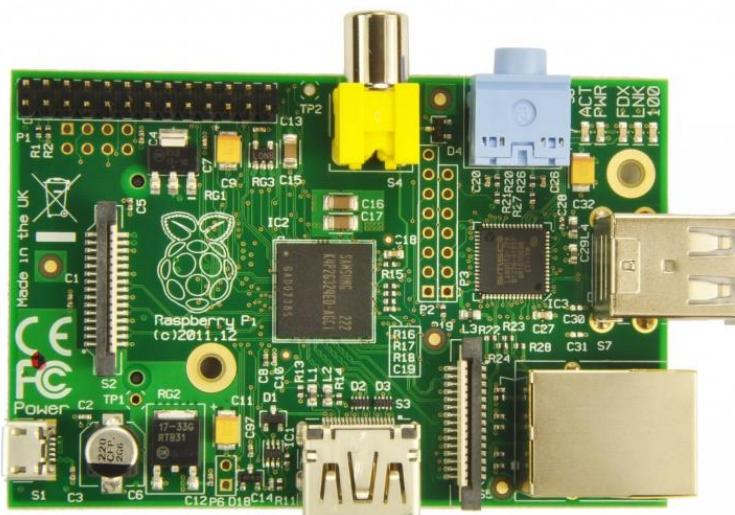


Figura 78 - Raspberry Pi Model B.

. Fonte: <https://elinux.org/File:RaspiFront.JPG>

Tabela 3 - Especificações da Raspberry Pi Model B.

Sistema Presente no Chip	Broadcom BCM2835 (CPU + GPU. SDRAM é um chip separado)
CPU	700 MHz ARM11 ARM1176JZF-S core
Memoria	512 MiB
Fonte de Energia	5 V (DC) via Micro USB tipo B
Voltagem de Operação	5V
Dimensões	85.0 x 56.0 mm x 17mm
Peso	40g

3.3.2.3 Mini Painel Solar Fotovoltaico

Para medir a insolação foi utilizado um mini painel solar fotovoltaico como visto na Figura 79. Esse painel gera uma tensão através do Efeito Fotovoltaico e é formado por um conjunto de células fotovoltaicas associadas em paralelo. Suas especificações não estão disponíveis por isso não serão apresentadas nesta seção.



Figura 79 - Painel Solar Fotovoltaico utilizado no Projeto.

3.4. Resultados Obtidos

Todos os resultados foram obtidos após compilar o programa principal utilizando a versão 5.4.0 do compilador GCC no sistema operacional Linux Mint. Foi feita também uma simulação na Raspberry Pi, que operou no sistema operacional Raspbian.

3.4.1 Simulação com Valores dos Sensores (Raspberry Pi)

Os resultados obtidos neste trabalho são referentes a um ambiente de simulação que capta valores dos sensores e da internet para gerar uma lista contendo as informações referentes aos blocos que representam um gramado. Essas informações especificam a quantidade de água necessária para cada bloco, o ângulo entre o bloco e as coordenadas do bloco.

Além das informações para a irrigação, o *software* desenvolvido entrega uma imagem que demonstra visualmente a necessidade de água de cada bloco e quais blocos serão irrigados. A Figura 80 ilustra um resultado utilizando a previsão do tempo da cidade de Fortaleza. Os círculos azuis são pintados de acordo com a quantidade de água necessária, quanto mais próximo do preto, menos água é necessária. Isso pode ser verificado ao analisar a imagem, as áreas mais verdes apresentam um círculo mais escuro.

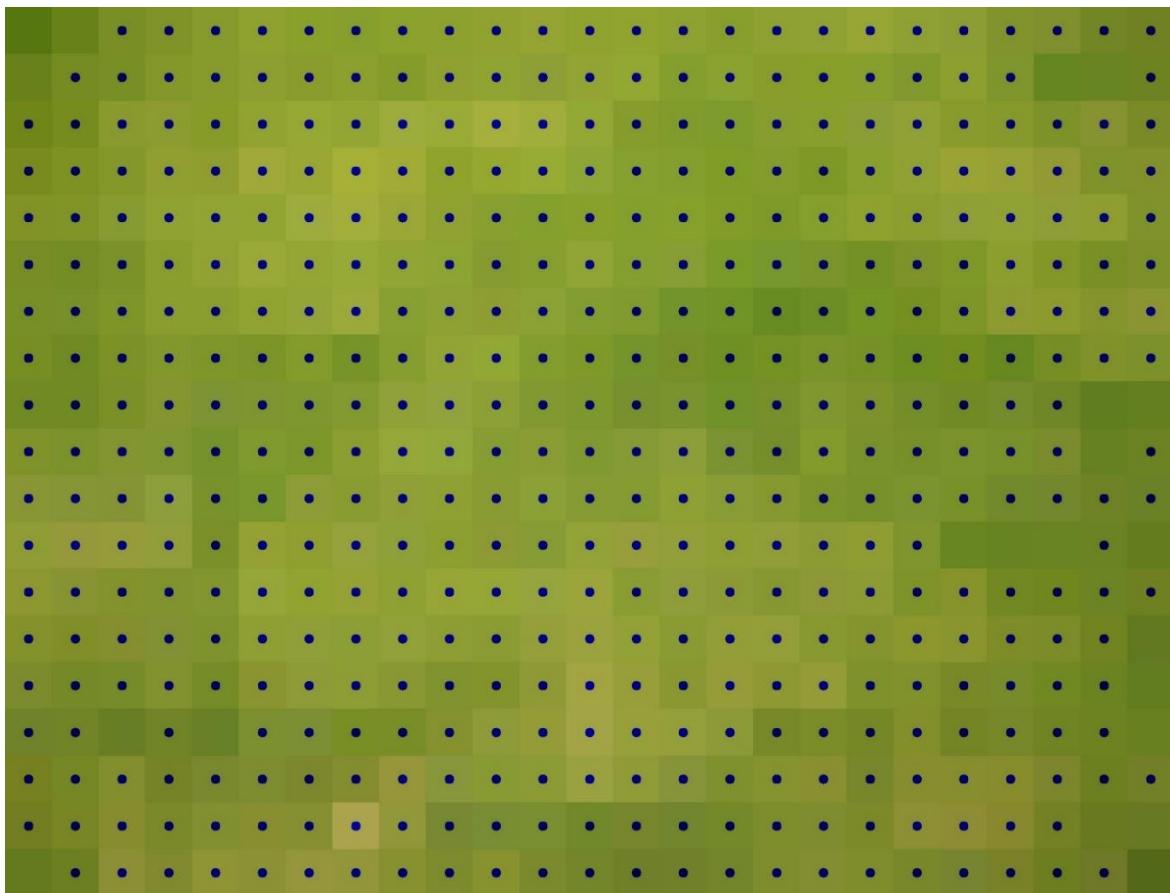


Figura 80 - Imagem Resultado do Programa Principal.

Outra parte importante abrangida nas simulações feitas para o desenvolvimento deste trabalho foi a captação dos valores via sensores. A comunicação serial entre Raspberry e Arduino funcionou como o desejado e se mostrou mais simples do que o esperado. Ao invés de utilizar as portas seriais de cada um dos componentes, optou-se por conectar o Arduino diretamente na Raspberry Pi via entrada USB e fazer a leitura por essa porta.

3.4.2 Simulação com Valores Alterados e Imagem Original (Notebook)

Visando obter resultados variados para validar o projeto, foram feitas quatro simulações com os mesmo valores variando apenas a imagem utilizada. Os valores utilizados foram:

Tabela 4 - Valores Utilizados em Simulação do Sistema.

Coeficiente	0.794872
Velocidade do Vento	7.2 m/s (que resulta num valor de 2 de 5 na tomada de decisão)
Umidade	2 (de 4)
Insolação	4.8 (de 5)

Para ilustrar o resultado, decidiu-se separar apenas uma porção da imagem para demonstrar o efeito dos valores selecionados na Tabela 4. A porção selecionada é a mesma em todos os casos desenvolvidos. Para a imagem original e sem resultados anteriores, o resultado encontra-se na Figura 81

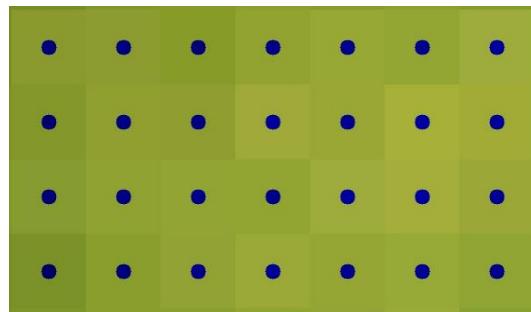


Figura 81 - Porção selecionada para demonstrar a simulação sem resultados anteriores.

A Figura 81 contém os blocos com coordenadas em X: entre 2 e 8 e Y: entre 2 e 5. Os valores para cada bloco e a decisão tomada pelo software encontram-se na Tabela 5:

Tabela 5 - Quantidade de Água e decisão tomada para os blocos na Figura 81 .

X	Y	Quantidade de Água	Regado
2	2	50	Sim
3	2	51	Sim
4	2	49	Sim
5	2	55	Sim
6	2	58	Sim
7	2	57	Sim
8	2	63	Sim
2	3	46	Sim
3	3	53	Sim
4	3	53	Sim
5	3	62	Sim
6	3	59	Sim
7	3	66	Sim
8	3	62	Sim
2	4	49	Sim
3	4	54	Sim
4	4	56	Sim
5	4	56	Sim
6	4	63	Sim
7	4	65	Sim
8	4	60	Sim
2	5	42	Sim
3	5	50	Sim
4	5	56	Sim
5	5	60	Sim
6	5	58	Sim
7	5	60	Sim
8	5	56	Sim

Após a execução do programa para essa simulação sem resultados anteriores, ele é executado novamente agora contabilizando os resultados da última execução, simulando assim a presença dos resultados de um dia anterior. O resultado pode ser visto na **Erro! Fonte de referência não encontrada.**. Já os resultados podem ser vistos na Tabela 6. Apesar de a Tabela 6 sinalizar que a maioria dos blocos necessitam ser regados, podemos ver na Figura 82 que a quantidade é muito pequena.

Tabela 6 - Quantidade de Água e decisão tomada para os blocos na Figura 82.

X	Y	Quantidade de Água (0-100)	Regado
2	2	10	Não
3	2	11	Sim
4	2	10	Não
5	2	11	Sim
6	2	12	Sim
7	2	11	Sim
8	2	13	Sim
2	3	0	Não
3	3	11	Sim
4	3	11	Sim
5	3	13	Sim
6	3	12	Sim
7	3	14	Sim
8	3	13	Sim
2	4	10	Não
3	4	11	Sim
4	4	11	Sim
5	4	11	Sim
6	4	13	Sim
7	4	13	Sim
8	4	12	Sim
2	5	0	Não
3	5	11	Sim
4	5	11	Sim
5	5	12	Sim
6	5	11	Sim
7	5	12	Sim
8	5	11	Sim

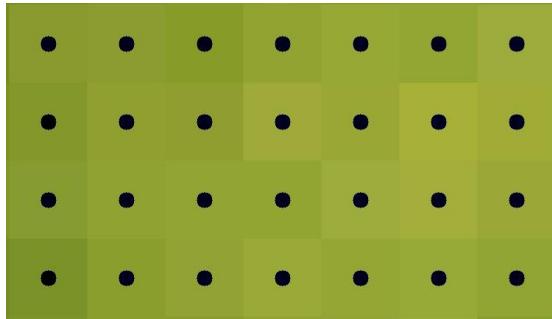


Figura 82 - Porção selecionada para demonstrar a simulação com resultados anteriores.

3.4.2 Simulação com Valores Alterados e Imagem Modificada

Para validar o projeto em caso de um gramado pouco saudável, modificou-se a imagem original para que a seção selecionada para ilustrar os resultados das simulações apresentasse uma coloração amarela. O resultado pode ser visto na Figura 83. Os valores utilizados são os mesmos presentes na Tabela 4.

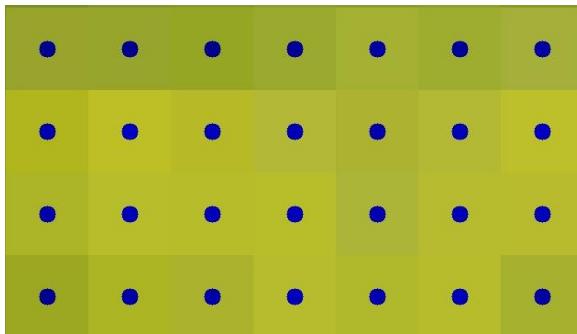


Figura 83 - Porção selecionada modificada para demonstrar a simulação sem resultados anteriores.

Observando os resultados presentes na Tabela 7 pode-se ver que a quantidade de água necessária calculada é maior que os resultados vistos na Tabela 5.

Tabela 7 - Quantidade de Água e decisão tomada para os blocos na Figura 83.

X	Y	Quantidade de Água (0-100)	Regado
2	2	57	Sim
3	2	57	Sim
4	2	56	Sim
5	2	59	Sim
6	2	65	Sim
7	2	62	Sim

8	2	65	Sim
2	3	69	Sim
3	3	77	Sim
4	3	73	Sim
5	3	72	Sim
6	3	69	Sim
7	3	73	Sim
8	3	77	Sim
2	4	68	Sim
3	4	74	Sim
4	4	73	Sim
5	4	74	Sim
6	4	69	Sim
7	4	73	Sim
8	4	73	Sim
2	5	58	Sim
3	5	67	Sim
4	5	67	Sim
5	5	74	Sim
6	5	70	Sim
7	5	73	Sim
8	5	65	Sim

Assim como foi feito na simulação com a imagem original, para a imagem modificada foi executado o programa novamente visando simular a presença do resultado do dia anterior. Os resultados encontram-se na Figura 84 e Tabela 8. Com os resultados apresentados, pode-se ver que mesmo possuindo uma necessidade de água maior devido ao gramado amarelado, a presença de água do dia anterior ainda é um fator determinante para a decisão do dia, eclipsando a cor do gramado, que pelo que pode ser visto na Tabela 7, exerce uma grande influência no resultado final.

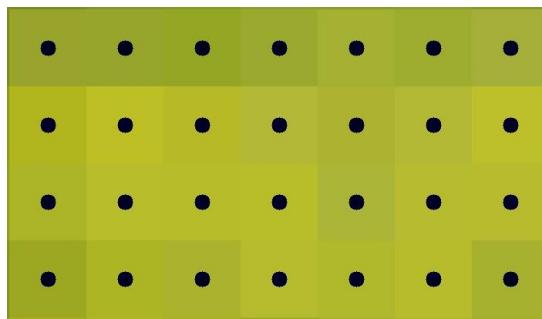


Figura 84 - Porção selecionada modificada para demonstrar a simulação com resultados anteriores.

Tabela 8 - Quantidade de Água e decisão tomada para os blocos na Figura 84.

X	Y	Quantidade de Água (0-100)	Resultado
2	2	11	Sim
3	2	11	Sim
4	2	11	Sim
5	2	12	Sim
6	2	14	Sim
7	2	12	Sim
8	2	14	Sim
2	3	15	Sim
3	3	15	Sim
4	3	15	Sim
5	3	15	Sim
6	3	14	Sim
7	3	15	Sim
8	3	15	Sim
2	4	14	Sim
3	4	15	Sim
4	4	15	Sim
5	4	15	Sim
6	4	14	Sim
7	4	15	Sim
8	4	15	Sim
2	5	12	Sim
3	5	14	Sim
4	5	14	Sim
5	5	15	Sim
6	5	15	Sim
7	5	15	Sim
8	5	13	Sim

3.5. Dificuldades e Limitações

As maiores dificuldades assim como limitações encontradas no desenvolvimento deste projeto estão relacionadas às características da linguagem C++. Por uma limitação da Raspberry em relação à biblioteca OpenCV, o projeto teve que ser desenvolvido totalmente em C++ com exceção dos scripts referentes à comunicação serial entre a Raspberry e o Arduino.

Ao compararmos C++ a Python, por exemplo, apesar de oferecer muitos recursos, a linguagem C++ exige um esforço maior para realizar tarefas que são simples em Python. A ideia inicial do projeto era utilizar uma Árvore de Decisão para o processo de tomada de decisão, porém, implementar uma árvore de decisão simples em C++ é trabalhoso e nada trivial, ao passo que em Python seria uma tarefa mais simples. Outras vantagens da linguagem Python são a existência e facilidade de uso de bibliotecas externas, algo que facilita o desenvolvimento do projeto e permite um foco maior à implementação de módulos mais importantes.

3.6. Considerações Finais

Levando em consideração o objetivo inicial do projeto, que era criar um sistema de irrigação que contribua para um consumo menor de água e seja de fácil implementação, pode-se dizer que o objetivo foi alcançado. Isso pode ser confirmado ao analisarmos as Tabelas Tabela 5, Tabela 6, Tabela 7 e Tabela 8. Nelas, através da variável Quantidade de Água podemos ver que todos os fatores, tanto os presentes na Tabela 4 quanto a saúde da grama, contribuem para um consumo equilibrado, principalmente se o gramado foi regado no dia anterior.

Pelas simulações realizadas, foi possível concluir que mesmo em condições que exigem muita água para irrigar o gramado, a saúde do mesmo exerce uma influência superior, como era o esperado. Além da saúde do gramado, o programa respondeu à presença de água do dia anterior como desejado ao apresentar valores muito baixos, o que evita o desperdício e também mantém a umidade do solo equilibrada.

CAPÍTULO 4: CONCLUSÃO

4.1. Contribuições

O resultado deste trabalho traz como contribuição um protótipo de fácil implementação e baixo custo que diminui gastos desnecessários com água ao deixar a cargo de um *software* decidir

o quanto de água será utilizado. Esse projeto também serve como inspiração para outros projetos na área de Automação Residencial que utilizem o *backbone* Raspberry Pi + Arduino, visto que apresenta uma maneira simples de comunicação entre os dois.

4.2. Relacionamento entre o Curso e o Projeto

O curso de Engenharia de Computação, graças a sua natureza, permite contato com as duas principais áreas cobertas por esse projeto: a Ciência da Computação e Engenharia Elétrica. Mesmo que não seja muito aprofundado em algumas disciplinas, o conteúdo visto no curso permite uma versatilidade que poucos cursos possuem.

Sem todo o conhecimento adquirido durante os anos de curso sobre programação, não seria possível completar esse projeto. Matérias como Computação Gráfica e Algoritmos e Estrutura de Dados forneceram um bom ponto inicial para o desenvolvimento do software principal do projeto. Além da parte computacional, o conteúdo adquirido em disciplinas como Laboratório de Circuitos Eletrônicos e Laboratório de Sistema Digitais ajudaram na hora de trabalhar com o hardware necessário.

4.3. Considerações sobre o Curso de Graduação

Como citado anteriormente, o curso de Engenharia de Computação oferecido pelo campus de São Carlos da Universidade de São Paulo possui uma grade versátil que permite contato com a Ciência da Computação e a Engenharia Elétrica. Apesar das vantagens que essa versatilidade traz, muitas vezes permeou durante a minha graduação o sentimento de que essa característica poderia ter sido melhor aproveitada.

A sensação na maioria das aulas é que eram dois cursos diferentes sendo ministrados sob a alcunha de um só. Poucas vezes se teve um contato entre as duas frentes para interligar os conhecimentos de cada área.

Outro grande problema foi a intensa carga horária, a Engenharia de Computação é o terceiro curso com mais horas necessárias para se formar no campus de São Carlos e o segundo curso de

engenharia com mais horas de trabalho. Esses números por si sós ilustram a dificuldade do curso, mas, soma-se a isso, vários professores que passaram trabalhos que extrapolaram o número de créditos estabelecidos pela ementa ou até mesmo caso de matérias sem créditos de trabalho que tiveram trabalhos extensos e difíceis. Esse cenário impossibilita uma dedicação maior às matérias, forçando o aluno que pretende cumprir todos os créditos do semestre a percorrer superficialmente o conteúdo dado.

Por fim, minha maior crítica ao curso é a falta de prática nas disciplinas. Foram poucas as que tiveram laboratórios para aprendermos o lado prático do conteúdo. Embora a teoria seja de suma importância, não se pode esperar que o conhecimento se dê somente por ela. Sinto que algumas matérias poderiam ter dado lugar a mais laboratórios, possibilitando um aprendizado mais completo. E, como foi dito acima, com a carga horária atual do curso, engajar-se em uma atividade extracurricular ou numa iniciação científica se tornou impraticável.

4.4. Trabalhos Futuros

Para futuros trabalhos, seria interessante implementar o circuito completo com vários sprinklers sendo controlados por outros Arduinos e uma válvula que controle a pressão da água, assim como um processo de tomada de decisão que utilize uma Árvore de Decisão para classificar os blocos e um sistema de captura de imagens automático, utilizando webcams.

Referências

ALBUQUERQUE, Márcio Portes; ALBUQUERQUE, Marcelo Portes. **Processamento de imagens:** métodos e análises. Rio de Janeiro: FACET, 2001.

ALMEIDA, Eliane et al. **Energia Solar Fotovoltaica: Revisão Bibliográfica.** 2016

BOLZANI, C. A. M. **Residências Inteligentes.** [S.l.]: Livraria da Física, 2004.

- BROWN, Matthew; LOWE, David G. **Automatic Panoramic Image Stitching using Invariant Features**. 2007.
- CASADOMO. **Domótica - Introducción**. Agosto 2010. Disponível em: <<http://www.casadomo.com/>>. Acesso em 10 out. 2017
- ENVI. Guia do ENVI em Português**. Sulsoft, 2000. Diponível em <www.sulsoft.com.br/>. Acesso em 15 out. 2006.
- KOBIYAMA, Masato; CHAFFE, Pedro Luiz Borges. **EVAPOTRANSPIRAÇÃO**. [20-]
- KOPETZ, Hermann. **Real-Time Systems, Design Principles for Distributed Embedded Applications**. 2011.
- KUEHNI, Rolf G. **Color Space and Its Divisions: Color Order from Antiquity to the Present**. 2003.
- LOESDAU, Martin; CHABRIER, Sébastien; GABILLON, Alban. **Hue and Saturation in the RGB Color Space**. 2014.
- MAALEL, Nourhene et al; **Reliability for emergency applications in Internet of Things**. 2013.
- MATHEMATICAL MORPHOLOGY. **Wikipédia**. Disponível em <https://en.wikipedia.org/wiki/Mathematical_morphology>. Acesso em 25 de out. de 2017
- PLATANIOTIS, Konstantinos N.; N. VENETSANOPoulos, Anastasios. **Color Image Processing and Applications**. 2000.
- RELATÓRIO CRHB. Brasília: Agência Nacional de Águas – ANA, 2012.
- SANTRA, Santanu; ACHARJYA, Pinaki Pratim. **A Study And Analysis on Computer Network Topology For Data**. 2013.
- TAPPARO, Tiago Vilela. **Sistema de irrigação inteligente, Departamento de Sistemas de Computação do Instituto de Ciências Matemáticas e de Computação**. ICMC-USP, 2016.
- WORTMEYER, C.; FREITAS, F.; CARDOSO, L. **Automação residencial: Busca de tecnologias visando o conforto, a economia, a praticidade e a segurança do usuário**. In: **II Simpósio de Excelência em Gestão e Tecnologia**. SEGeT2005. [S.l.: s.n.], 2005.