UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

PLANTARUM: API PARA RECONHECIMENTO DE PLANTAS

ARNO WILSON CASSANIGA

BLUMENAU
2012

2012/2-04

ARNO WILSON CASSANIGA

PLANTARUM: API PARA RECONHECIMENTO DE PLANTAS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Aurélio Faustino Hoppe, Mestre - Orientador

BLUMENAU
2012

2012/2-04

PLANTARUM: API PARA RECONHECIMENTO DE PLANTAS

Por

ARNO WILSON CASSANIGA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Aurélio Faustino Hoppe, Mestre – Orientador, FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre – FURB
Membro:	Prof. Everaldo Artur Grahl, Mestre – FURB

Blumenau, 17 de dezembro de 2012

Dedico este trabalho à minha família e amigos, que sempre ofereceram-me seu apoio e motivação para a conclusão deste trabalho.

AGRADECIMENTOS

À minha família, por toda a motivação e incentivo ao longo de mais uma etapa da minha vida. Agradeço em especial ao meu pai Lúcio Cassaniga, minha mãe Níria Cassaniga e minha avó Olga Koth, que desde jovem ensinaram-me a importância dos estudos e sempre incentivaram-me a dar o meu melhor em todas as escolhas feitas ao decorrer da minha vida.

Aos meus amigos e professores, por fornecerem o conhecimento que permitiu a execução deste trabalho.

Ao meu orientador, Aurélio Faustino Hoppe, por ter acreditado na viabilidade e conclusão deste trabalho, prestando o conhecimento e todo o auxílio que necessitei durante o desenvolvimento deste projeto.

When the solution is simple, God is answering.

Albert Einstein

RESUMO

Este trabalho apresenta um protótipo de uma *Application Programming Interface* (API) para a classificação de plantas baseada no processamento de imagens de folhas. A partir da extração de características geométricas, de cor, nervação, textura e descritores morfológicos, a API desenvolvida é capaz de inferir a espécie de planta que uma determinada folha pertence a partir da comparação deste conjunto de características com uma base de dados de espécies previamente registrada. Os resultados obtidos demonstram que a API desenvolvida é eficiente na classificação de plantas, porém ainda são necessários ajustes para eliminar a necessidade de intervenção humana na localização do pecíolo e da ponta da folha.

Palavras-chave: Processamento de imagens. Classificação de plantas. Descritores morfológicos.

ABSTRACT

This work presents an Application Programming Interface (API) prototype for plant classification based on leaf images processing. By extracting geometric features, color features, venation, texture and morphological descriptors, the designed API is able to infer the plant specimen a given leaf belongs to, by comparing this feature set with a specimen database previously populated. The results show the designed API is efficient regarding plant classification, however adjustments are still required to remove the need of human intervention locating the petiole and leaf tip.

Key-words: Image processing. Plant classification. Morphological descriptors.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

API – *Application Programming Interface*

BS – *Binary Superposition*

C-R – *Centroid Radii*

DFT – *Discrete Fourier Transform*

DLL – *Dynamic Link Library*

FFT – *Fast Fourier Transform*

FM – *Fourier Moments*

IDE – *Integrated Development Environment*

IDFT – *Inverse Discrete Fourier Transform*

MER – Modelo Entidade-Relacionamento

MI – *Moment Invariants*

PCA – *Principal Component Analysis*

PFT – *Polar Fourier Transform*

RBFN – *Radial Basis Function Network*

RF – Requisito Funcional

RGB – *Red Green Blue*

RNA – Rede Neural Artificial

RNF – Requisito Não Funcional

RNP – Rede Neural Probabilística

ROI – *Region of Interest*

SVM – *Support Vector Machines*

SVM-BDT – *Support Vector Machines* em *Binary Decision Trees*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

O Brasil é considerado um dos países com maior diversidade biológica do mundo. Tal biodiversidade torna a árdua tarefa de classificação de espécies (taxonomia) de plantas ainda mais trabalhosa. Esta tarefa é tradicionalmente realizada manualmente, devido ao caráter especializado deste processo. A relação entre grande variedade de espécies vegetais e o contingente de profissionais especializados não se encontra em equilíbrio, resultando em um cenário onde a população de profissionais especialistas não consegue suprir as necessidades de levantamento e classificação das milhares de espécies do riquíssimo reino vegetal. Várias áreas além da botânica, como a medicina e a fisioterapia, dependem fortemente da descoberta de novas espécies, assim como de suas características e propriedades (PLOTZE, 2004, p. 6).

Segundo Wu et al. (2007), existem vários meios para a descrição de plantas, como características visuais, informações celulares e até mesmo descritivos, na forma textual, redigidos pelos próprios botânicos. Enfatiza-se que uma abordagem apropriada do problema deve preferencialmente basear-se na utilização de imagens digitais da planta como informação de entrada. Este tipo de solução torna-se interessante a partir do momento que minimiza a interferência humana na extração das características da planta (nos casos em que as características são informadas textualmente pelo especialista) e na viabilidade, em termos financeiros e tecnológicos, da aquisição das fotos das plantas.

Plotze (2004, p. 2) destaca que a utilização de métodos computacionais para a resolução de problemas na área da botânica acaba implicitamente resultando em novas técnicas e progressos de várias áreas relacionadas à computação, como processamento de imagens e visão computacional.

Dentro do contexto computacional, o desenvolvimento de ferramentas de auxílio ao reconhecimento e classificação de folhas ainda se mostra uma tarefa desafiadora devido à falta de modelos apropriados. Um ponto chave para a realização desta tarefa consiste na seleção de características estáveis e apropriadas para a discriminação de diferentes espécies (SINGH; GUPTA; GUPTA, 2010).

Diante do cenário descrito acima, este trabalho apresenta uma API que realiza a classificação de plantas a partir de imagens digitais de suas folhas, utilizando-se de características configuráveis como fator discriminante entre as diferentes espécies trabalhadas.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma API para o reconhecimento e classificação de plantas a partir de uma imagem digital de suas folhas.

Os objetivos específicos do trabalho são:

a) extrair de uma imagem de entrada a região correspondente à folha da planta;
b) extrair características relevantes da folha, em especial as geométricas, de cor e descritores morfológicos;
c) classificar as plantas dentre as espécies previamente registras em uma base de dados.

1.2 ESTRUTURA DO TRABALHO

O primeiro capítulo apresenta a motivação e os objetivos a serem atingidos com este trabalho.

O segundo capítulo contém a base teórica dos principais conceitos e técnicas exploradas no desenvolvimento do projeto.

No terceiro capítulo estão descritos a arquitetura, a implementação e os resultados obtidos nos testes de validação da API.

O quarto capítulo contém as conclusões formuladas a partir do trabalho e sugestões para extensões ao mesmo.

2 FUNDAMENTAÇÃO TEÓRICA

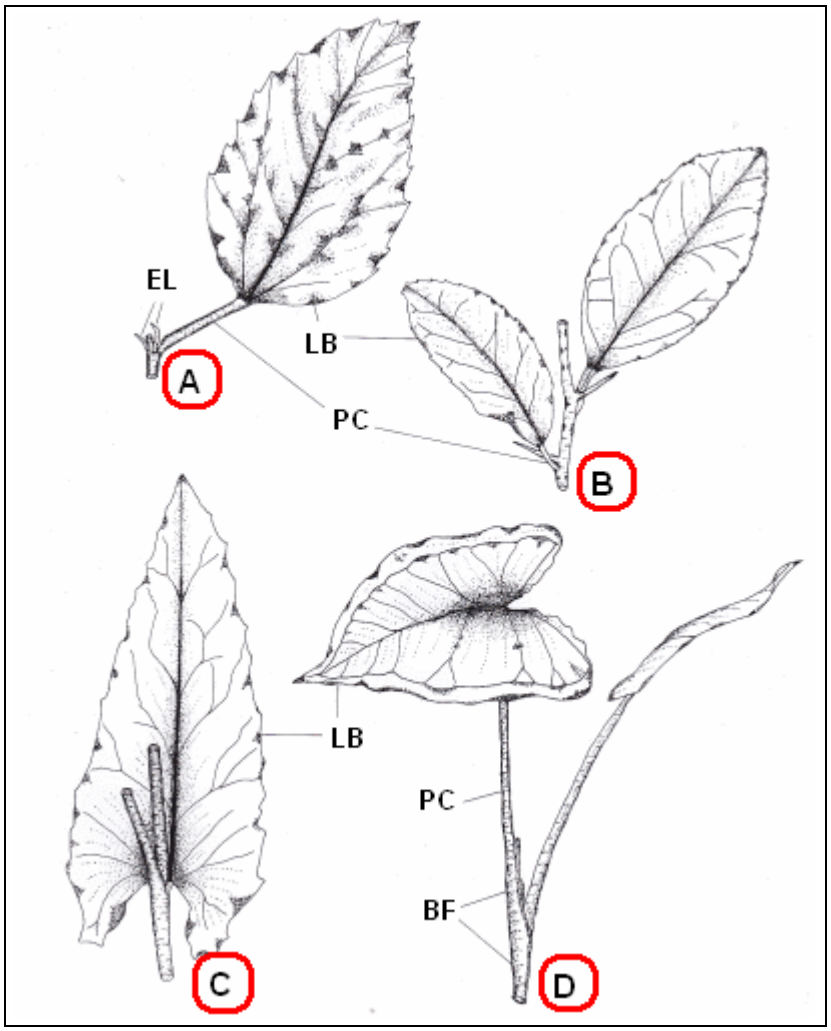
A seção 2.1 apresenta conceitos básicos sobre a morfologia de órgãos foliares, útil para a compreensão dos elementos disponíveis para análise pela API. Na seção 2.2 é exibido um apanhado sobre os conceitos de segmentação de imagens e descritas técnicas para sua execução. A seção 2.3 destina-se a introduzir os descritores de fourier, utilizados neste trabalho para descrever a forma dos órgãos foliares. A seção 2.4 tem por finalidade apresentar a estrutura de uma Rede Neural Probabilística (RNP), classificador utilizado neste projeto para inferir as espécies de planta das folhas submetidas ao processo de classificação. Por fim, na seção 2.5 encontram-se os trabalhos correlatos, relacionados também à classificação de plantas.

2.1 MORFOLOGIA DE ÓRGÃOS FOLIARES

Segundo Souza (2003, p. 125), a folha é um órgão lateral da planta cujas principais funções são a fotossíntese e a transpiração. Este órgão é geralmente laminar e de estrutura dorsiventral, apresentando grande variedade de formas.

Souza (2003, p. 130) destaca que uma folha completa apresenta as seguintes partes: estípulas, bainha, pecíolo e limbo. Porém, é muito comum espécies de plantas possuirem folhas incompletas, não apresentando todas as partes listadas anteriormente.

A Figura 1 exibe exemplos de órgãos foliares, destacando as partes que os compõem.



Fonte: adaptado de Souza (2003, p. 130).

Figura 1 - Exemplo de órgãos foliares

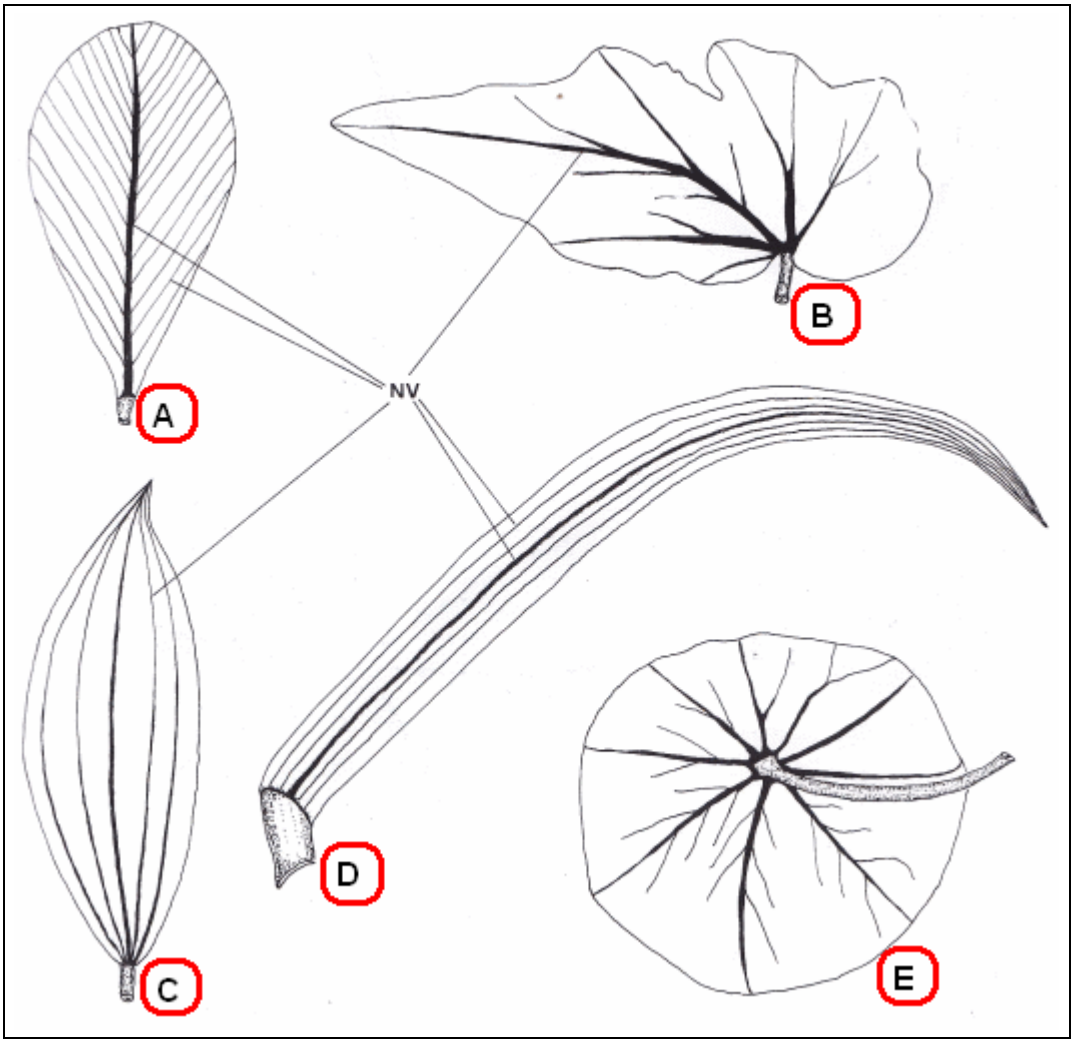
Os órgãos foliares apresentados na Figura 1 denotam exemplos das espécies Hibiscus rosa-sinensis (vulgo graxa-de-estudante, item A), Citrus limonum (limão, item B), Sonchus oleraceus (chicória-brava, item C) e uma folha da espécie Alocasia (família Araceae, no item D).

As partes foliares destacadas na Figura 1 como BF, EL, LB e PC são, respectivamente, a bainha foliar, estípula, limbo e pecíolo.

As estípulas são caracterizados como apêndices localizados na base da folha, geralmente verdes e de formato variável. A bainha constitui a base da folha, que envolve de forma parcial ou completa o caule onde está inserida. O pecíolo caracteriza o pedúnculo que une o limbo à base foliar ou diretamente ao caule e por fim o limbo compõe a parte plaminar da folha (caracterizado por geralmente apresentar-se amplo), com a função de, essencialmente, efetuar a fotossíntese (SOUZA, 2003, p. 131).

Souza (2003, p. 138) descreve que, quando é feita uma referência para a nervação do limbo, isto se tratado padrão apresentado pelos tecidos vasculares presentes na região laminar

da folha. A Figura 2 exibe os tipos de nervação apresentados em órgãos foliares.



Fonte: adaptado de Souza (2003, p. 139).

Figura 2 - Tipos de folhas quanto à nervação

Os tipos de folhas, quando classificadas em respeito à sua nervação são subdivididas em uninérvea, palmatinérvea, estriada e radiada. A folha uninérvea (item A) caracteriza-se por apenas uma nervura longitudinal, central ou principal. A palmatinérvea (item B e item C) é identificada por três ou mais nervuras que partem da base ou logo acima do limbo. Folhas com nervação estriada (item D), apresentam nervuras retilíneas dispostas de forma perpendicular e convergindo para o ápice do limbo. Por fim, órgãos foliares com nervação radiada (item E) são aqueles cujas nervuras partem de um ponto central do limbo (folha peltada) e irradiando para a margem da região laminar (SOUZA, 2003, p. 138).

2.2 SEGMENTAÇÃO DE IMAGENS

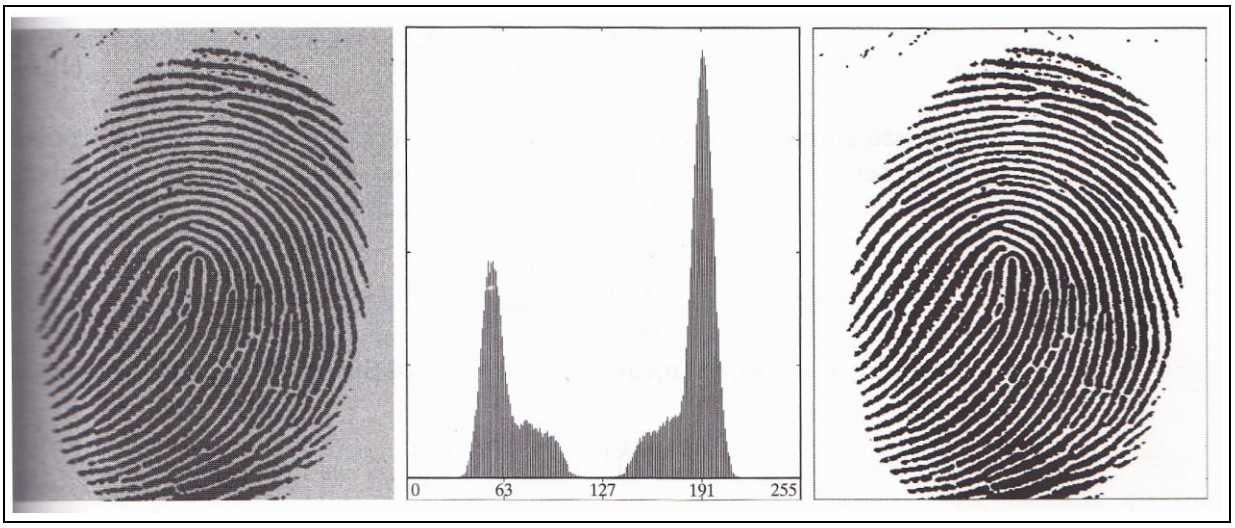
Segmentação é o processo que subdivide uma imagem em objetos ou regiões constituintes. No caso de imagens não triviais, a segmentação mostra-se como uma das tarefas mais difíceis da área de processamento de imagens, sendo que sua precisão determina o eventual sucesso ou falha de procedimentos de análise computadorizada. Um ponto importante neste processo é aplicá-lo apenas o suficiente para a detecção dos objetos ou regiões de interesse, não havendo necessidade se segmentar uma imagem acima do nível de detalhe necessário para a identificação destes elementos (GONZALES; WOODS, 2008, p. 689).

Segundo Costa e Cesar Jr. (2009, p. 230), um elemento visual importante considerado na segmentação de imagens é o contraste entre os objetos e o *background*. Russ (2011, p. 395) indica o processo de *thresholding* como uma técnica simples para a separação entre os objetos de interesse e o *background*. Esta técnica caracteriza-se por selecionar uma faixa de intensidade dos pixels da imagem e classificá-los como sendo integrantes dos objetos de interesse. Esta imagem é então geralmente apresentada de forma binária, efetuando a distinção das regiões discriminadas pelo processo.

Conforme Costa e Cesar Jr. (2009, p. 230), uma das abordagens mais populares de segmentação, considerando regiões de alto contraste entre os objetos e o *background*, é a detecção de bordas, que consiste em identificar conjuntos de pontos correspondentes às maiores áreas de variação em, frequentemente, imagens em escala de cinza.

Segundo Gonzales e Woods (2008, p. 742), o processo de *thresholding* pode ser visto a partir de uma problemática relacionada à decisão estatística, onde o objetivo é minimizar a taxa média de erro ocasionada na classificação dos pixels em dois ou mais grupos. Um método elegante para o thresholding de uma imagem, baseda nesta problemática é o método de Otsu.

A Figura 3 ilustra o resultado da utilização desta técnica.



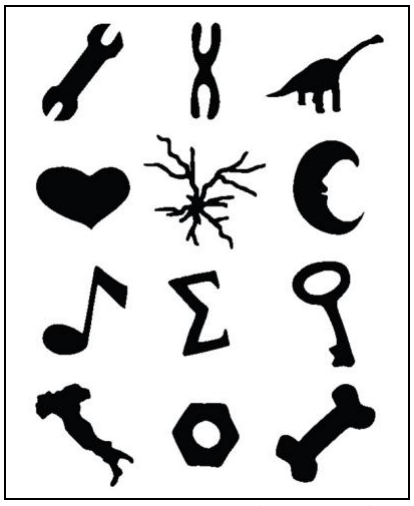
Fonte: adaptado de Gonzales e Woods (2008, p. 743).

Figura 3 - Utilização do método de Otsu para *thresholding*

O método de Otsu é ótimo, visto que ele maximiza a *between-class variance*, medida utilizada em análise discriminante estatística. A idéia básica consiste em que classes devidamente limiarizadas devem apresentar boa distinção no que se refere à intensidade dos valores apresentados em seu histograma (GONZALES; WOODS, 2008, p. 742).

2.3 DESCRITORES DE FOURIER

Dentre os aspectos visuais de um objeto, a forma se sobressai devido à sua importância: pode-se perceber que, mesmo com a falta de informações visuais adicionais (como cor, textura, movimento e profundidade) é possível reconhecer de imediato os objetos apresentados na Figura 4. De modo geral, a formas 2D são frequentemente arquétipos de objetos pertencentes à uma mesma classe (COSTA; CESAR JR, 2009, p. 3).

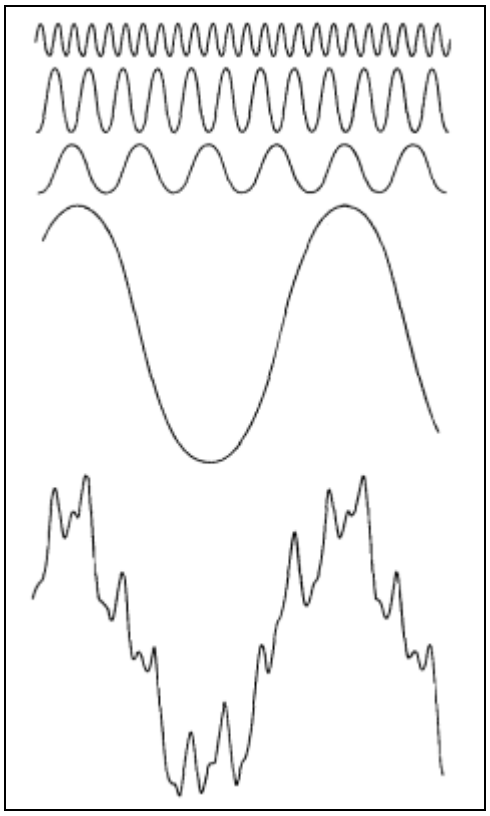


Fonte: Costa e Cesar Jr (2009, p. 4).

Figura 4 – Algumas formas 2D facilmente reconhecidas

De acordo com Russ (2011, p. 610), um método poderoso para descrição de formas matematicamente são os descritores de Fourier. Conforme Costa e Cesar Jr. (2009, p. 441), a idéia básica por trás desta abordagem consiste em representar o objeto de interesse na forma de um sinal unidimensional ou bidimensional, aplicar a transformada de Fourier e então efetuar o cálculo do número de coeficientes desejado para a representação do sinal.

A idéia por trás da teoria empregada na transformada de Fourier é de que dada qualquer função periódica, esta pode ser expressa na forma de uma soma de senos e/ou cossenos de diferentes frequências, cada uma multiplicada por um coeficiente distinto (GONZALES; WOODS, 2008, p. 200). A Figura 5 ilustra a composição de uma função períodica, a partir de tal soma.



Fonte: Gonzales e Woods (2008, p. 201).
Figura 5 - Composição de função periódica conforme Fourier

Segundo Gonzales e Woods (2008, p. 200), a transformada de Fourier possui a importante característica de que, uma função submetida à transformada, pode ser completamente reconstruída a partir da inversão do processo da transformada.

O Quadro 1 e o Quadro 2 ilustram, respectivamente, a *Discrete Fourier Transform* (DFT) de um sinal unidimensional e sua inversa, denominada *Inverse Discrete Fourier Transform* (IDFT).

$$F_m = \sum_{n=0}^{M-1} f_n e^{-j2\pi mn/M} \qquad \text{m = 0, 1, 2, ..., M - 1}$$

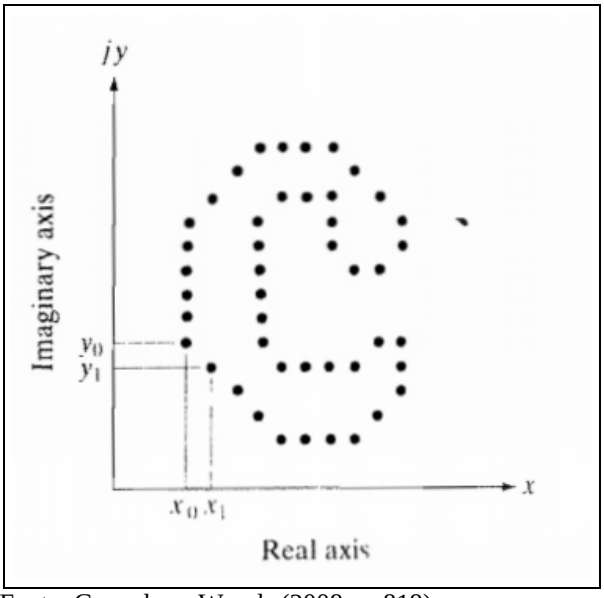
Fonte: Gonzales e Woods (2008, p. 221).
Quadro 1 - Fórmula da DFT

$$f_n = \frac{1}{M} \sum_{m=0}^{M-1} F_m e^{j2\pi mn/M} \qquad \text{n = 0, 1, 2, ..., M - 1}$$

Fonte: Gonzales e Woods (2008, p. 222).
Quadro 2 - Fórmula da IDFT

Nas transformadas apresentadas, *M* denota a quantidade de valores que compõem o sinal, *m* simboliza o m-ésimo coeficiente de Fourier, *n* equivale ao n-ésimo item do sinal e f_n denota o valor deste n-ésimo item.

Segundo Gonzales e Woods (2008, p. 818), os pontos que compõem o contorno de uma imagem podem ser representados na forma de um número complexo. A Figura 6 exibe a relação entre os eixos de um plano cartesiano e os componentes de um número complexo.



Fonte: Gonzales e Woods (2008, p. 818).
Figura 6 - Relação entre eixos cartesianos e números complexos

O Quadro 3 exibe a representação de um par de coordenadas na forma de número complexo.

$$s(k) = x(k) + jy(k)$$

Fonte: Gonzales e Woods (2008, p. 818).
Quadro 3 - Representação de coordenada cartesiana em número complexo

Com a representação ilustrada no Quadro 3, é possível representar um vetor contendo *k* pontos cartesianos (oriundos do contorno da imagem) com um vetor de *k* números complexos, conforme o Quadro 4.

$$[(x_0, y_0), (x_1, y_1), \dots (x_{k-1}, y_{k-1})] \Rightarrow [s_0, s_1, \dots s_{k-1}]$$

Quadro 4 - Representação de um vetor de pontos cartesianos em números complexos

Segundo Zhang e Lu (2005), os processos de reconhecimento de objetos devem ser invariantes à rotação, escala e translação. Sendo assim, faz-se necessária a computação de descritores de Fourier que sejam invariáveis à tais transformações.

Para tornar o processo de descrição de uma forma invariável à translação, Zhang e Lu (2005) sugerem, em vez de números complexos representando as coordenadas dos pontos do contorno, números reais simbolizando as distâncias entre os pontos do contorno e a centróide da figura.

A invariância à escala da figura é obtida a partir da normalização de escala da imagem. Considerando apenas as magnitudes dos coeficientes resultantes da DFT, é possível tornar os

descritores imunes à mudanças na escolha do ponto inicial do vetor do contorno (rotação, no caso da utilização da distância a partir da centróide).

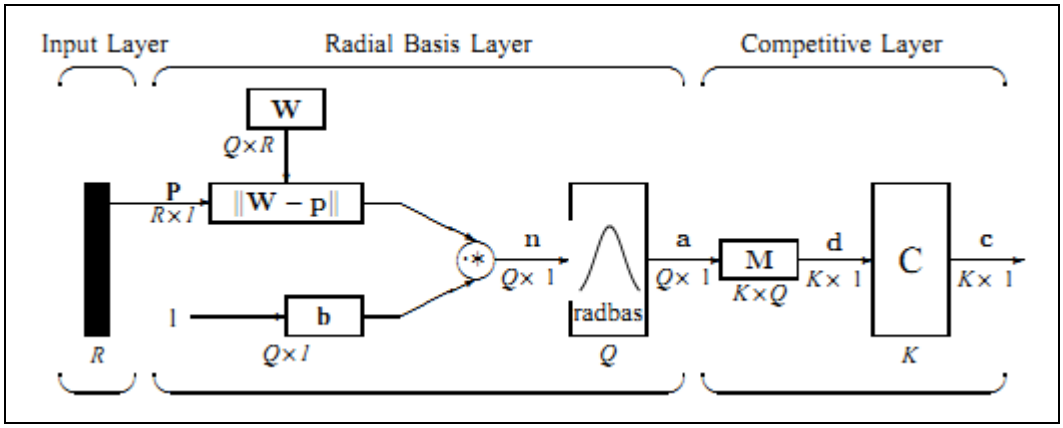
2.4 REDE NEURAL PROBABILÍSTICA (RNP)

Uma Rede Neural Artificial (RNA) pode ser definida como um conjunto de nêurons artificais simulando o funcionamento do cérebro humano, podendo inclusive ser considerada uma “caixa preta mágica” treinada para alcançar um processo inteligente esperado, a partir de um conjunto de informações de entrada e saída (WU et al., 2007).

Segundo Wu et al. (2007), um dos tipos de RNA é denominado *Radial Basis Function Network* (RBFN), que utiliza internamente uma função cuja imagem assemelha-se ao formato de um sino. Uma Rede Neural Probabilística (RNP) é um tipo de rede neural que deriva da RBFN.

Conforme Singh, Gupta e Gupta (2010), uma RNP trabalha computando a distância entre o vetor de entrada e cada vetor de treinamento, gerando um novo vetor indicando o quão próximo o vetor de entrada está dos vetores de treinamento. Então, para classe de entradas possíveis, é feita uma soma da contribuição destes vetores de distância, determinando a probabilidade do vetor de entrada pertencer à cada classe registrada. Por fim, a classe do vetor de entrada é determinada a partir da seleção da maior probabilidade calculada entre as classes registradas.

A Figura 7, exibe a estrutura de uma RNP.



Fonte: Wu et al. (2007).

Figura 7 - Arquitetura de uma RNP

Na Figura 7, a barra preta na camada de entrada indica o vetor (de dimensão *R*) de valores a ser classificado. *W* denota o conjunto de *Q* vetores armazenados como base de comparação, de mesma dimensão *R*, cujas distâncias para com o vetor de entrada são calculadas. Cada vetor da base de comparação possui um bias (contido no vetor *b*), um coeficiente de suavização aplicado à função de distância dos vetores. *M* denota a matriz de peso da camada competitiva e *C* a função competitiva, que elege o maior valor adquirido a partir da camada de base radial como o selecionado.

Segundo Singh, Gupta e Gupta (2010), a RNP possui várias vantagens, dentre elas o curto tempo de processamento gasto no treino (em comparação à outros processos de treino de rede neurais, como *Back Propagation*), a simplicidade de sua estrutura e processo de treinamento, sua robustez à ruídos nos valores e a possibilidade do seu resultado convergir para o do classificador ótimo de Bayes (uma vez que amostras sufientes para treino sejam oferecidas).

2.5 TRABALHOS CORRELATOS

Esta seção destina-se a apresentar alguns dos trabalhos voltados ao reconhecimento de plantas utilizando processamento de imagens digitais, empregando técnicas de segmentação, extração de características e classificação em espécies de plantas previamente cadastradas.

Wu et al. (2007) apresentaram uma solução que abordou as etapas de segmentação, extração de características e classificação das plantas. Para efetuar a segmentação, a imagem em cores de entrada foi convertida para uma nova em escala de cinza, aplicada técnica de *thresholding* para binarizar a imagem, separando a área correspondente à folha do fundo da imagem. A imagem binarizada é então submetida à um filtro de Laplace[1], onde o resultante acaba sendo apenas o contorno da folha. Após o usuário informar dois pontos correspondentes aos extremos da folha, são computadas características baseadas na geometria da região da folha, como diâmetro, comprimento, largura, área e perímetro em adição as nervuras das folhas. Por fim, estas características são utilizadas como entrada para o mecanismo de

[1] Filtro isotrópico que destaca regiões em uma imagem onde ocorrem descontinuidades de intensidade enquanto atenua regiões com baixa variação de intensidade, comumente utilizado no realce e segmentação de imagens (GONZALES; WOODS, 2008, p. 160).

classificação, feito a partir da utilização de uma Rede Neural Probabilística (RNP), técnica escolhida devido à sua velocidade no processo de treinamento, precisão e estrutura simples.

O trabalho desenvolvido por Singh, Gupta e Gupta (2010) foca em uma comparação entre três métodos de classificação selecionados para a classificação de plantas a partir de folhas: RNP, *Fourier Moments* (FM) e *Support Vector Machines* estruturadas em *Binary Decision Trees* (SVM-BDT). Neste trabalho tanto a segmentação das imagens quanto a extração das características da folha são feitas de modo similar ao trabalho de Wu et al. (2007). Como informações de entradas para RNP e SVM-BDT foram utilizadas as características baseadas na geometria e nervuras da folha, porém, para FM a informação utilizada foi 512 descritores obtidos a partir da aplicação da *Fast Fourier Transform* (FFT) em cima de uma sequência de distâncias radiais do contorno da folha. Após a avaliação dos resultados, este trabalho concluiu que SVM-BDT gerou os melhores resultados para a base de imagens utilizada, em termos de precisão.

Chaki e Parekh (2012) apresentam, em seu trabalho, uma técnica relativamente simples de classificação baseada na intersecção de imagens binárias de folhas. A técnica em questão, denominada *Binary Superposition* (BS), é comparada com outras duas técnicas de representação de formas: *Moment Invariants* (MI) e *Centroid-Radii* (C-R). A técnica MI trabalha com uma entrada que considera a intensidade dos pixels da imagem. C-R necessita de uma imagem binária da imagem da folha para então efetuar a extração dos contornos da folha, sendo utilizado o detector de bordas de Canny para este trabalho. Por fim, BS trabalha utilizando apenas imagens binárias das folhas. O método proposto consiste em multiplicar uma imagem binária da folha a ser reconhecida com cada uma das imagens da base disponível e somar a quantidade de pixels com valor diferente de zero, agrupando por espécie de planta. Após a soma, a espécie de planta que detenha o maior valor é designada como provável espécie da folha analisada. O trabalho conclui que BS apresenta os melhores resultados em relação às outras técnicas, em termos de precisão na classificação.

Kadir et al (2011) optaram pela extração de uma gama maior de características, que incorporam descritores geométricos (finura, circularidade e irregularidade), descritores de Fourier extraídos a partir da *Polar Fourier Transform* (PFT), descritores estatísticos da cor (média, desvio padrão, *skewness* e *kurtosis*) de cada um dos canais RGB, descritores das nervuras da folha (a partir da razão da área das nervuras em relação à área da folha) e ainda descritores de textura da folha baseados na medida de lacunaridade de fractais. Foram feitas diversas combinações das características citadas e empregadas como entrada para uma RNP, registrando para cada combinação a precisão de classificação das espécies de plantas.

O Quadro 5 apresenta de forma comparativa algumas características dos trabalhos apresentados nesta seção.

características / trabalhos relacionados	Wu et al. (2007)	Singh, Gupta e Gupta (2010)	Chaki e Parekh (2012)	Kadir et al. (2011)
mecanismo de classificação	RNP	SVM-BDT	Máxima	RNP
espécies diferentes	32 (Flavia)	32 (Flavia)	3	32 (Flavia)
características utilizadas	geométricas, nervuras	geométricas, nervuras	silhueta	geométricas, nervuras, descritores de Fourier, cor, textura
invariância geométrica	translação, escala, rotação	translação, escala, rotação	-	translação, escala, rotação
número de descritores	5	5	-	62
precisão média	90,3%	96%	99%	93,75%

Quadro 5 - Características dos trabalhos relacionados

A partir do Quadro 5, é possível identificar que os trabalhos utilizaram o mesmo banco de imagens, denominado Flavia *dataset*. Este banco, atualmente constituído de 32 espécies diferentes de folhas com uma média de 60 imagens por espécie, foi construído ao longo do trabalho desenvolvido por Wu et al. (2007) e disponibilizado gratuitamente com o intuito de auxiliar futuros trabalhos relacionados.

A partir dos resultados obtidos por Kadir et al. (2009), torna-se evidente que a utilização de características geométricas e nervuras da planta, como descritores morfológicos, descritores de cor e textura, contribuem para um melhor resultado no processo de classificação.

3 DESENVOLVIMENTO

Neste capítulo serão apresentadas as etapas do desenvolvimento da API de reconhecimento de plantas. Na seção 3.1 são enumerados os requisitos principais do projeto desenvolvido. A seção 3.2 apresenta a especificação da API, contendo a arquitetura do projeto e sua base de dados. A seção 3.3 detalha a implementação das principais técnicas e algoritmos utilizados pela API. Por fim, a seção 3.4 apresenta a série de testes efetuados para a validação do projeto e os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF) da API de reconhecimento de plantas são:

a) identificar na imagem de entrada a região correspondente à folha da planta (RF);
b) extrair descritores a partir das características estruturais das folhas (RF);
c) disponibilizar um mecanismo de treinamento para a inclusão e o reconhecimento de novas espécies de plantas (RF);
d) classificar as folhas de acordo com sua espécie (RF);
e) ser implementado utilizando a linguagem C/C++ (RNF);
f) ser implementado utilizando o ambiente de desenvolvimento Microsoft Visual Studio (RNF);
g) ser desenvolvido para a plataforma Windows (RNF);
h) ser implementado utilizando a biblioteca OpenCV (RNF).

3.2 ESPECIFICAÇÃO

A especificação da API foi representada em diagramas da *Unified Modeling Language* (UML), utilizando a ferramenta Enterprise Architect juntamente com um Modelo de Entidade-Relacionamento (MER), descrevendo a estrutura da base de dados da API. Neste

trabalho, foram elaborados os diagramas de casos de uso, classes e MER, sendo descritos nas próximas seções.

3.2.1 Diagrama de casos de uso

A Figura 8 exibe o diagrama de casos de uso com as ações disponibilizadas pelo aplicativo de demonstração da API. Identificou-se apenas um ator, denominado `Usuário`, o qual utiliza todas as funcionalidades do aplicativo.

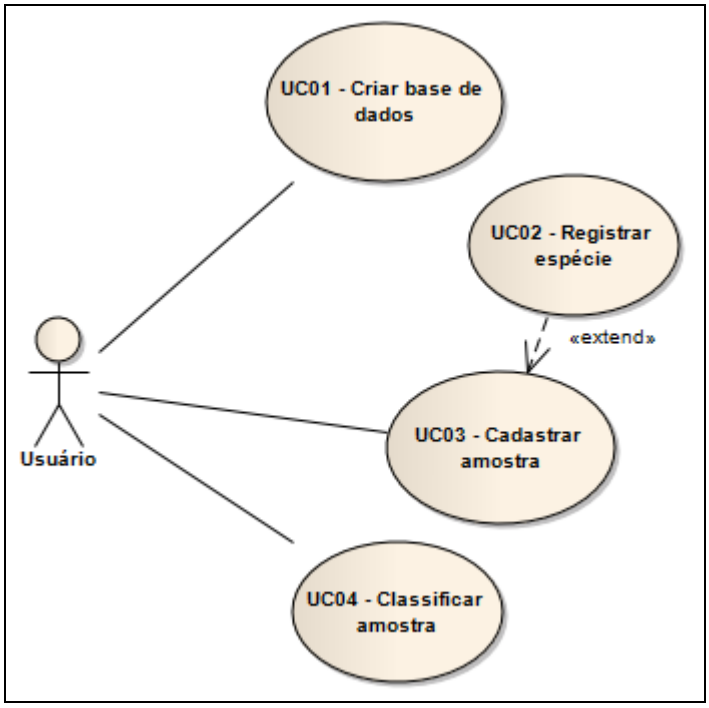


Figura 8 - Diagrama de casos de uso

Segue detalhamento dos casos de uso exibidos no diagrama da Figura 8:

a) `UC01 – Criar base de dados`: permite a criação de uma nova base de dados para classificação de exemplares, indicando quais são as características e parâmetros a serem adotados para a classificação;

b) `UC02 – Registrar espécie`: permite adicionar ao catálogo de espécies (na base de dados da API) uma nova espécie de planta;

c) `UC03 – Cadastrar amostra`: permite adicionar na base de dados uma nova amostra de uma espécie de planta específica;

d) `UC04 – Classificar amostra`: permite classificar uma amostra dentre as espécies de plantas previamente registradas na base de dados.

3.2.2 Diagrama de pacotes

Para facilitar a visualização e o entendimento do relacionamento entre as classes, optou-se por agrupá-las em pacotes de acordo com sua especificidade. A Figura 9 exibe o diagrama de pacotes que compõem a API.

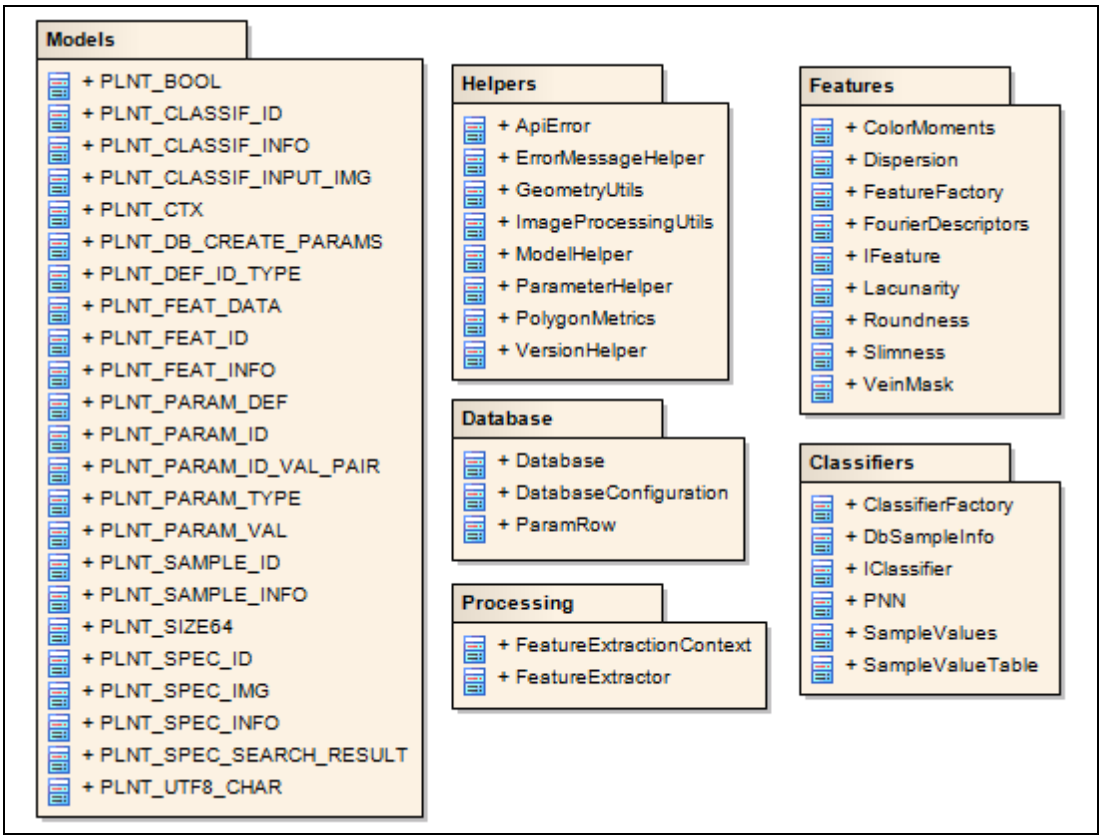


Figura 9 - Resumo de classes por categoria

Nas próximas seções serão detalhadas as classes internas de cada pacote, elucidando suas responsabilidades dentro da API. Na seção 3.2.2.1 será descrito o pacote `Models`, que contém os tipos de dados e estruturas utilizadas para comunicação com a API. A seção 3.2.2.2 contém o detalhamento do pacote `Database`, que comporta as classes responsáveis pelo acesso à base de dados da API. A seção 3.2.2.3 apresenta as classes contidas no pacote `Features`, implementações de características utilizadas para discriminar as diferentes amostras de espécies de plantas. Na seção 3.2.2.4 estão descritas as classes presentes no pacote `Classifiers`, que compõem a implementação dos classificadores (utilizados para determinar as espécies das amostras) disponibilizados pela API. A seção 3.2.2.5 exibe o pacote `Processing`, contendo entidades responsáveis pelo processo de iteração pelas carcacterísticas utilizadas na base de dados da API, invocando a computação dos descritores de cada uma delas. Por fim, na seção 3.2.2.6, são descritas as classes presentes no pacote

`Helpers`, que representam um conjunto de entidades contendo funções utilitárias ou auxiliares utilizadas pelos demais pacotes contidos no projeto.

3.2.2.1 Modelos de dados públicos

A interface de acesso à API, disponibilizada para programas externos, compreende uma série de funções públicas que encapsulam a criação e utilização das classes (internas) desenvolvidas no projeto. Tais funções utilizam-se de argumentos com tipos de dados definidos pela própria API, apresentados na Figura 10.

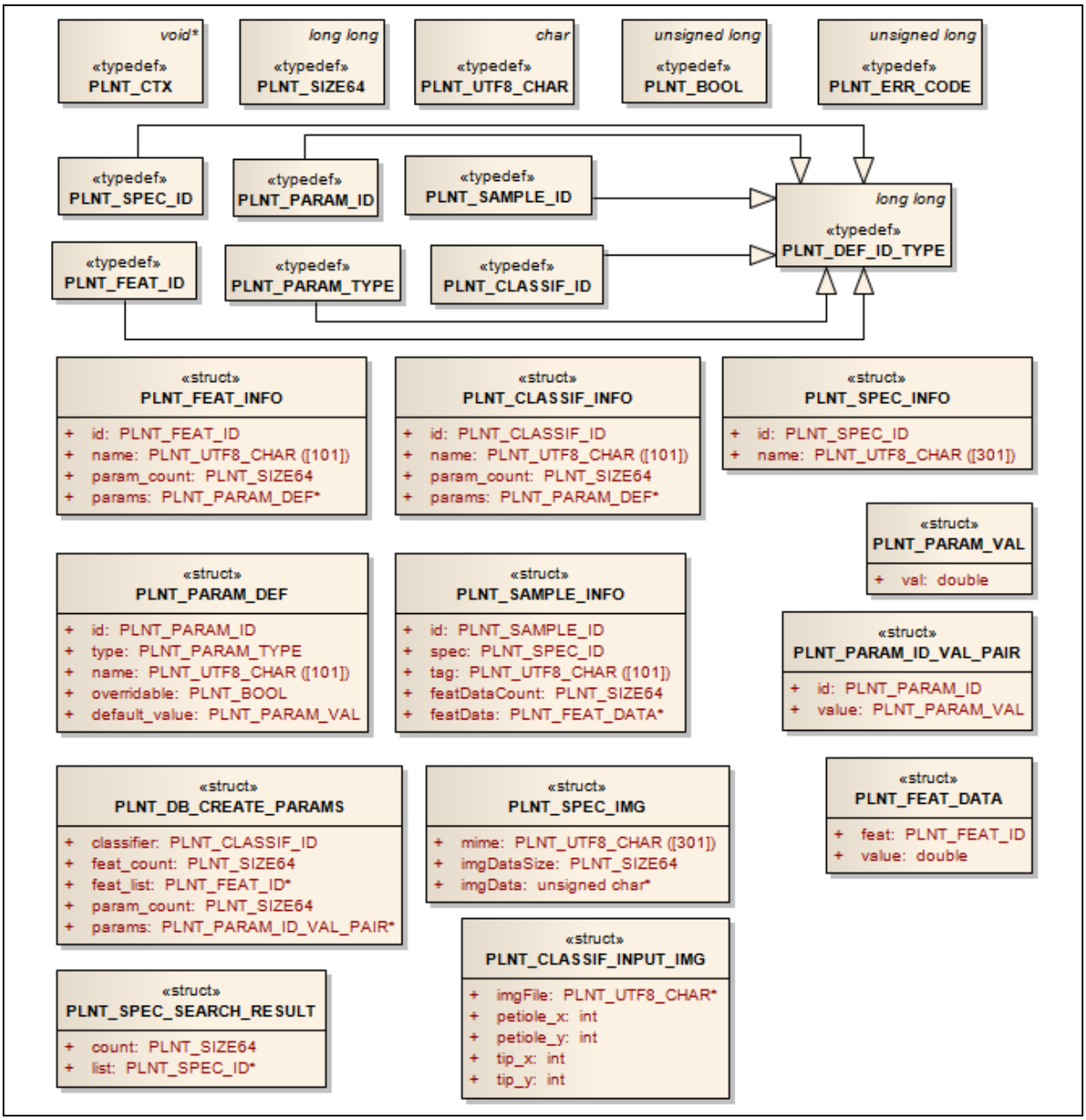


Figura 10 - Estruturas de dados exportadas pela API

O tipo PLNT_CTX é utilizado como contexto (sessão) do aplicativo usuário da API. Neste objeto são armazenadas informações cujo escopo estendem-se até o momento em que o aplicativo finaliza o uso da API, tais informações são consumidas no momento da chamada de várias funções contidas na API.

Valores lógicos utilizados nos modelos são representados pelo tipo PLNT_BOOL e integrais simbolizando tamanhos de coleções e estruturas de dados são indicados pelo tipo PLNT_SIZE64.

Informações no formato de texto (*strings*) são utilizadas encodadas em UTF8 e representadas por *arrays* do tipo PLNT_UTF8_CHAR.

Eventuais erros que possam ocorrer na execução das funções da API estão mapeados em identificadores no formato de valores númericos, representados pelo tipo integral PLNT_ERR_CODE.

Identificadores utilizados pela API derivam de um tipo padrão denominado PLNT_DEF_ID_TYPE: PLNT_SPEC_ID, PLNT_FEAT_ID, PLNT_CLASSIF_ID, PLNT_SAMPLE_ID, PLNT_PARAM_ID e correspondem respectivamente aos identificadores de espécies, características, classificadores, amostras e parâmetros da API. Os tipos de dados dos parâmetros são representados pelo tipo PLNT_PARAM_TYPE.

As estruturas PLNT_FEAT_INFO e PLNT_CLASSIF_INFO representam, respectivamente, dados de características e classificadores presentes na API. Nelas têm-se informações como seu identificador, nome e parâmetros relevantes para sua execução. Estes parâmetros são descritos pela estrutura PLNT_PARAM_DEF, que contém informações sobre o código do parâmetro, nome, tipo, valor padrão e um indicador se existe a possibilidade de alterá-lo após a criação da base de dados.

Espécies de plantas registradas na base de dados da API são representadas pela estrutura PLNT_SPEC_INFO, contendo o código e o nome da espécie. Caso a espécie, para fins ilustrativos, possuir uma imagem associada ao seu cadastro, esta informação é representada pela estrutura PLNT_SPEC_IMG contendo o *array* de bytes que compõe a imagem juntamente com seu *mime type*. Os resultados de pesquisas por espécies são disponibilizados no formato do tipo PLNT_SPEC_SEARCH_RESULT, contendo os identificadores das espécies que atendem aos critérios da consulta.

Valores de parâmetros, independentemente dos tipos nominais, são armazenados como valores numéricos de ponto flutuante e representados pela estrutura PLNT_PARAM_VAL. Associações entre os parâmetros e estes valores são feitos a partir da estrutura

PLNT_PARAM_ID_VAL_PAIR.

Valores descritores computados pelas características suportadas pela API são descritos pela estrutura PLNT_FEAT_DATA, contendo o identificador da característica e o valor processado.

Exemplares de plantas utilizados como base de treino, após o processamento de suas características e adição na base de dados da API, são descritos pela estrutura PLNT_SAMPLE_INFO, que contempla informações como o identificador da amostra e espécie à que ela pertence, a coleção de valores descritores que a representam e ainda um possível rótulo, em formato textual, escolhido pelo usuário.

A estrutura PLNT_DB_CREATE_PARAMS é utilizada no momento da criação da base de dados e contém informações sobre a configuração das características, classificador utilizado para as amostras e parâmetros de execução tanto das características quanto do classificador.

As rotinas de classificação de amostra utilizam-se das informações armazenadas no tipo PLNT_CLASSIF_INPUT_IMG. Esta estrutura comporta o nome do arquivo de imagem a ser analisado e as coordenadas cartesianas (informadas pelo usuário) do pecíolo e da ponta da folha sendo amostrada.

3.2.2.2 Base de dados

A API desenvolvida armazena as informações de espécies, parâmetros, características, classificadores e amostras dentro de uma base de dados simples. O acesso à estas informações é feito através das classes definidas na Figura 11.

Figura 11 - Classes de acesso à base de dados

O acesso à base de dados é feito unicamente através da classe `Database`, que encapsula um objeto do tipo `CppSQLite3DB` contendo uma conexão à uma base de dados SQLite. Basicamente, objetos da classe `Database` disponibilizam métodos para criação e abertura de um arquivo de base de dados, obtenção de características e classificadores cadastrados, listagem e cadastro de espécies e amostras assim como o informe de valores dos parâmetros de execução.

A estrutura `ParamRow` contém atributos que descrevem o valor de um parâmetro assim como o objeto (característica ou classificador) ao qual o parâmetro pertence. A classe `DatabaseConfiguration` é utilizada logo após a criação da base de dados e contém informações sobre o classificador, características e parâmetros a serem utilizados em uma

respectiva base de dados. Este objeto é utilizado no processo de configuração de uma base de dados, que ocorre logo após sua criação.

3.2.2.3 Características

As características implementadas pela API compreendem circularidade, dispersão, magreza, descritores de Fourier, máscara de nervura, lacunaridade e estatísticas de cor. Tais características são atributos ou aspectos que auxiliam na distinção (caracterização) das espécies de plantas. A Figura 12 e a Figura 13 apresentam as entidades responsáveis pela extração de valores descritores a partir do processamento de imagens das plantas (amostras).

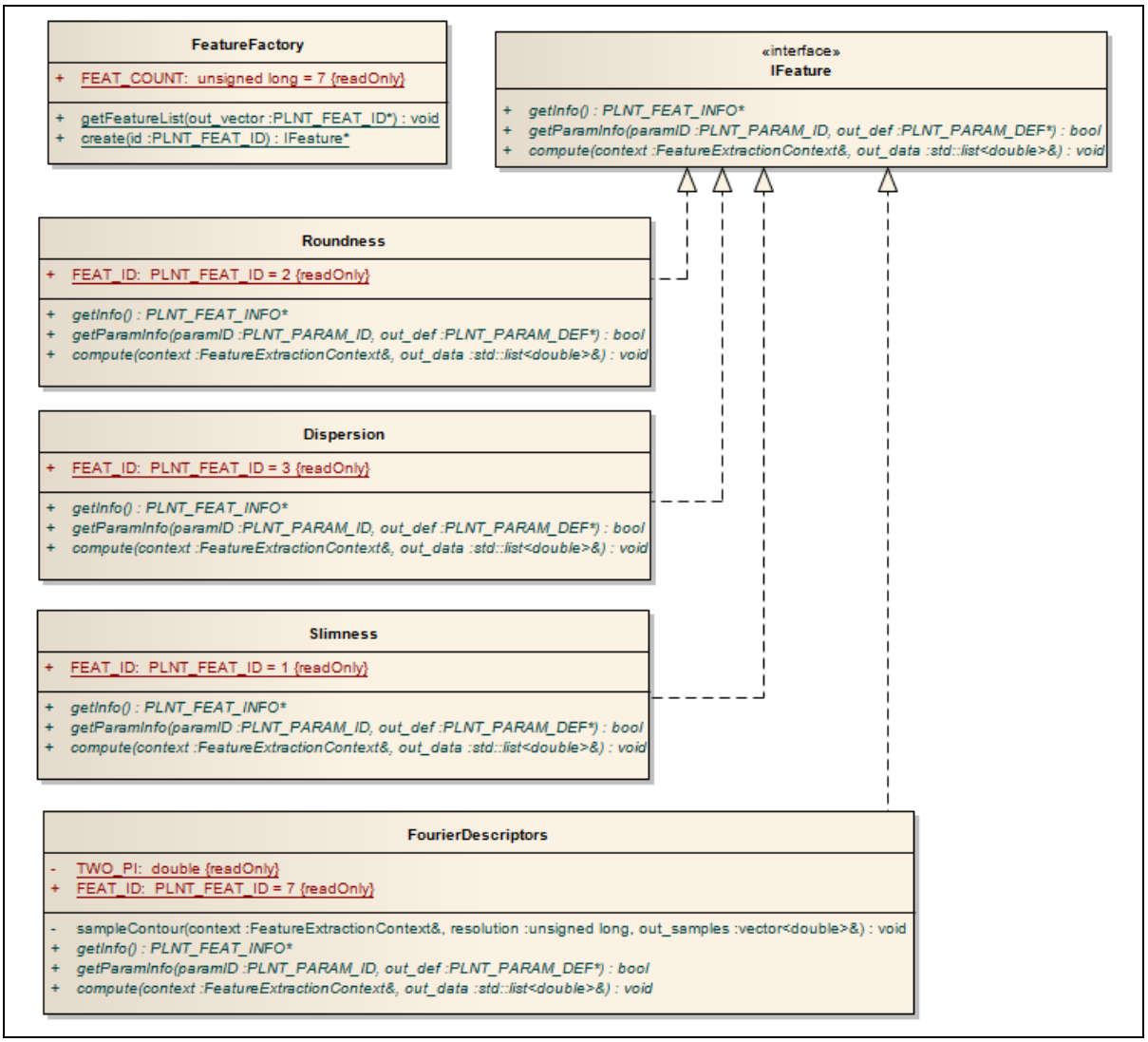


Figura 12 – Diagrama de classes de características 1

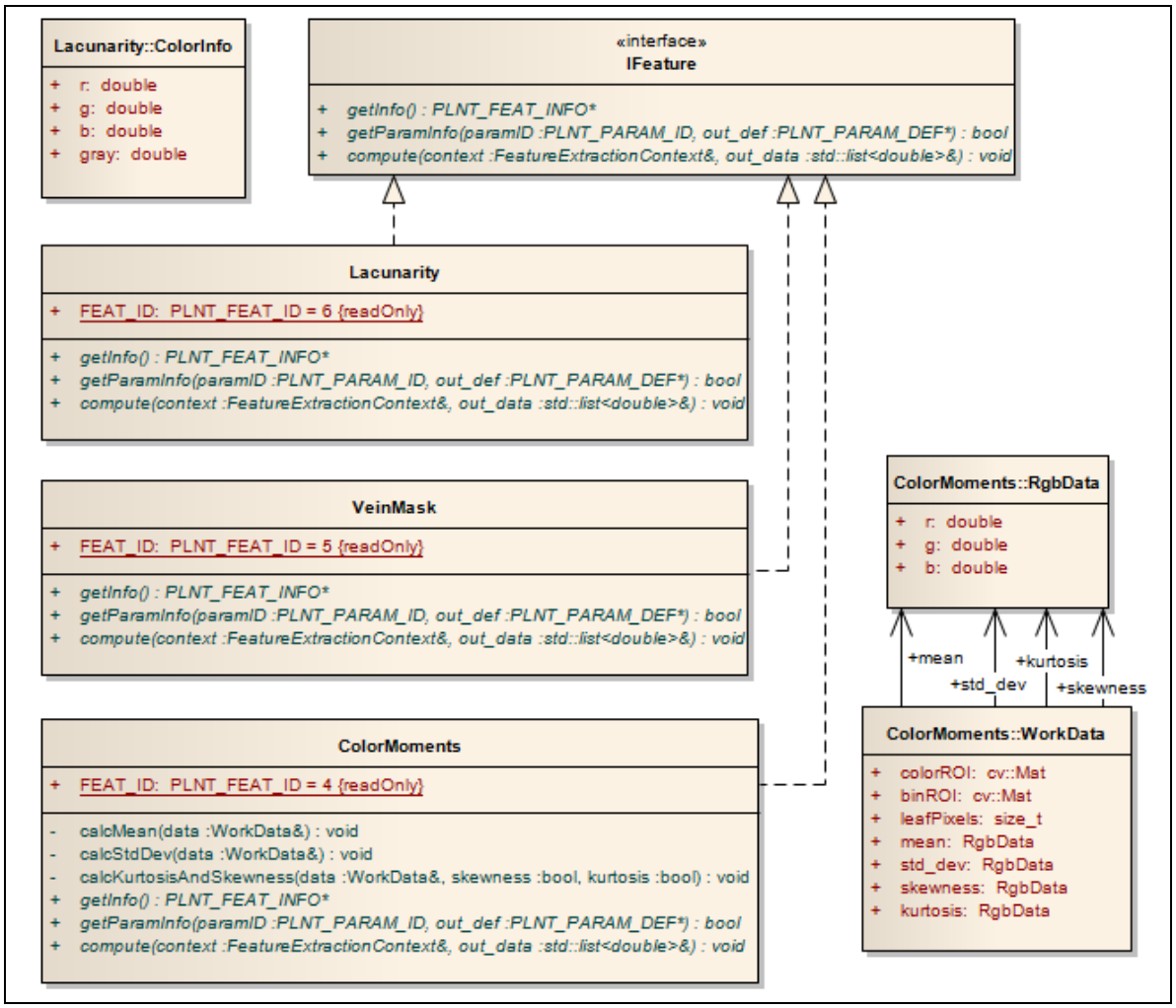


Figura 13 – Diagrama de classes de características 2

Todas as características extraídas pela API implementam a interface `IFeature` e são instanciadas a partir da classe `FeatureFactory`.

O contrato com a interface `IFeature` determina que todas as características presentes na API forneçam informações sobre sua identificação (através do método `getInfo`) assim como informações sobre a utilização de algum parâmetro em específico (por meio do método `getParamInfo`). O método `compute` calcula um conjunto de valores descritores que serve como quantificador da característica e o armazena como parâmetro de saída (`out_data`).

3.2.2.4 Classificadores

Classificadores são utilizados para determinar a espécie que uma amostra pertence, a partir do processamento de vetores de valores descritores da base de treino e da amostra a ser classificada. As classes relacionadas ao processo de classificação são exibidas na Figura 14.

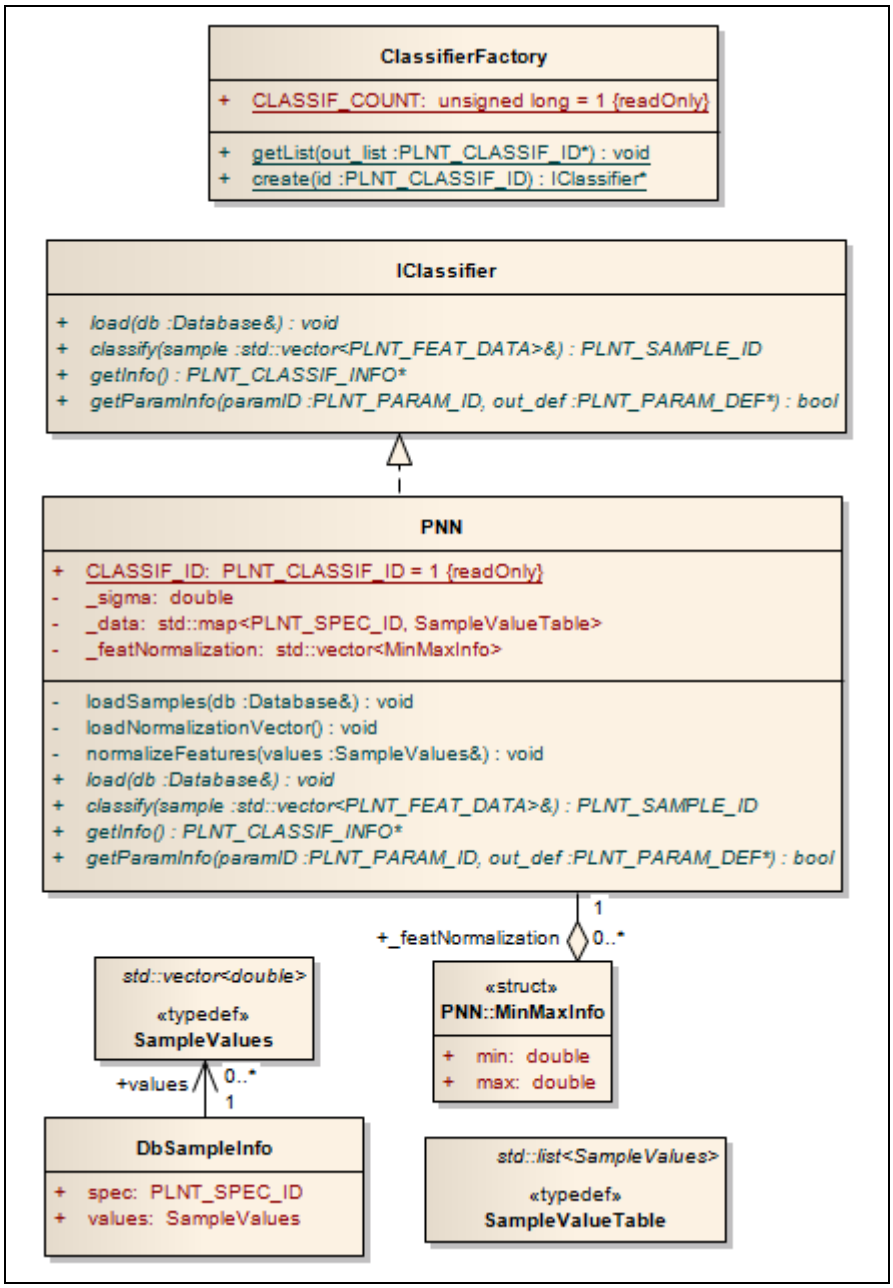


Figura 14 – Diagrama de classes de classificadores

Todos os classificadores disponibilizados pela API implementam a interface `IClassifier` e são instanciados a partir da classe `IClassifierFactory`.

O contrato com a interface `IClassifier` determina que classificadores presentes no protótipo forneçam informações sobre sua identificação (através do método `getInfo`) e sobre seus parâmetros de execução (através do método `getParamInfo`). O método `load` é responsável por efetuar o carregamento dos dados necessários para o processo de classificação. O método `classify` determina a espécie de uma amostra a partir do seu vetor de valores descritores.

A classe `PNN` representa o único tipo de classificador presente na API, implementando

uma Rede Neural Probabilística (RNP) com normalização automática dos vetores de dados utilizando a técnica *Min-Max*.

3.2.2.5 Extração de características

As classes apresentadas na Figura 15 são responsáveis por solicitar a computação das características (valores descritores) a partir de uma imagem de entrada.

Figura 15 – Classes responsáveis pela extração de características

A classe `FeatureExtractor` é responsável, através do método `extractFeatures`, pela criação de um novo contexto de extração de características (`FeatureExtractionContext`). A partir deste contexto, percorre-se cada característica registrada na base de dados requisitando a computação dos valores descritores para a amostra informada.

Objetos da classe `FeatureExtractionContext` disponibilizam acesso à recursos comumente utilizados na computação de valores descritores de características, como a base de dados que atualmente está sendo utilizada, as imagens da amostra em cores, escala de cinza e binária preto e demais informações.

3.2.2.6 Classes auxiliares

Na Figura 16 são exibidas as classes restantes da API, em sua maioria utilitários que agrupam funções frequentemente utilizadas pelos demais componentes do protótipo.

Figura 16 - Classes auxiliares

A classe de exceção `ApiError` é internamente utilizada para simbolizar erros de execução do protótipo antecipadamente mapeados, contendo o código específico do erro. Para a recuperação da descrição de um erro mapeado da API, internamente é utilizado o método `getErrorMessage`, presente na classe assistente `ErrorMessageHelper`.

A partir do método `getVersionString`, presente na classe auxiliar `VersionHelper`, a API disponibiliza informações sobre o número de sua versão e plataforma.

A classe assistente `ModelHelper` provém métodos utilizados para a destruição de

estruturas complexas exportadas (liberando a memória utilizada por tais objetos), assim como um método para a inicicialização simples da estrutura `PLNT_PARAM_DEF`.

A classe `ParameterHelper` auxilia em tarefas relacionadas à manipulação de parâmetros da API, como a obtenção da definição de um parâmetro a partir de seu identificador, descobrimento do classificador ou característica a qual um parâmetro específico pertence e rotinas que computam a sobrescrita de valores padrões de parâmetros.

A classe utilitária `GeometryUtils` contém uma coleção de métodos que auxiliam em tarefas relacionadas ao cálculo de medidas geométricas e álgebra linear, normalmente trabalhando em cima de pontos de um plano cartesiano 2D ou vetores. A estrutura `PolygonMetrics` é utilizada para armazenar informações básicas sobre as medidas de um polígono.

Por fim, na classe `ImageProcessingUtils` estão presentes métodos que implementam operações básicas em cima de matrizes contendo o mapa dos pixels de uma imagem. Este objeto é composto por operações como conversão de imagens de cores para escala de cinza e binárias, extração de contorno e cálculo de média dos valores dos canais RGB.

3.2.3 Modelo de Entidade-Relacionamento

As informações de espécies, amostras, características, classificador e parâmetros utilizados pela API são mantidos em uma base de dados relacional simples, específica e composta por apenas 1 (um) arquivo, visando a fácil distribuição das informações ali contidas.

As entidades, apresentadas na Figura 17, em sua maioria armazenam os identificadores de objetos implementados na API que são utilizados na base de dados específica, em conjunto com os dados das amostras de plantas e espécies cadastradas.

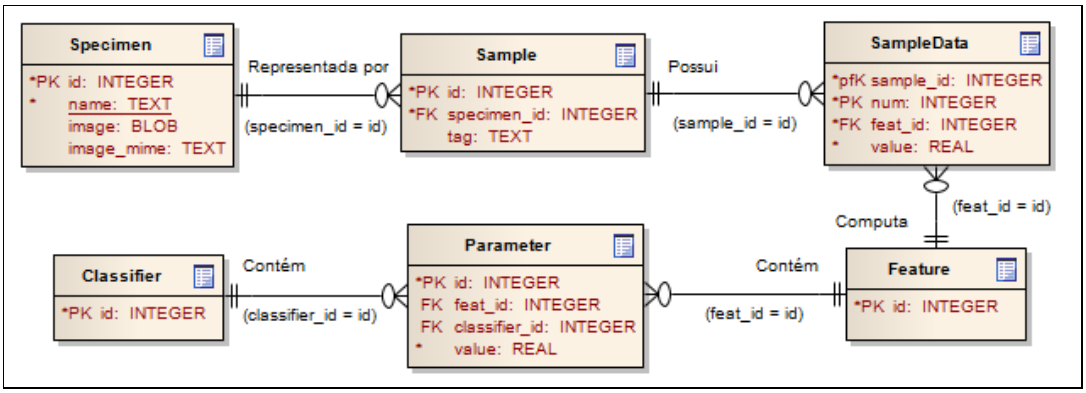


Figura 17 - Modelo de Entidade-Relacionamento

A entidade `Classifier` armazena o identificador do classificador escolhido para a

base de dados, então contém normalmente apenas 1 (um) registro.

As características selecionadas pelo usuário no momento da criação da base de dados têm seus identificadores armazenados na entidade `Feature`.

A entidade `Parameter` armazena os parâmetros configurados para a base de dados, contendo o identificador do parâmetro, o valor informado e relacionamentos com as entidades `Feature` e `Classifier`, indicando qual o componente a que o parâmetro pertence.

Espécies de plantas registradas pela API são armazenadas na entidade `Specimen`, onde ficam contidos o nome da espécie e as informações da imagem ilustrativa vinculada à espécie.

Cada amostra incluída na base de dados, fica armazenada na entidade `Sample`, que contém um vínculo com a espécie a que pertence e a informação de rótulo passada no momento de sua inclusão. Os valores descritores extraídos pelas características são armazenados na entidade `SampleData` e associados à amostra.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

A API foi desenvolvida utilizando como plataforma as linguagens C e C++, oferecidas pela plataforma Microsoft Visual C++. Como ambiente de desenvolvimento foi selecionada a IDE Microsoft Visual Studio 2010.

As tecnologias utilizadas na implementação do projeto são enumeradas a seguir:

a) OpenCV: Biblioteca de visão computacional que, em sua versão 2.3.1, foi utilizada como base nas rotinas de processamento de imagem, onde foram aproveitados algoritmos como de abertura de arquivos de imagem, conversão para imagens em tons de cinza e segmentação, operações morfológicas, extração de contornos e outros. Sua interface de acesso é disponibilizada em C e C++;
b) SQLite: Empregada em sua versão 3.7.14.1, é uma biblioteca que implementa uma

engine de banco de dados relacional que não necessita da instalação de serviços. É utilizada neste projeto para a persistência das informações contidas no MER. O acesso à biblioteca é feita a partir de sua interface em C;

c) CppSQLite: Utilizado na versão 3.2, compreende um conjunto de classes que encapsulam as estruturas e chamadas à biblioteca SQLite, provendo uma abordagem orientada à objetos em respeito ao acesso à engine de base de dados;

d) Windows Forms: API disponibilizada pelo .NET Framework para desenvolvimento de interfaces gráficas que encapsula as chamadas à API do Windows. A aplicação de demonstração da API foi construída utilizando Windows Forms e a plataforma .NET Framework 4.

O processo para o registro das amostras de plantas na base de dados compreende, conforme exibido na Figura 18, a aquisição dos dados de entrada, computação dos valores descritores das características e armazenamento dos valores descritores.

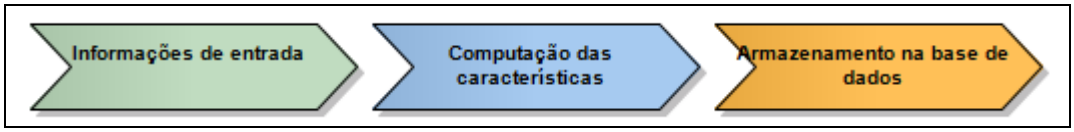


Figura 18 - Processo básico do registro de amostras

Algumas informações extraídas a partir dos dados de entrada (como a imagem binarizada, a imagem em escala de cinza, contorno, etc) são denominadas recursos e servem como base para o cálculo de diversas características implementadas na API. Tais informações são agrupadas em um contexto de extração de características, sendo repassadas para as rotinas de computação de cada característica, fazendo com que esses recursos não necessitem ser construídos por cada característica. A Figura 19 exemplifica o fluxo dessas informações.

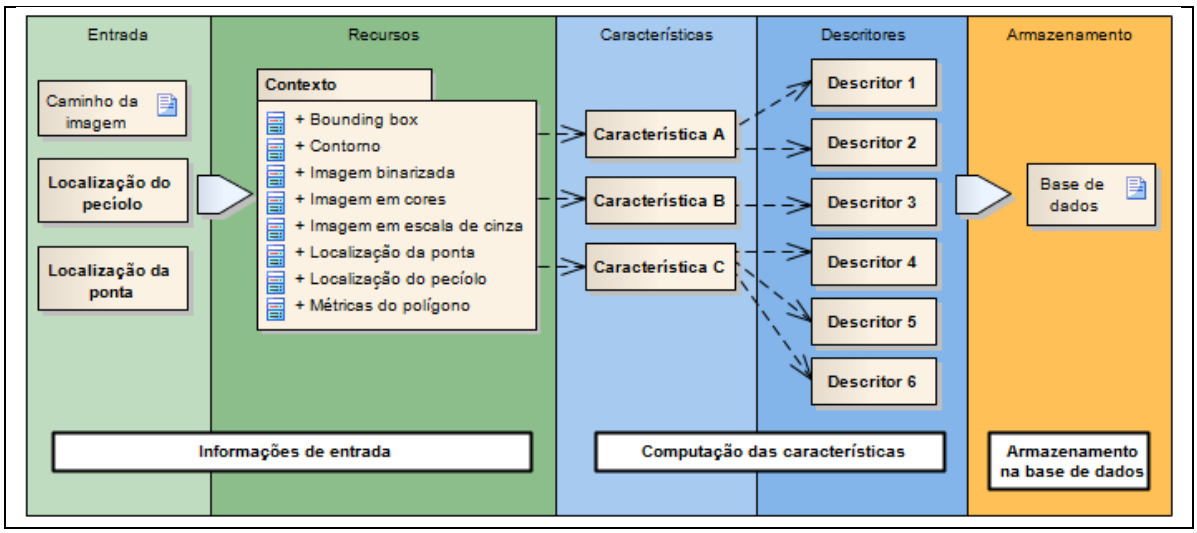


Figura 19 - Fluxo de informações do registro de amostras

Como mostra a Figura 19, as informações de entradas providas pelo usuário são

processadas, passando pelas etapas de segmentação, extração de contorno e cálculo de medidas, disponibilizando estes valores como contexto para o processo de computação das características. Estas, como resultado de sua computação, podem gerar um ou mais valores descritores (que simbolizam quantitativamente cada característica) que são por fim armazenados na base de dados da API juntamente com a espécie informada pelo usuário.

O processo de classificação de uma amostra, conforme ilustrado na Figura 20 é semelhante ao registro de uma amostra na base de dados da API, porém ao invés de cadastrar os descritores computados na base de dados, estes são enviados ao classificador.

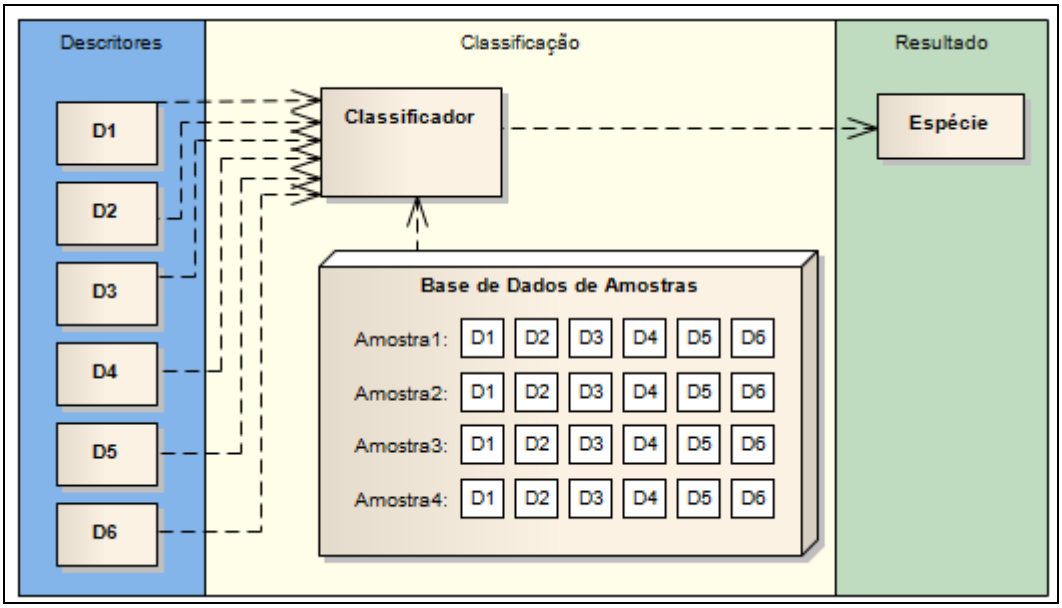


Figura 20 – Fluxo de informações no processo de classificação

A Figura 20 demonstra que o processo de classificação envolve a comparação de um vetor de descritores extraídos a partir da amostra a ser classificada com vetores de descritores de amostras armazenadas na base de dados da API, deduzindo então a espécie da amostra de entrada.

Nas próximas seções são detalhados o funcionamento dos processos de computação dos recursos a partir das informações de entrada, características e classificação das amostras.

3.3.1.1 Computação dos recursos contidos no contexto

As informações de entrada fornecidas pelo usuário (caminho da imagem e coordenadas do pecíolo e da ponta da folha) por si só não são informações suficientes para o processo de extração de características. Esse fato motivou a criação do contexto de extração de características: uma coleção de informações relevantes ao processo de extração de

características.

O contexto de extração de características contém informações básicas, denominadas recursos, computadas a partir dos dados de entrada. Tais recursos são compartilhados pelas características, durante o processo de computação dos valores descritores.

Nas próximas seções são descritas as etapas envolvidas na computação de cada um dos recursos, mencionados anteriormente, utilizados na API.

3.3.1.1.1 Imagem em cores

A partir do nome do arquivo informado pelo usuário, a API utiliza a função `imread` (contida na biblioteca OpenCV) para abrir o arquivo de imagem, obtendo uma matriz com os valores dos canais RGB para cada pixel da imagem.

As imagens utilizadas pela API retratam folhas sobre um plano de fundo, gerando um grande contraste em relação às cores da planta, como é possível observar na Figura 21, uma das amostras utilizadas no projeto.

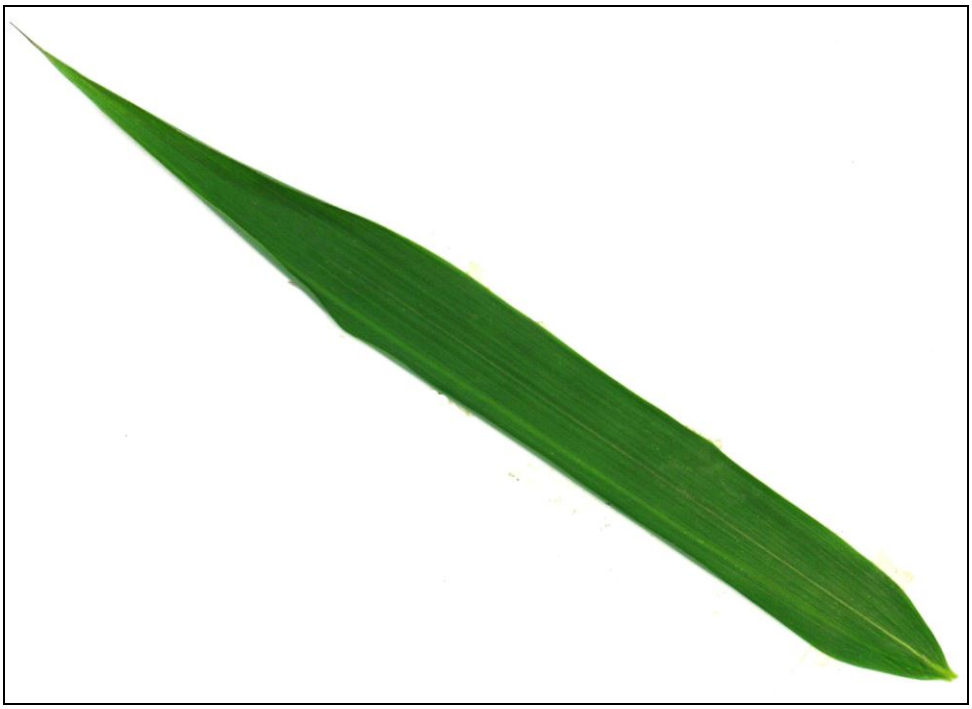


Figura 21 - Amostra utilizada no projeto

3.3.1.1.2 Imagem em tons de cinza

Ao converter a imagem original (em cores) para tons de cinza, cria-se uma representação adequada para os processos e características que trabalham com a intensidade dos pixels da imagem, uma vez que os múltiplos canais de cor são mapeados para apenas um canal nesta nova representação. O Quadro 6 exibe a implementação do método `toGrayscale` (membro da classe utilitária `ImageProcessingUtils`), responsável por efetuar a conversão de imagens em cores para imagens em tons de cinza.

```
01. void ImageProcessingUtils::toGrayscale(cv::Mat &in, cv::Mat &out)
02. {
03.     out.create(in.rows, in.cols, CV_8UC1);
04.
05.     CvMat inputImg = in;
06.     CvMat outputImg = out;
07.
08.     cvCvtColor(&inputImg, &outputImg, CV_BGR2GRAY);
09. }
```

Quadro 6 - Conversão para tons de cinza

Como exposto no Quadro 6, a função `cvCvtColor` (contida na biblioteca OpenCV) é utilizada para a conversão de uma imagem em cores para uma imagem em tons de cinza, a partir do informe da matriz contendo a imagem em cores, uma matriz para armazenar a imagem resultante e a constante `CV_BGR2GRAY`.

A Figura 22 ilustra o resultado do processo de conversão da imagem de amostra exibida na Figura 21.

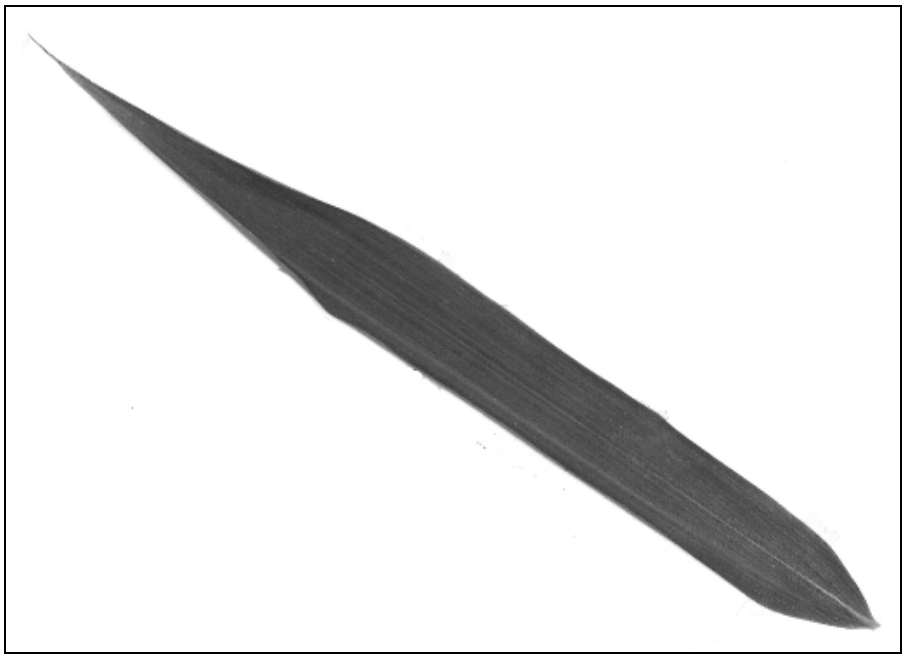


Figura 22 - Amostra em tons de cinza

3.3.1.1.3 Imagem binarizada

A imagem binarizada provém uma matriz que serve como máscara para determinar a área da imagem pertencente à folha, compreendendo o processo de segmentação da imagem da planta. Para a obtenção da imagem binarizada, a imagem em tons de cinza previamente computada na seção 3.3.1.1.2 é submetida à técnica de *thresholding* adaptativo de Otsu. A implementação do método `toBinary` (presente na classe utilitária `ImageProcessingUtils`), responsável pela conversão de uma imagem em escala de cinza para binarizada, é exibida no Quadro 7.

```
01. void ImageProcessingUtils::toBinary(cv::Mat &in, cv::Mat &out)
02. {
03.     cv::Mat strel_disk7 = cv::getStructuringElement(cv::MORPH_ELLIPSE,
                                                        cv::Size(7, 7));
04.     cv::Mat strel_cross3 = cv::getStructuringElement(cv::MORPH_CROSS,
                                                 cv::Size(3,3), cv::Point(1,1));
05.
06.     cv::Mat aux;
07.     cv::Mat aux2;
08.
09.     cv::threshold(in, aux, 0, 255, cv::THRESH_OTSU | cv::THRESH_BINARY_INV);
10.
11.     cv::morphologyEx(aux, aux2, cv::MORPH_CLOSE, strel_disk7);
12.     cv::morphologyEx(aux2, aux, cv::MORPH_OPEN, strel_cross3);
13.     cv::morphologyEx(aux, out, cv::MORPH_CLOSE, strel_disk7);
14. }
```

Quadro 7 - Conversão para imagem binarizada

O método que aplica o *thresholding* na imagem é o `threshold` (presente na biblioteca OpenCV), podendo ser observado a presença de uma *flag* no último parâmetro indicando que deve ser utilizado o método de Otsu.

O método `toBinary` faz, também, uma série de operações morfológicas de abertura e fechamento (a partir da linha 11 no Quadro 7) no intuito de corrigir imperfeições oriundas de ruídos na imagem original.

A Figura 23 exibe o resultado da conversão da imagem exibida na Figura 22 para imagem binarizada (nesta figura as cores estão invertidas, para melhor visualização).

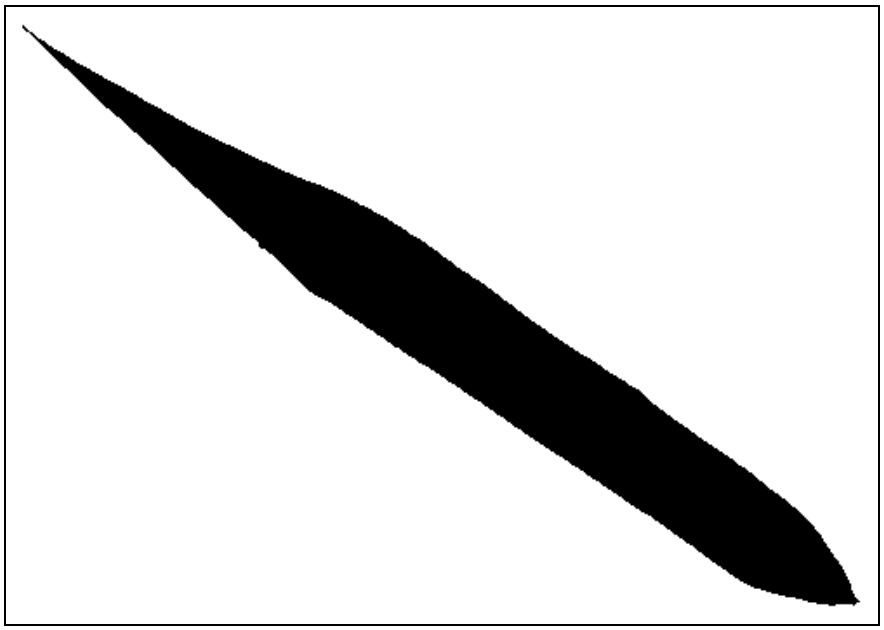


Figura 23 – Imagem binarizada da amostra

3.3.1.1.4 Contorno

Após a aquisição da imagem binarizada da amostra, é possível efetuar a extração do contorno da folha, representando os limites da região da imagem identificada como o órgão foliar. O processo de extração do contorno tem como resultado um *array* de pontos em um plano cartesiano representando a fronteira entre a região correspondente à folha e o fundo da imagem.

O Quadro 8 apresenta a implementação do método `getContour` (membro da classe utilitária `ImageProcessingUtils`), responsável pela extração do contorno a partir de uma imagem binarizada.

```
01. void ImageProcessingUtils::getContour(cv::Mat &in, vector<cv::Point> &out)
02. {
03.     vector<vector<cv::Point>> contours;
04.     cv::findContours(in, contours, CV_RETR_EXTERNAL,
                                                CV_CHAIN_APPROX_TC89_KCOS);
05.
06.     size_t maxIdx = 0;
07.     for (size_t i = 1; i < contours.size(); ++i) {
08.         if (contours[i].size() > contours[maxIdx].size()) {
09.             maxIdx = i;
10.         }
11.     }
12.
13.     out = contours[maxIdx];
14. }
```

Quadro 8 - Extração do contorno

Dentro do método `getContour`, a extração do contorno é feita a partir da chamada do

método `findContours` da biblioteca OpenCV. Este método recebe como parâmetro o `CV_RETR_EXTERNAL` indicando o interesse no contorno externo e o parâmetro `CV_CHAIN_APPROX_TC89_KCOS` indicando o tipo de aproximação a ser utilizado no vetor com os pontos do contorno.

Existe também, dentro do método `getContour`, um tratamento para que na presença de mais de um contorno externo (situação resultante de ruídos na imagem), o que possuir o maior número de pontos seja selecionado como o contorno real da região da folha.

A Figura 24 exibe o resultado do desenho formado pelos pontos resultantes do processo de extração do contorno da Figura 23.

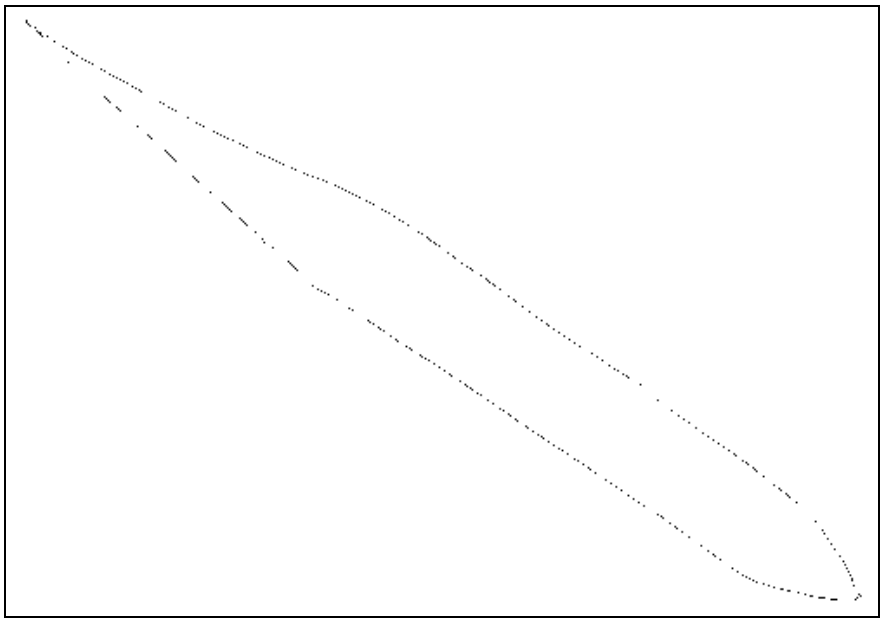


Figura 24 - Contorno extraído da amostra

3.3.1.1.5 *Bounding Box*

Nas imagens de amostras, a *bounding box* correspondendo à região da folha é calculada em cima do *array* de pontos que compõem o contorno do órgão foliar. Ela é normalmente utilizada nos processos de extração de características para a criação de uma *Region of Interest* (ROI) dos recursos imagens utilizados.

No Quadro 9 é apresentada a implementação do método `getBoundingBox` (pertencente à classe `FeatureExtractionContext`) que invoca o cálculo da *bounding box* a partir do contorno da folha.

```
01. const cv::Rect& FeatureExtractionContext::getBoundingBox() {
02.     if (_bBox.height == 0 && _bBox.width == 0) {
03.         _bBox = cv::boundingRect(this->getContour());
04.     }
05.     return _bBox;
06. }
```

Quadro 9 - Obtenção da bounding box

O método `getBoundingBox` faz uso da função `boundingRect`, presente na biblioteca OpenCV, retornando um retângulo (`Rect`) contendo as dimensões da *bounding box*. A imagem apresentada na Figura 25 é o resultado do desenho deste retângulo (em vermelho) por cima da imagem do contorno da folha, exibida na Figura 24.

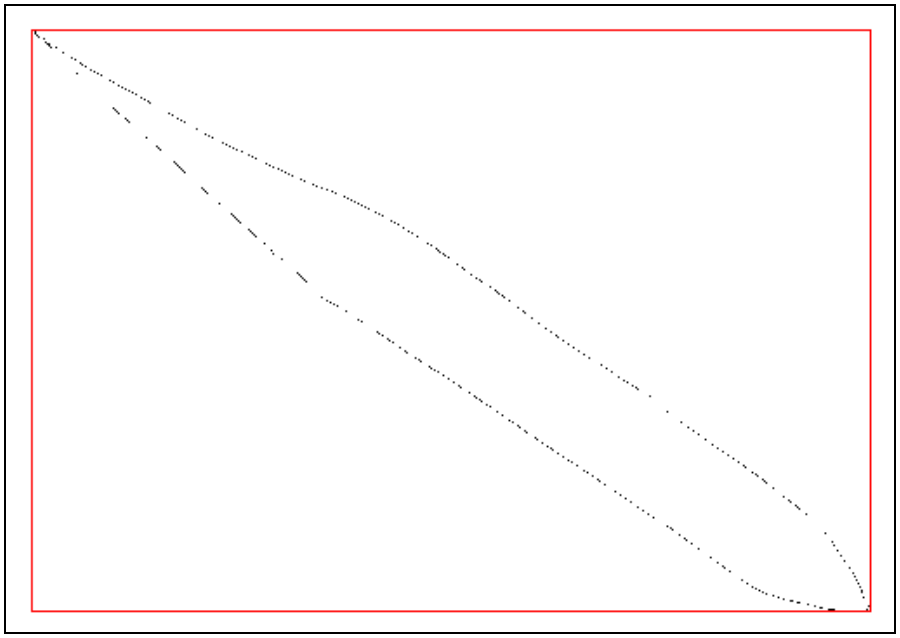


Figura 25 - Bounding box da amostra

3.3.1.1.6 Métricas do polígono

A partir do contorno extraído da imagem de amostra, o *array* de pontos passa a ser considerado um polígono. Ao qual, será utilizado para efetuar o cálculo de algumas medidas geométricas. Estas medidas, representadas pela estrutura `PolygonMetrics`, referem-se à área, perímetro (comprimento do contorno) e centróide da forma geométrica. No Quadro 10 é apresentada a implementação do método `getPolygonMetrics` (parte da classe utilitária `GeometryUtils`), responsável pelo cálculo destas medidas.

```
01. bool GeometryUtils::getPolygonMetrics(const vector<cv::Point>& vertexArr,
    PolygonMetrics& out_metrics)
02. {
03.     typedef vector<cv::Point>::const_iterator IteType;
04.
05.     if (vertexArr.size() < 3) return false;
06.     memset(&out_metrics, 0, sizeof(PolygonMetrics));
07.
08.     IteType prev = vertexArr.begin() + (vertexArr.size() - 1);
09.     IteType curr = vertexArr.begin();
10.
11.     for (; curr != vertexArr.end(); prev = curr++)
12.     {
13.        out_metrics.perimeter += GeometryUtils::euclidianDistance(*prev,*curr);
14.
15.        out_metrics.area += (curr->x + prev->x) * (curr->y - prev->y);
16.
17.        out_metrics.xc += curr->x;
18.        out_metrics.yc += curr->y;
19.     }
20.
21.     out_metrics.area = abs(out_metrics.area / 2);
22.     out_metrics.xc /= vertexArr.size();
23.     out_metrics.yc /= vertexArr.size();
24.
25.     return true;
26. }
```

Quadro 10 - Cálculo de métricas de polígono

3.3.1.2 Características

A API de reconhecimento de plantas disponibiliza as seguintes implementações de características: circularidade (classe `Roundness`), dispersão (classe `Dispersion`), magreza (classe `Slimness`), descritores de Fourier (classe `FourierDescriptors`), máscara de nervura (classe `VeinMask`), lacunaridade (classe `Lacunarity`) e estatísticas de cor (classe `ColorMoments`).

As próximas seções destinam-se ao detalhamento da implementação das características mencionadas.

3.3.1.2.1 Circularidade

A circularidade indica o quanto uma forma geométrica assemelha-se à um círculo perfeito. Em seu cálculo, esta característica trabalha apenas com as informações de métricas do polígono. A fórmula utilizada para o cálculo desta característica é exibida no Quadro 11.

$$roundness = \frac{4\pi A}{P^2}$$

Fonte: Kadir et al. (2011).

Quadro 11 - Fórmula para cálculo da circularidade

Na fórmula citada, *A* indica a área total da folha e *P* o perímetro da mesma.

O Quadro 12 exibe a implementação do método `compute` na classe `Roundness`, onde ocorre a computação dos valores descritores da circularidade.

```
01.  void Roundness::compute(FeatureExtractionContext& context, std::list<double>
     &out_data) const
02.  {
03.    double roundness = GeometryUtils::getRoundness(*context.getPolygonMetrics());
04.    out_data.push_back(roundness);
05.  }
```

Quadro 12 - Computação da circularidade

Neste caso, o método `compute` apenas invoca o método `getRoundness` (membro da classe utilitária `GeometryUtils`) passando como parâmetro as métricas do polígono armazenadas no contexto de extração recebido e adiciona o valor de retorno (descritor) na lista de saída `out_data`.

A implementação do método `getRoundness` é exibida no Quadro 13.

```
01.  double GeometryUtils::getRoundness(const PolygonMetrics& metrics) {
02.    return (4 * M_PI * metrics.area) / (metrics.perimeter * metrics.perimeter);
03.  }
```

Quadro 13 - Utilitário para cálculo de circularidade

Pode-se observar a distinção, no quesito de circularidade, entre as amostras ilustradas na Figura 26 a partir da diferença entre os coeficientes computados para esta característica.

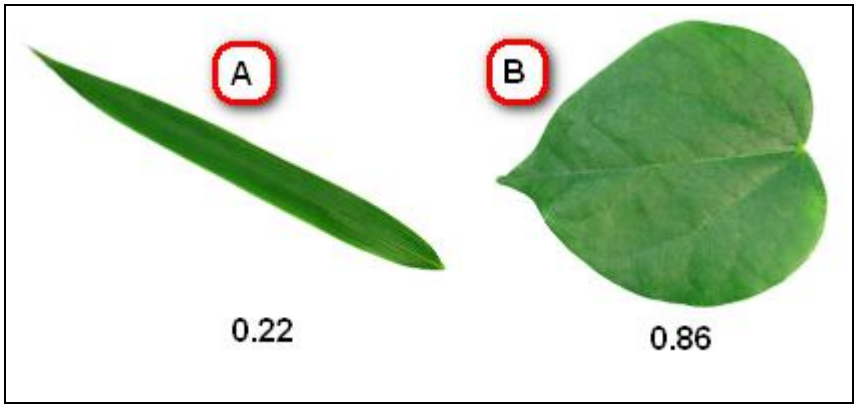


Figura 26 - Distinção por circularidade

Em relação à Figura 26, pode-se comprovar visualmente que o formato da amostra B possui maior similaridade com um círculo do que a amostra A.

3.3.1.2.2 Dispersão

A dispersão, neste projeto, representa a razão entre a maior e menor distância de um ponto localizado no contorno da folha até o centro do polígono. A fórmula utilizada para o cálculo desta característica é apresentada no Quadro 14.

$$dispersion = \frac{\max(\sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2})}{\min(\sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2})}$$

Fonte: Kadir et al. (2011).

Quadro 14 - Fórmula para o cálculo da dispersão

Onde $(\bar{x}, \bar{y})$ denota a centróide do polígono formado pela região da folha e (x_i, y_i) representa um ponto contido no contorno do mesmo.

A dispersão utiliza-se do *array* de pontos do contorno da folha e das métricas do polígono, ambos armazenados no contexto recebido pelo método `compute` (implementado na classe `Dispersion`). Tal método é apresentado no Quadro 15.

```
01.  void Dispersion::compute(FeatureExtractionContext& context,
     std::list<double> &out_data) const
02.  {
03.    double dispersion = GeometryUtils::getDispersion(context.getContour(),
                                        *context.getPolygonMetrics());
04.    out_data.push_back(dispersion);
05.  }
```

Quadro 15 - Computação da dispersão

O método `compute` apresentado realiza apenas a chamada do método utilitário `getDispersion` (presente na classe `GeometryUtils`) fornecendo como argumentos o contorno e as métricas do polígono recebidos. O valor da dispersão obtido (linha 3 do Quadro 15) é então armazenado no parâmetro de saída `out_data`. A implementação do método `getDispersion` é exibida no Quadro 16.

```
01. double GeometryUtils::getDispersion(const vector<cv::Point>& vertexArr,
    const PolygonMetrics& metrics)
02. {
03.     cv::Point mid((int)GeometryUtils::round(metrics.xc),
                              (int)GeometryUtils::round(metrics.yc));
04.
05.     double min = GeometryUtils::euclidianDistance(vertexArr[0], mid);
06.     double max = min;
07.
08.     for (size_t i = 1; i < vertexArr.size(); ++i) {
09.         double dist = GeometryUtils::euclidianDistance(vertexArr[i], mid);
10.
11.         if (dist < min) { min = dist; }
12.         else if(dist > max) { max = dist; }
13.     }
14.
15.     return max / min;
16. }
```

Quadro 16 - Utilitário para cálculo de dispersão

A discriminação das amostras utilizando a dispersão é ilustrada na Figura 26, onde a diferença entre os valores obtidos do coeficiente indicam a diferença entre as amostras.

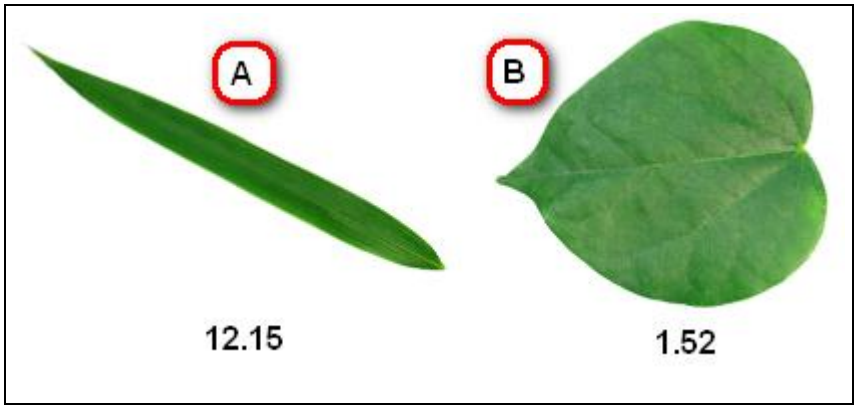


Figura 27 – Distinção por dispersão

Pode ser comprovado visualmente, a partir da Figura 27, que a amostra A possui uma maior diferença entre as distâncias (em relação ao centro da folha) dos pontos localizados em seu contorno.

3.3.1.2.3 Magreza

Na API de reconhecimento de plantas, a definição de magreza é a razão entre o comprimento e a largura da folha, orientando-se pelas coordenadas do pecíolo e da ponta da folha informadas pelo usuário. A fórmula utilizada para o cálculo da magreza é apresentada no Quadro 17.

$$slimness = \frac{l_1}{l_2}$$

Fonte: Kadir et al. (2011).

Quadro 17 - Fórmula para cálculo da magreza

Na fórma citada, l_1 denota a medida do comprimento (em pixels) da folha e l_2 a largura da mesma.

O processo de computação da magreza, para obtenção dos valores a serem utilizados na razão mencionada acima, é subdividido em 3 (três) etapas: cálculo do comprimento, alinhamento da imagem com eixo x e cálculo da largura. A Figura 28 ilustra o resultado das operações, realizadas pelas etapas mencionadas, em cima do contorno da amostra.

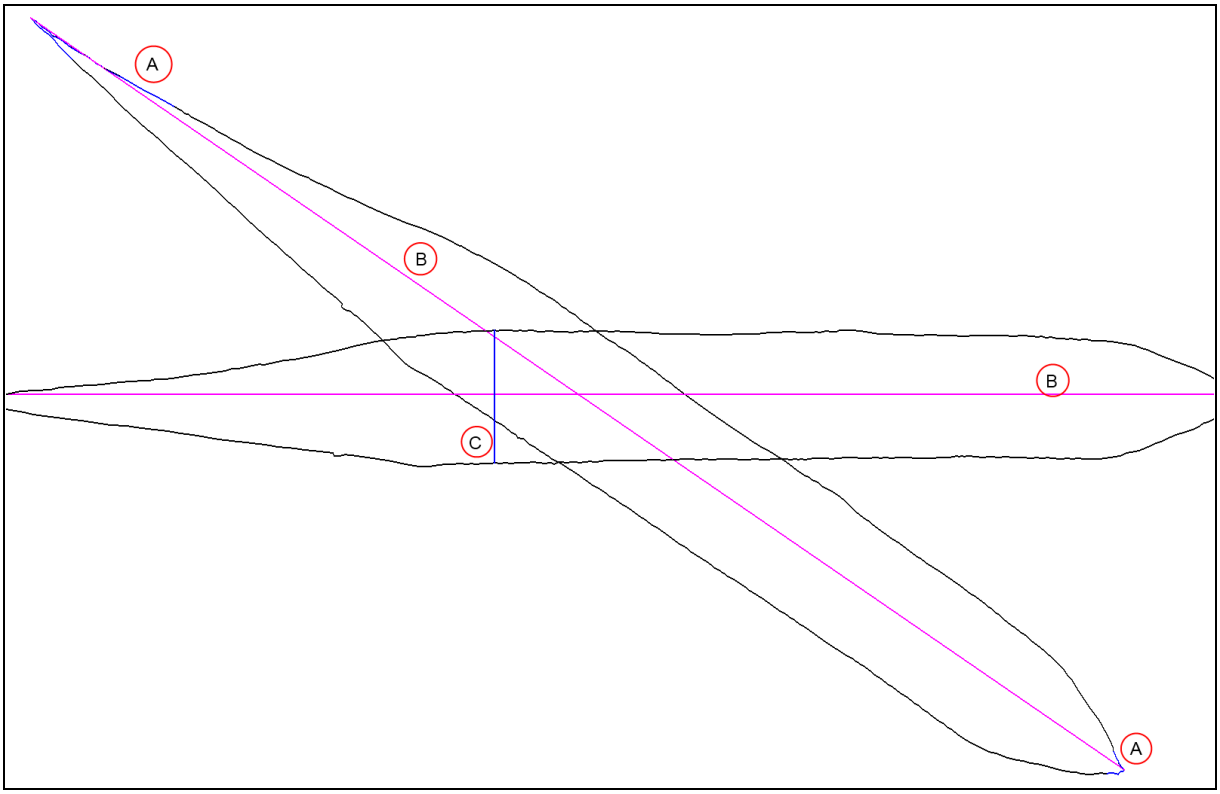


Figura 28 - Etapas do cálculo da magreza da amostra

Primeiramente, utilizando os pontos correspondentes ao pecíolo e ponta da folha (informados pelo usuário) é calculada uma margem de erro em cima da precisão dos dados informados. Utilizando tal margem de erro, os pontos do contorno próximos aos pontos informados pelo usuário são elencados em uma lista de candidatos para par de pontos mais distantes (pontos azuis no item A da Figura 28).

Em cima da lista de pontos candidatos é aplicado o algoritmo de *convex hull* (Quadro 18, linha 44) criando um novo polígono com apenas os elementos mais exteriores. Este novo polígono gerado pelo *convex hull* é passado como argumento para a função utilitária

`getFartestPoints`, que retorna o par de pontos mais distante do polígono (Quadro 18, linha 47). Este par de pontos mais distantes representa, então, o comprimento da folha (item B na Figura 28).

O Quadro 18 exibe a implementação correspondente ao cálculo do comprimento da folha dentro do método `compute` da classe `Slimness`.

```
01.  void Slimness::compute(FeatureExtractionContext& context, std::list<double>
     &out_data) const
02.  {
03.      //calcula tolerância para calibrar os pontos informados pelo usuário
04.      double cosThres =
     cos(PLNT_DEG_TO_RAD(context.getParamValue(PLNT_PRMID_SLIMNESS_ERR_DEG).val));
05.      //pega os pontos informados pelo usuário e transforma em vetores
06.      cv::Vec2i p1 = (cv::Vec2i)context.getPetiolePos();
07.      cv::Vec2i p2 = (cv::Vec2i)context.getTipPos();
08.
09.      //calcula o meio da reta informada pelo usuário
10.      cv::Vec2i mid = p1 + (0.5 * (p2 - p1));
11.      //calcula vetor mid->p2
12.      cv::Vec2i v_mid_p2 = p2 - mid;
13.      double v_mid_p2_len = sqrt(v_mid_p2.ddot(v_mid_p2));
14.
15.      //pontos do contorno
16.      const vector<cv::Point>& cont = context.getContour();
17.      cv::Vec2i aux;
18.      //candidados a nova reta corrigida
19.      vector<cv::Point> candidates;
20.
21.      //procura pontos que são colineares aos vetores mid e p2, considerando a
22.      //tolerância
23.      size_t prev = cont.size() - 1;
24.      for (size_t cur = 0; cur < cont.size(); prev = cur++) {
25.          //cria line iterator e pula o primeiro elmento (como vai ser
26.          //circular, para não repetir o ponto)
27.          cv::LineIterator lit(context.getBinaryImg(), cont[prev], cont[cur]);
28.          ++lit;
29.          //itera pontos decompostos pelo line iterator
30.          for (int i = 1; i < lit.count; ++i, ++lit) {
31.              //calcula vetor que vai de mid até a posição (mid->pos)
32.              aux = ((cv::Vec2i)lit.pos()) - mid;
33.              //calcula o cosseno do angulo com mid->p2
34.              double pCos = v_mid_p2.ddot(aux) / (v_mid_p2_len *
                                                    sqrt(aux.ddot(aux)));
35.              //verifica se atende à tolerância, adicionando nos pontos
36.              //candidados
37.              if (abs(pCos) >= cosThres) {
38.                  candidates.push_back(lit.pos());
39.              }
40.          }
41.      }
42.      //faz hull nos pontos candidatos
43.      vector<cv::Point> lenHull;
44.      cv::convexHull(candidates, lenHull);
45.      //calcula maior distância entre os pontos do hull
46.      int fp1, fp2;
47.      plantarum::GeometryUtils::getFartestPoints(lenHull, fp1, fp2);
48.
49.      //ROTAÇAO DO POLIGONO
50.      //CÁLCULO DA LARGURA
51.  }
```

Quadro 18 - Cálculo do comprimento da amostra

Após a execução do trecho de código apresentado no Quadro 18, os pontos contidos

nos índices `fp1` e `fp2` do *array* `lenHull` tornam-se os pontos que representam o comprimento da folha.

O próximo passo do algoritmo é rotacionar o polígono que representa a folha, de modo que a reta que liga os pontos do comprimento do polígono fique alinhada com o eixo X (Quadro 19, linhas 18 à 50), como é demonstrado na Figura 28 através da imagem da folha na posição horizontal. A implementação deste trecho do algoritmo é exibido no Quadro 19.

```
01. void Slimness::compute(FeatureExtractionContext& context, std::list<double>
    &out_data) const
02. {
03.     //CÁLCULO DO COMPRIMENTO
04.
05.     vector<cv::Point> rotated;
06.     cv::Vec2i farP1;
07.     cv::Vec2i farP2;
08.
09.     //verifica se os pontos mais distantes já estão alinhados ao eixo X, não
10.     //precisando rotacionar os pontos
11.     if (lenHull[fp1].y == lenHull[fp2].y) {
12.         rotated = cont;
13.         farP1 = lenHull[fp1];
14.         farP2 = lenHull[fp2];
15.     }
16.     //vamos rotacionar os pontos para que a linha do comprimento fique
17.     //colinear ao eixo x
18.     else {
19.         //cria vetor no mesmo x que v2 e mesmo y que v1 (formando um
20.         //triângulo com com v1 e v2)
21.         aux[0] = lenHull[fp2].x;
22.         aux[1] = lenHull[fp1].y;
23.
24.         //descobre angulo deste triângulo (diferença entre entre uma linha
25.         //horizontal e a atual mais longa)
26.         double theta = plantarum::GeometryUtils::vAngle(
                    (cv::Vec2i)lenHull[fp1], aux, (cv::Vec2i)lenHull[fp2]);
27.         //calcula vetor que aponta para o meio da reta mais longa (este será
28.         //o centro da rotação)
29.         cv::Vec2i axMid = (cv::Vec2i)lenHull[fp1] +
                  (0.5 * ((cv::Vec2i)lenHull[fp2] - (cv::Vec2i)lenHull[fp1]));
30.
31.         //constrói matriz de rotação 2D utilizando o meio calculado e a
32.         //diferença
33.         cv::Mat myMatrix =
34.         cv::getRotationMatrix2D(cv::Point2f((float)axMid[0], (float)axMid[1]),
                              PLNT_RAD_TO_DEG(theta), 1);
35.
36.         //cria lista contendo os pontos identificados como mais distantes
37.         vector<cv::Vec2i> srcList(2);
38.         srcList[0] = (cv::Vec2i)lenHull[fp1];
39.         srcList[1] = (cv::Vec2i)lenHull[fp2];
40.         //transforma estes pontos pela matriz calculada
41.         vector<cv::Vec2i> dstList(2);
42.         cv::transform(srcList, dstList, myMatrix);
43.         //seta pontos distantes agora transformados
44.         farP1 = dstList[0];
45.         farP2 = dstList[1];
46.
47.         //transforma todos os pontos do contorno também
48.         rotated.resize(cont.size());
49.         cv::transform(cont, rotated, myMatrix);
50.     }
51.
52.     //CÁLCULO DA LARGURA
53. }
```

Quadro 19 - Rotação para alinhamento do comprimento com eixo X

Após a execução do trecho de código apresentado no Quadro 19, o *array* `rotated` contém os pontos que compõem o contorno do polígono com a reta do comprimento alinhada com o eixo X e os pontos representados pelos pontos `farP1` e `farP2` os extremos de tal reta.

O valor restante indicando a largura da folha (ilustrado na Figura 28 como o item C) é calculado buscando a reta com a maior extensão ortogonal à linha do comprimento do

polígono (Quadro 20, linhas 30 à 48). O Quadro 20 exibe a implementação deste passo.

```
01. void Slimness::compute(FeatureExtractionContext& context, std::list<double>
    &out_data) const
02. {
03.     //CÁLCULO DO COMPRIMENTO
04.     //ROTAÇÃO DO POLÍGONO
05.
06.     //struct utilizada para quardar o máximo de mínimo Y para uma coordenada X
07.     typedef struct tagMinMax {
08.         int min_y;
09.         int max_y;
10.     } MinMax;
11.
12.     //monta mapa de ortogonais e seus pontos extremos, para cada valor em X
13.     map<int, MinMax> ortog;
14.     for (vector<cv::Point>::const_iterator it = rotated.begin();
                                                 it != rotated.end(); ++it)
15.     {
16.         map<int, MinMax>::iterator item = ortog.find(it->x);
17.         if (item == ortog.end()) {
18.             MinMax& newItem = ortog[it->x];
19.             newItem.min_y = it->y;
20.             newItem.max_y = it->y;
21.         }
22.         else if (item->second.min_y > it->y) {
23.             item->second.min_y = it->y;
24.         }
25.         else if (item->second.max_y < it->y) {
26.             item->second.max_y = it->y;
27.         }
28.     }
29.     //percorre o mapa, procurando qual a maior distância
30.     if (ortog.size() > 0) {
31.         //pega iterador para mapa e já salva o primeiro como o mais distante
32.         map<int, MinMax>::const_iterator it = ortog.begin();
33.         map<int, MinMax>::const_iterator fartest = it++;
34.         int maxDist = fartest->second.max_y - fartest->second.min_y;
35.
36.         //percorre o restante do mapa, verificando se tem maior distância
37.         //para algum outro x
38.         for (; it != ortog.end(); ++it) {
39.             int currDist = it->second.max_y - it->second.min_y;
40.             if (maxDist < currDist) {
41.                 maxDist = currDist;
42.                 fartest = it;
43.             }
44.         }
45.
46.         //calcula a slimness (razão entre o comprimento e a largura)
47.         slimness = plantarum::GeometryUtils::vLen(farP1 - farP2) /
                                                        (double)maxDist;
48.     }
49.
50.     //adiciona o valor do cálculo na saída
51.     out_data.push_back(slimness);
52. }
```

Quadro 20 - Cálculo da largura e magreza do polígono

Conforme apresentado no Quadro 20, o valor calculado representando a largura da amostra é armazenado na variável local `maxDist`. Por fim, este valor juntamente com os pontos que indicam o comprimento da amostra (`farP1` e `farP2`) são utilizados para o cálculo da magreza da folha.

Na Figura 29 pode-se observar a dintinção entre diferentes amostras, a partir da análise do valor computado da magreza.

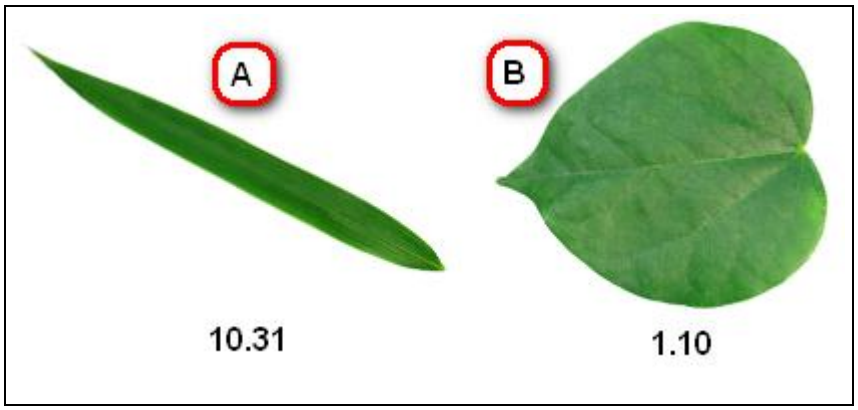


Figura 29 - Distinção por magreza

A partir da Figura 29 é possível concluir visualmente que a amostra A possui uma aparência mais magra do que a amostra B.

3.3.1.2.4 Descritores de Fourier

Os descritores de Fourier, na API de reconhecimento de plantas, são coeficientes resultantes da DFT em cima de um sinal unidimensional construído a partir da distância entre os pontos do contorno da amostra e a sua centróide. A classe responsável pela implementação dos descritores de Fourier é a `FourierDescriptors`.

A implementação da extração dos descritores é realizada em 2 etapas: primeiramente, através do método `sampleContour`, é computado um vetor com amostras das distâncias do centro da amostra até os pontos do contorno. Em seguida, este vetor (equivalendo a um sinal unidimensional) serve como entrada para a transformada de Fourier, implementada dentro do método `compute`.

O Quadro 21 demonstra detalhes da implementação do método `sampleContour`.

```
01. /**Faz sampling do contorno da imagem, calculando a distância para sua
    centróide*/
02. void FourierDescriptors::sampleContour(FeatureExtractionContext& context,
    unsigned long resolution, vector<double> &out_samples) const
03. {
04.     //pega contorno da imagem
05.     const vector<cv::Point> &contour = context.getContour();
06.     //pega métricas do polígono e então calcula o raio da imagem
07.     const PolygonMetricsPtr metrics = context.getPolygonMetrics();
08.     double imgRadius = GeometryUtils::getRadius(contour, *metrics);
09.     //cria vetor simbolizando ponto médio
10.     cv::Vec2i midPoint((int)GeometryUtils::round(metrics->xc),
                                  (int)GeometryUtils::round(metrics->yc));
11.
12.     //variáveis de controle do número do ponto sendo iterado
13.     int currStep = 0;
14.     double inc = metrics->perimeter / (double)resolution;
15.     double nextSampleStep = 1;
16.
17.     //vetor auxiliar para cálculos
18.     cv::Vec2i aux;
19.     //itera de forma circular o contorno
20.     size_t prev = contour.size() - 1;
21.     for (size_t cur = 0; cur < contour.size(); prev = cur++) {
22.         //inicializa um line iterator e pula o primeiro ponto (pois iria
23.         //repetir, já que estamos iterando tudo)
24.         cv::LineIterator lit(context.getBinaryImg(), contour[prev],
                                                         contour[cur]);
25.         ++lit;
26.         //itera o line iterator
27.         for (int i = 1; i < lit.count; ++i, ++lit) {
28.             //incrementa step atual
29.             ++currStep;
30.             //verifica se vai selecionar este ponto como sample
31.             while (plantarum::GeometryUtils::round(nextSampleStep) ==
                                                    (double)currStep) {
32.                 //calcula diferença entre os vetores e então pega a distância
33.                 //entre eles [magnitude do vetor diferencial]
34.                 aux = ((cv::Vec2i)lit.pos()) - midPoint;
35.                 double distance = plantarum::GeometryUtils::vLen(aux);
36.
37.                 //adiciona no array de samples (normaliza pelo raio)
38.                 out_samples.push_back(distance / imgRadius);
39.                 //incrementa o next sample
40.                 nextSampleStep += inc;
41.             }
42.         }
43.     }
44. }
```

Quadro 21 - Amostragem do contorno da imagem

O método `sampleContour`, exibido no Quadro 21, percorre o contorno coletando o número de amostras informado pelo parâmetro `resolution` (a partir da linha 21). Estas amostras têm suas distâncias para a centróide da imagem calculadas e armazenadas no parâmetro de saída `out_samples`.

Após a obtenção das amostras do contorno, elas são submetidas à transformada de Fourier para obtenção dos coeficientes utilizados para aproximar o sinal formado pelo vetor dessas amostras (Quadro 22, linhas 20 à 34). O Quadro 22 exibe a implementação desta etapa.

```
01. /**Computa a feature, em cima do contexto, e joga em out_data*/
02. void FourierDescriptors::compute(FeatureExtractionContext& context,
03.                                  std::list<double> &out_data) const
04. {
05.     //pega parâmetros
06.     unsigned long descQuantity = (unsigned long)context.getParamValue(
                                         PLNT_PRMID_FOURIER_QUANTITY).val;
07.     unsigned long contourRes = (unsigned long)context.getParamValue(
                                         PLNT_PRMID_FOURIER_RESOLUTION).val;
08.
09.     if (descQuantity < 1 || contourRes < 1) return;
10.
11.     //faz sampling do contorno da imagem
12.     vector<double> samples;
13.     this->sampleContour(context, contourRes, samples);
14.
15.     //incrementa +1 na quantidade de coeficientes, pq o DC deve ser calculado
16.     //também
17.     ++descQuantity;
18.
19.     //vetor de coeficientes fourier
20.     vector<cv::Vec2d> coefs;
21.     //para cada coeficiente
22.     for (unsigned long fi = 0; fi < descQuantity; ++fi) {
23.         cv::Vec2d coef(0.0, 0.0);
24.         //percorre vetor de samples
25.         for (size_t i = 0; i < samples.size(); ++i) {
26.             double theta = TWO_PI * fi * (i / (double)samples.size());
27.             double val = samples[i];
28.
29.             coef[0] += val * cos(theta);
30.             coef[1] += val * -sin(theta);
31.         }
32.         //adiciona na lista de coeficientes calculados
33.         coefs.push_back(coef);
34.     }
35.     //percorre coeficientes, ignorando o primeiro (DC) e calculando apenas as
36.     //magnitudes
37.     for (size_t i = 1; i < coefs.size(); ++i) {
38.         cv::Vec2d& theVal = coefs[i];
39.         //calcula magnitude e adiciona no output
40.         double magnitude = sqrt(theVal.ddot(theVal));
41.         out_data.push_back(magnitude);
42.     }
43. }
```

Quadro 22 - Cálculo coeficientes de Fourier

O método `compute`, apresentado no Quadro 22, primeiramente realiza a amostragem dos pontos pertencentes ao contorno da folha a partir da chamada do método `sampleContour` (linha 13). Em cima desta amostragem, que caracteriza um sinal unidimensional, o método aplica a DFT (implementada da linha 22 à 34) e computa a quantidade de coeficientes parametrizada (variável `descQuantity`, na linha 6).

Apenas as magnitudes dos coeficientes são mantidas (como pode ser observado na linha 40 do Quadro 22), tornando os descritores invariáveis em função da rotação de imagens.

3.3.1.2.5 Máscara de nervura

A máscara de nervura é uma característica que expressa uma medida quantitativa da área total que as nervuras presentes na folha ocupam, em relação à área total da amostra. O processo de cálculo desta característica, implementada pela classe `VeinMask`, resulta em imagens como a apresentada na Figura 30 (na figura as cores estão invertidas e está aplicado um realce para melhor vizualização). Optou-se por substituir a imagem utilizada nas ilustrações das extrações de características por uma que apresenta uma nervação mais evidente.

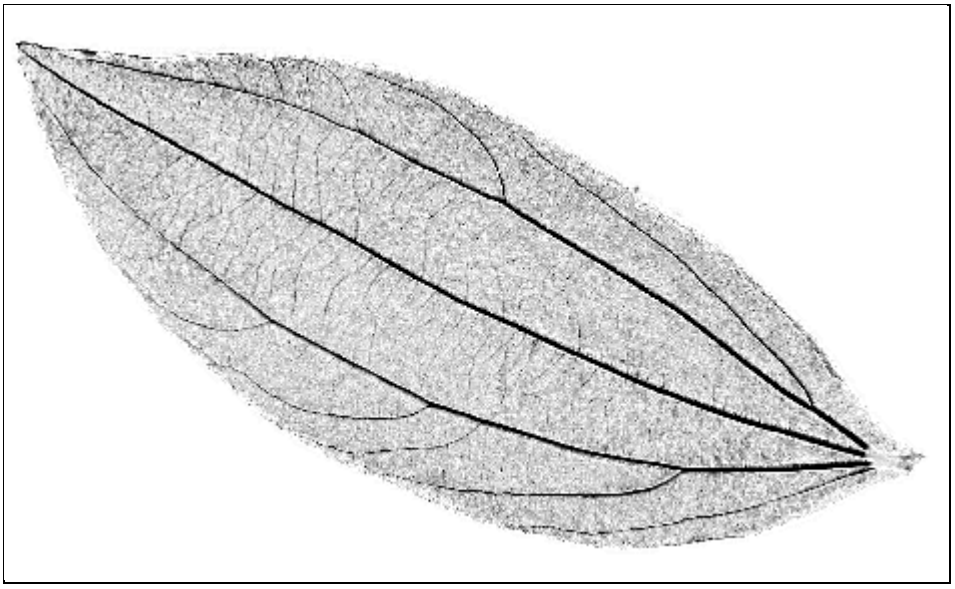


Figura 30 - Máscara de nervura da amostra

A área ocupada pelas nervuras da folha é obtida através da operação *top-hat* que (Quadro 23, linha 22), por definição, é a diferença entre a imagem original e sua abertura. No Quadro 23 é apresentada a implementação do método `compute`, membro da classe `VeinMask`, que efetua a computação dos valores dessa característica.

```
01. /**Computa a feature, em cima do contexto, e joga em out_data*/
02. void VeinMask::compute(FeatureExtractionContext& context, std::list<double>
    &out_data) const
03. {
04.     //pega número de máscaras a serem utilizadas e cria matriz de output de
05.     //transformações
06.     size_t maskNum =
                    (size_t)context.getParamValue(PLNT_PRMID_VEIN_MASK_COUNT).val;
07.     cv::Mat output;
08.
09.     //calcula quantos pixels fazem parte da imagem binária
10.     double pCount = cv::countNonZero(context.getBinaryImg());
11.
12.     //aplica o tanto de máscaras desejadas de tophat
13.     for (size_t i = 0; i < maskNum; ++i)
14.     {
15.         //calcula diâmetro do círculo utilizado como elemento estruturante.
16.         //Iniciamos com RAIO = 1 e então incrementando de 1 em 1. Isso
17.         //significa que o DIAMETRO do primeiro elemento é 3, o prox 5,7,9...
18.         int diameter = 3 + (i * 2);
19.         cv::Mat strel = cv::getStructuringElement(cv::MORPH_ELLIPSE,
                                                cv::Size(diameter, diameter));
20.
21.         //faz tophat e conta a quantidade de pixels
22.         cv::morphologyEx(context.getGrayImg(),output,cv::MORPH_TOPHAT, strel);
23.         int nonZero = cv::countNonZero(output);
24.
25.         //feature é a razão entre os pixels não zero totais e da máscara
26.         out_data.push_back(nonZero / pCount);
27.     }
28. }
```

Quadro 23 - Cálculo da máscara de nervura

A implementação apresentada no Quadro 23 efetua o *top-hat* utilizando o número de máscaras a serem calculadas parametrizado na base de dados, sendo que em cada uma o elemento estruturante utilizado para as operações morfológicas sofre alteração no tamanho.

3.3.1.2.6 Lacunaridade

A API de reconhecimento de plantas utiliza a lacunaridade como uma característica baseada na textura na planta. A lacunaridade, implementada pela classe `Lacunarity`, é uma medida aplicada à fractais que indica se o padrão apresentado pela fractal possui poucas ou muitas lacunas (espaços onde o padrão não se manifesta). A fórmula utilizada para o cálculo da lacunaridade é apresentada no Quadro 24.

$$L_p = \left(\frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} \left(\frac{P_{mn}}{\frac{1}{MN} \sum_{k=1}^{M} \sum_{l=1}^{N} P_{kl}} - 1 \right)^p \right)^{1/p}$$

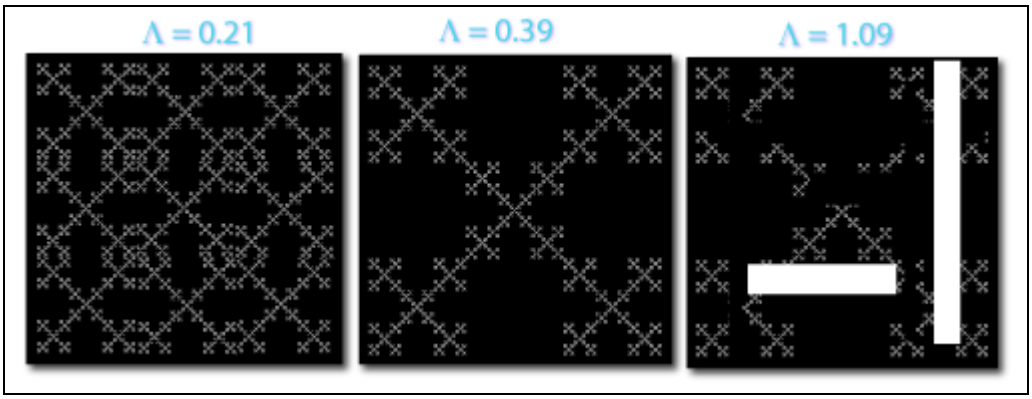
Fonte: Kadir et al. (2011).

Quadro 24 - Fórmula para cálculo da lacunaridade

Na fórmula citada, *M* denota o número de colunas da imagem, *N* simboliza o número de linhas e P_{mn} o valor do pixel na coordenada (m, n) da imagem. O elemento *p* é o coeficiente da lacunaridade sendo calculado.

A Fonte: adaptado de Karperien (2010).

Figura 31 ilustra como o aumento das imperfeições em um fractal faz com que o valor da lacunaridade aumente.



Fonte: adaptado de Karperien (2010).

Figura 31 - Relação entre a lacunaridade e homogeneidade de um fractal

Nos quadros a seguir, são apresentados os trechos da implementação do método `compute`, na classe `Lacunarity`, responsável pelo cálculo da lacunaridade. Primeiramente, descrito no Quadro 25, é executado o processo de computação dos coeficientes a serem utilizados no cálculo da lacunaridade, assim como as *ROI* das imagens a serem trabalhadas.

```
01. /**Computa a feature, em cima do contexto, e joga em out_data*/
02. void Lacunarity::compute(FeatureExtractionContext& context,
    std::list<double> &out_data) const
03. {
04.     //pega quantidade de coeficientes e verifica se é maior que zero
05.     unsigned long coefCount = (unsigned long)context.getParamValue(
                                                    PLNT_PRMID_LACUNARITY_COUNT).val;
06.     if (coefCount < 1) return;
07.     //pega o primeiro dos coeficientes e o delta dele
08.     unsigned long coefStart = (unsigned long)context.getParamValue(
                                            PLNT_PRMID_LACUNARITY_FIRST_COEF).val;
09.     unsigned long coefDelta = (unsigned long)context.getParamValue(
                                            PLNT_PRMID_LACUNARITY_COEF_DELTA).val;
10.
11.     //cria vetor com os coeficientes calculados
12.     vector<unsigned long> coeficients(coefCount);
13.     for (unsigned long i = 0; i < coefCount; ++i) {
14.         coeficients[i] = coefStart + (i * coefDelta);
15.     }
16.     //cria vetor com os dados computados para cada coeficiente
17.     vector<ColorInfo> coefData(coefCount);
18.
19.     //pega bounding box
20.     const cv::Rect& bbox = context.getBoundingBox();
21.
22.     //cria ROI da colorida, grayscale e binária
23.     cv::Mat colorROI = context.getRawImg()(bbox);
24.     cv::Mat grayROI = context.getGrayImg()(bbox);
25.     cv::Mat binROI = context.getBinaryImg()(bbox);
26.
27.     //CÁLCULO DA MÉDIA DOS VALORES DOS CANAIS RGB
28.
29.     //COMPUTAÇÃO DOS VALORES DA LACUNARIDADE
30.
31.     //RETORNO DOS DADOS
32. }
```

Quadro 25 - Computação dos coeficientes da lacunaridade

Após o cálculo dos coeficientes da lacunaridade, a média dos valores dos canais RGB (na imagem em cores) e da intensidade dos pixels (na imagem em tons de cinza) é calculada, conforme exibido no trecho de código contido no Quadro 26.

```
01. /**Computa a feature, em cima do contexto, e joga em out_data*/
02. void Lacunarity::compute(FeatureExtractionContext& context, std::list<double>
    &out_data) const
03. {
04.     //CÁLCULO DOS COEFICIENTES DA LACUNARIDADE E ROI'S
05.
06.     //calcula média das cores e do grayscale
07.     cv::Vec3d colorMeans;
08.     double grMean;
09.     size_t totalPixels;
10.     ImageProcessingUtils::getColorMeans(colorROI, grayROI, binROI,
                                        &totalPixels, colorMeans, &grMean);
11.     //converte os dados para estrutura apropriada (vetor está em BGR)
12.     ColorInfo mean;
13.     mean.gray = grMean;
14.     mean.r = colorMeans[2];
15.     mean.g = colorMeans[1];
16.     mean.b = colorMeans[0];
17.
18.     //COMPUTAÇÃO DOS VALORES DA LACUNARIDADE
19.
20.     //RETORNO DOS DADOS
21. }
```

Quadro 26 - Cálculo da média dos canais para lacunaridade

Após o cálculo da média dos canais, o próximo passo é a aplicação da fórmula da lacunaridade, cuja implementação é exibida no Quadro 27.

```
01.  /**Computa a feature, em cima do contexto, e joga em out_data*/
02.  void Lacunarity::compute(FeatureExtractionContext& context, std::list<double>
     &out_data) const
03.  {
04.      //CÁLCULO DOS COEFICIENTES DA LACUNARIDADE E ROI'S
05.
06.      //CÁLCULO DA MÉDIA DOS VALORES DOS CANAIS RGB
07.
08.      //iteração para cálculo da lacunaridade
09.      for (int i = 0; i < colorROI.rows; ++i) {
10.          for (int j = 0; j < colorROI.cols; ++j) {
11.              //verifica se na imagem binária este pixel faz parte da folha
12.              if (binROI.at<uchar>(i, j)) {
13.                  //pega pixel da imagem colorida (BGR)
14.                  cv::Vec3b pixel = colorROI.at<cv::Vec3b>(i, j);
15.                  //valores rgb
16.                  uchar rVal = pixel[2];
17.                  uchar gVal = pixel[1];
18.                  uchar bVal = pixel[0];
19.                  //valor gray
20.                  uchar grayVal = grayROI.at<uchar>(i, j);
21.
22.                  //calcula a diferença entre os valores e sua média
23.                  double dGray = (grayVal / mean.gray) - 1;
24.                  double dR = (rVal / mean.r) - 1;
25.                  double dG = (gVal / mean.g) - 1;
26.                  double dB = (bVal / mean.b) - 1;
27.
28.                  //eleva as diferenças para cada coeficiente da lacunaridade e
29.                  //soma
30.                  for (size_t i = 0; i < coeficients.size(); ++i) {
31.                      int c = (int)coeficients[i];
32.                      coefData[i].gray += pow(dGray, c);
33.                      coefData[i].r += pow(dR, c);
34.                      coefData[i].g += pow(dG, c);
35.                      coefData[i].b += pow(dB, c);
36.                  }
37.              }
38.          }
39.      }
40.      //finaliza lacunaridade
41.      for (size_t i = 0; i < coeficients.size(); ++i) {
42.          double expoent = 1 / (double)coeficients[i];
43.          //finaliza cada um dos canais
44.          coefData[i].gray = pow(coefData[i].gray / totalPixels, expoent);
45.          coefData[i].r = pow(coefData[i].r / totalPixels, expoent);
46.          coefData[i].g = pow(coefData[i].g / totalPixels, expoent);
47.          coefData[i].b = pow(coefData[i].b / totalPixels, expoent);
48.      }
49.
50.      //RETORNO DOS DADOS
51.  }
```

Quadro 27 - Implementação da fórmula da lacunaridade

Por fim, os valores da lacunaridade computados para cada coeficiente, armazenados no *array* `coefData`, são copiados para o parâmetro de saída `out_data`, como pode ser observado no Quadro 28, a partir da linha 11.

```
01. /**Computa a feature, em cima do contexto, e joga em out_data*/
02. void Lacunarity::compute(FeatureExtractionContext& context, std::list<double>
    &out_data) const
03. {
04.     //CÁLCULO DOS COEFICIENTES DA LACUNARIDADE E ROI'S
05.
06.     //CÁLCULO DA MÉDIA DOS VALORES DOS CANAIS RGB
07.
08.     //COMPUTAÇÃO DOS VALORES DA LACUNARIDADE
09.
10.     //copia valores para output
11.     for (vector<ColorInfo>::iterator it = coefData.begin(); it !=
                                                        coefData.end(); ++it) {
12.         out_data.push_back(it->gray);
13.         out_data.push_back(it->r);
14.         out_data.push_back(it->g);
15.         out_data.push_back(it->b);
16.     }
17. }
```

Quadro 28 - Cópia dos valores da lacunaridade para resultado

3.3.1.2.7 Estatísticas de cor

Na API de reconhecimento de plantas, as estatísticas de cor (implementadas pela classe `ColorMoments`) correspondem aos valores estatísticos da média (Quadro 29), desvio padrão (Quadro 30), *skewness* (Quadro 31) e *kurtosis* (Quadro 32) dos canais RGB da imagem colorida da amostra.

$$\mu = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N} P_{ij}$$

Fonte: Kadir et al. (2011).

Quadro 29 - Fórmula do cálculo da média

$$\sigma = \sqrt{\frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}\left(P_{ij} - \mu\right)^2}$$

Fonte: Kadir et al. (2011).

Quadro 30 - Fórmula do cálculo do desvio padrão

$$\theta = \frac{\sum_{i=1}^{M}\sum_{j=1}^{N}\left(P_{ij} - \mu\right)^3}{MN\sigma^3}$$

Fonte: Kadir et al. (2011).

Quadro 31 - Fórmula do cálculo da *skewness*

$$\gamma = \frac{\sum_{i=1}^{M}\sum_{j=1}^{N}\left(P_{ij}-\mu\right)^4}{MN\sigma^4}$$

Fonte: Kadir et al. (2011).

Quadro 32 - Fórmula do cálculo da *kurtosis*

Nas fórmulas destacadas, M denota o número de colunas da imagem, N denota o número de linhas e P_{ij} a intensidade de cor do canal sendo calculado no pixel localizado na posição (i, j) na imagem.

O método responsável pelo cálculo da média dos canais é o `calcMean` (membro da classe `ColorMoments`), exibido no Quadro 33.

```
01. /**Calcula média das imagens*/
02. void ColorMoments::calcMean(WorkData& data) const
03. {
04.     //calcula média das cores (cinza não é utilizado)
05.     cv::Vec3d means;
06.     double dummyGray;
07.     ImageProcessingUtils::getColorMeans(data.colorROI, data.binROI,
                              data.binROI, &data.leafPixels, means, &dummyGray);
08.     //traduz o vetor para r,g,b [vetor está em BGR]
09.     data.mean.r = means[2];
10.     data.mean.g = means[1];
11.     data.mean.b = means[0];
12. }
```

Quadro 33 - Cálculo da média dos canais RGB

Com base nos valores da média dos canais RGB, é efetuado o cálculo do desvio padrão. O método utilizado para tal computação é o `calcStdDev` (presente também na classe `ColorMoments`) cuja implementação é apresentada no Quadro 34.

```
01. /**Calcula desvio padrão*/
02. void ColorMoments::calcStdDev(WorkData& data) const
03. {
04.     //desvio padrão
05.     for (int i = 0; i < data.colorROI.rows; ++i) {
06.         for (int j = 0; j < data.colorROI.cols; ++j) {
07.             //verifica se na imagem binária este pixel faz parte da folha
08.             if (data.binROI.at<uchar>(i, j)) {
09.                 //pega pixel da imagem colorida (BGR)
10.                 cv::Vec3b pixel = data.colorROI.at<cv::Vec3b>(i, j);
11.                 //soma a diferenca ao quadrado da intensidade para a média
12.                 data.std_dev.r += ((double)pixel[2] - data.mean.r) *
                                          ((double)pixel[2] - data.mean.r);
13.                 data.std_dev.g += ((double)pixel[1] - data.mean.g) *
                                         ((double)pixel[1] - data.mean.g);
14.                 data.std_dev.b += ((double)pixel[0] - data.mean.b) *
                                        ((double)pixel[0] - data.mean.b);
15.             }
16.         }
17.     }
18.     //finaliza desvio padrão
19.     data.std_dev.r = sqrt(data.std_dev.r / (double)data.leafPixels);
20.     data.std_dev.g = sqrt(data.std_dev.g / (double)data.leafPixels);
21.     data.std_dev.b = sqrt(data.std_dev.b / (double)data.leafPixels);
22. }
```

Quadro 34 - Cálculo do desvio padrão dos canais RGB

Finalmente, com os valores da média e desvio padrão calculados, é possível obter as informações *skewness*[2] (assimetria) e *kurtosis*[3] (curtose). Tais valores são obtidos a partir do método `calcKurtosisAndSkewness`, descrito no Quadro 35.

[2] Medida que simboliza a quantidade e direção da *skew*, desvio da simetria horizontal (BROWN, 2012).

[3] Medida que simboliza o quão alto e agudo é o pico central de uma distribuição, em relação à curva (em forma de sino) de uma distribuição normal (BROWN, 2012).

```
01. /**Calcula kurtosis e skewness*/
02. void ColorMoments::calcKurtosisAndSkewness(WorkData& data, bool skewness,
    bool kurtosis) const
03. {
04.     //skewness e kurtosis
05.     for (int i = 0; i < data.colorROI.rows; ++i) {
06.         for (int j = 0; j < data.colorROI.cols; ++j) {
07.             //verifica se na imagem binária este pixel faz parte da folha
08.             if (data.binROI.at<uchar>(i, j)) {
09.                 //pega pixel da imagem colorida (BGR)
10.                 cv::Vec3b pixel = data.colorROI.at<cv::Vec3b>(i, j);
11.                 //valores rgb
12.                 uchar rVal = pixel[2];
13.                 uchar gVal = pixel[1];
14.                 uchar bVal = pixel[0];
15.                 //diferenças para a média
16.                 double dR = (double)rVal - data.mean.r;
17.                 double dG = (double)gVal - data.mean.g;
18.                 double dB = (double)bVal - data.mean.b;
19.                 //soma da skewness
20.                 if (skewness) {
21.                     data.skewness.r += dR * dR * dR;
22.                     data.skewness.g += dG * dG * dG;
23.                     data.skewness.b += dB * dB * dB;
24.                 }
25.                 //soma para kurtosis
26.                 if (kurtosis) {
27.                     data.kurtosis.r += dR * dR * dR * dR;
28.                     data.kurtosis.g += dG * dG * dG * dG;
29.                     data.kurtosis.b += dB * dB * dB * dB;
30.                 }
31.             }
32.         }
33.     }
34.     //finaliza skewness e kurtosis
35.     for (size_t i = 0; i < 3; ++i) {
36.         if (skewness) {
37.             data.skewness.r /= (double)data.leafPixels * (data.std_dev.r *
                                    data.std_dev.r * data.std_dev.r);
38.             data.skewness.g /= (double)data.leafPixels * (data.std_dev.g *
                                    data.std_dev.g * data.std_dev.g);
39.             data.skewness.b /= (double)data.leafPixels * (data.std_dev.b *
                                    data.std_dev.b * data.std_dev.b);
40.         }
41.         if (kurtosis) {
42.             data.kurtosis.r /= (double)data.leafPixels * (data.std_dev.r *
                                data.std_dev.r * data.std_dev.r * data.std_dev.r);
43.             data.kurtosis.g /= (double)data.leafPixels * (data.std_dev.g *
                                data.std_dev.g * data.std_dev.g * data.std_dev.g);
44.             data.kurtosis.b /= (double)data.leafPixels * (data.std_dev.b *
                                data.std_dev.b * data.std_dev.b * data.std_dev.b);
45.         }
46.     }
47. }
```

Quadro 35 - Cálculo da *skewness* e *kurtosis*

A utilização ou não de cada um dos valores estatísticos como valor descritor para as imagens das amostras é parametrizável na API. Como os cálculos dos valores estatísticos podem apresentar dependências entre si, algumas combinações nessas parametrizações podem fazer com que determinado valor estatístico precise ser calculado apesar de não ser utilizado como descritor para as amostras.

Dentro do método `compute` (na classe `ColorMoments`), responsável pela computação

dos descritores estatísticos de cor, o primeiro passo é identificar quais valores estatísticos serão calculados e quais deles serão realmente utilizados como descritores. Este trecho da implementação é exibido no Quadro 36.

```
01. /**Computa a feature, em cima do contexto, e joga em out_data*/
02. void ColorMoments::compute(FeatureExtractionContext& context,
    std::list<double> &out_data) const
03. {
04.     //variável com dados de trabalho
05.     WorkData data;
06.
07.     //pega bounding box do contorno da imagem
08.     const cv::Rect& bbox = context.getBoundingBox();
09.     //cria ROI da colorida e da binária
10.     data.colorROI = context.getRawImg()(bbox);
11.     data.binROI = context.getBinaryImg()(bbox);
12.
13.     //pega parâmetros
14.     bool useMean = (context.getParamValue(
                        PLNT_PRMID_COLOR_MOMENTS_USE_MEAN).val == PLNT_TRUE);
15.     bool useStdDev = (context.getParamValue(
                        PLNT_PRMID_COLOR_MOMENTS_USE_STDDEV).val == PLNT_TRUE);
16.     bool useSkewness = (context.getParamValue(
                        PLNT_PRMID_COLOR_MOMENTS_USE_SKEWNESS).val == PLNT_TRUE);
17.     bool useKurtosis = (context.getParamValue(
                        PLNT_PRMID_COLOR_MOMENTS_USE_KURTOSIS).val == PLNT_TRUE);
18.
19.     //como o cálculo de uma característica pode depender de outra, calcula
20.     //até que nível deve calcular
21.     int level = 0;
22.     if (useSkewness || useKurtosis) level = 3;
23.     else if (useStdDev) level = 2;
24.     else if (useMean) level = 1;
25.
26.     //se não tem que calcular nada, retorna
27.     if (level == 0) return;
28.
29.     //EXECUÇÃO DOS CÁLCULOS ESTATÍSTICOS
30. }
```

Quadro 36 - Determinação dos cálculos estatísticos necessários

Uma vez determinados os cálculos necessários, estes são executados e seus resultados tornam-se descritores, se assim parametrizados. Este comportamento é ilustrado no trecho de código exibido no Quadro 37.

```
01. /**Computa a feature, em cima do contexto, e joga em out_data*/
02. void ColorMoments::compute(FeatureExtractionContext& context,
    std::list<double> &out_data) const
03. {
04.     //DETERMINAÇÃO DOS CÁLCULOS ESTATÍSTICOS NECESSÁRIOS
05.
06.     //calcula média
07.     this->calcMean(data);
08.
09.     //verifica se precisa calcular além da média
10.     if (level > 1) {
11.         //calcula desvio padrão
12.         this->calcStdDev(data);
13.         //verifica se precisa calcular além
14.         if (level > 2) {
15.             this->calcKurtosisAndSkewness(data, useSkewness, useKurtosis);
16.         }
17.     }
18.
19.     //adiciona no resultado
20.     if (useMean) {
21.         out_data.push_back(data.mean.r);
22.         out_data.push_back(data.mean.g);
23.         out_data.push_back(data.mean.b);
24.     }
25.     if (useStdDev) {
26.         out_data.push_back(data.std_dev.r);
27.         out_data.push_back(data.std_dev.g);
28.         out_data.push_back(data.std_dev.b);
29.     }
30.     if (useSkewness) {
31.         out_data.push_back(data.skewness.r);
32.         out_data.push_back(data.skewness.g);
33.         out_data.push_back(data.skewness.b);
34.     }
35.     if (useKurtosis) {
36.         out_data.push_back(data.kurtosis.r);
37.         out_data.push_back(data.kurtosis.g);
38.         out_data.push_back(data.kurtosis.b);
39.     }
40. }
```

Quadro 37 - Execução dos cálculos estatísticos

3.3.1.3 Classificadores

Dentro da API de reconhecimento de plantas, classificadores são entidades cujo objetivo é determinar a espécie de uma amostra a partir da comparação do seu vetor de valores descritores com os de outras espécies, armazenados na base de dados.

Atualmente, o único classificador implementado na API é uma implementação simples de uma Rede Neural Probabilística (RNP).

3.3.1.3.1 Rede Neural Probabilística (RNP)

Na API de reconhecimento de plantas, a classe `PNN` implementa uma Rede Neural Probabilística (RNP) simples, sendo responsável por categorizar uma amostra de entrada em uma espécie contida na base de dados de amostras. Esta classe contém rotinas responsáveis por carregar informações das amostras presentes na base de dados, calcular coeficientes de normalização das características, normalizar vetores de descritores e classificar vetores de descritores de entrada como pertencentes à uma espécie.

A RNP implementada neste projeto possui estrutura similar à apresentada na seção 2.4. Ela considera que todas as amostras registradas na base de dados possuem o mesmo coeficiente de suavização para o cálculo da distância (não há vetor de bias) e que todos os descritores contribuem igualmente na classificação (não há vetor de pesos). Desta maneira, a fórmula utilizada para o cálculo do coeficiente de similaridade utilizada pela RNP implementada corresponde à apresentada no Quadro 38.

$$p(x|w_j) = \frac{1}{n_j} \sum_{k=1}^{n_j} \exp\left(-\frac{(x - X_k)^2}{2\sigma^2}\right)$$

Fonte: Kadir et al. (2011).

Quadro 38 - Fórmula para cálculo de similaridade utilizada na RNP

Na fórmula citada, x denota o vetor de descritores extraído da amostra a ser classificada, w_j simboliza a espécie para qual a amostra está sendo testada, n_j denota o número de amostras registradas na base de dados para a espécie em questão, X_k representa o vetor de descritores da k-ésima amostra da espécie na base de dados e σ o coeficiente de suavização utilizado.

O método `load` (presente na classe `PNN`) é derivado a partir do contrato com a interface `IClassifier` e responsável por efetuar o carregamento das informações necessárias para o funcionamento do classificador, como exibido no Quadro 39.

```
01. /**Carrega classificador a partir de dados da database*/
02. void PNN::load(Database& db)
03. {
04.     //pega sigma
05.     PLNT_PARAM_VAL sigmaParam;
06.     db.getParamValue(PLNT_PRMID_PNN_SIGMA, &sigmaParam);
07.     //seta sigma e limpa coleção atual de dados de teste
08.     _sigma = sigmaParam.val;
09.     _data.clear();
10.     _featNormalization.clear();
11.
12.     //pega todos os dados de samples
13.     this->loadSamples(db);
14.
15.     //verifica se tem algum dado
16.     if (_data.size() < 1) return;
17.
18.     //carrega vetor de normalização (min/max)
19.     this->loadNormalizationVector();
20.
21.     //percorre mapa das samples, normalizando seus valores
22.     for (std::map<PLNT_SPEC_ID, SampleValueTable>::iterator it =_data.begin();
                                    it != _data.end(); ++it)
23.     {
24.         //percorre samples da classe
25.         SampleValueTable &sampTab = it->second;
26.         for (SampleValueTable::iterator sampIt = sampTab.begin();
                                      sampIt != sampTab.end(); ++sampIt) {
27.             //normaliza features
28.             this->normalizeFeatures(*sampIt);
29.         }
30.     }
31. }
```

Quadro 39 - Carregamento das informações do classificador

O carregamento dos dados do classificador, conforme apresentado no Quadro 39, consiste na obtenção do valor parametrizado para o sigma, carregamento da lista de amostras armazenadas na base de dados (a partir do método `loadSamples`), computação do vetor de normalização para os valores descritores (método `loadNormalizationVector`) e finalmente a normalização dos descritores de cada amostra carregada (chamadas ao método `normalizeFeatures`).

No Quadro 40 é exibida a implementação do método `loadSamples`, responsável por efetuar a leitura dos vetores de valores descritores presentes na base de dados da API.

```
01. /**Carrega samples da base de dados*/
02. void PNN::loadSamples(Database &db)
03. {
04.     //inicializa vetor com a quantidade necessária
05.     PLNT_SIZE64 count = db.getSampleCount();
06.     std::vector<PLNT_SAMPLE_ID> sampleIds((size_t)count);
07.     //pega ponteiro e pede para a database preencher
08.     PLNT_SAMPLE_ID* idArr = &sampleIds[0];
09.     db.getSamples(idArr);
10.
11.     //percorre id's de samples
12.     for (std::vector<PLNT_SAMPLE_ID>::iterator it = sampleIds.begin();
                                                    it != sampleIds.end(); ++it)
13.     {
14.         //cria info de sample
15.         PLNT_SAMPLE_INFO* sampleInfo = new PLNT_SAMPLE_INFO();
16.
17.         try {
18.             //pega info
19.             db.getSampleInfo(*it, sampleInfo);
20.             //pega lista de features da classe
21.             SampleValueTable &valTab = _data[sampleInfo->spec];
22.             //adiciona novo vetor de valores na lista
23.             valTab.push_back(SampleValues());
24.             SampleValues &dataVector = valTab.back();
25.             //copia os valores para o vetor
26.             dataVector.resize((size_t)sampleInfo->featDataCount);
27.             for (size_t i = 0; i < dataVector.size(); ++i) {
28.                 dataVector[i] = sampleInfo->featData[i].value;
29.             }
30.         }
31.         catch (std::exception& ex) {
32.             ModelHelper::destroySampleInfo(sampleInfo);
33.             throw ex;
34.         }
35.         //libera memória
36.         ModelHelper::destroySampleInfo(sampleInfo);
37.     }
38. }
```

Quadro 40 - Carregamento de amostras a partir da base de dados

O método `loadNormalizationVector`, cuja função é determinar um vetor com informações sobre o valor máximo e mínimo encontrado na base de dados para cada descritor, é exibido no Quadro 41.

```
01. /**Carrega vetor de normalização de dados de features*/
02. void PNN::loadNormalizationVector()
03. {
04.     //percorre samples
05.     bool first = true;
06.     for (std::map<PLNT_SPEC_ID, SampleValueTable>::iterator it =_data.begin();
                                        it != _data.end(); ++it)
07.     {
08.         //percorre samples da classe
09.         SampleValueTable &sampTab = it->second;
10.         for (SampleValueTable::iterator sampIt = sampTab.begin();
                                    sampIt != sampTab.end(); ++sampIt) {
11.             //verifica se é a primeira entry
12.             if (first) {
13.                 first = false;
14.                 //inicializa vetor de normalização com a primeira entry
15.                 _featNormalization.resize(sampIt->size());
16.                 for (size_t j = 0; j < _featNormalization.size(); ++j) {
17.                     MinMaxInfo& minMax = _featNormalization[j];
18.                     minMax.min = minMax.max = sampIt->at(j);
19.                 }
20.             }
21.             else {
22.                 //cálculo de min/max de cada valor de feature
23.                 for (size_t j = 0; j < _featNormalization.size(); ++j) {
24.                     MinMaxInfo& minMax = _featNormalization[j];
25.                     double val = sampIt->at(j);
26.                     //checagem de novo máximo ou mínimo
27.                     if (val < minMax.min) { minMax.min = val; }
28.                     else if(val > minMax.max) { minMax.max = val; }
29.                 }
30.             }
31.         }
32.     }
33. }
```

Quadro 41 - Computação do vetor de normalização

O método `normalizeFeatures` é utilizado para normalizar (transformar o valor original para um novo valor, dentro do intervalo de 0 à 1) um conjunto de valores descritores utilizando o vetor de normalização previamente calculado. Sua implementação está descrita no Quadro 42.

```
01. /**Normaliza features*/
02. void PNN::normalizeFeatures(SampleValues &values) const
03. {
04.     //percorre features
05.     for (size_t j = 0; j < _featNormalization.size(); ++j) {
06.         //normaliza feature usando info de min e max
07.         const MinMaxInfo& minMax = _featNormalization[j];
08.         double normalized = (values[j] - minMax.min) /
                                        (minMax.max - minMax.min);
09.         //joga novamente para o vetor
10.         values[j] = normalized;
11.     }
12. }
```

Quadro 42 - Normalização de um vetor de valores descritores

O método `classify`, implementado como resultado do contrato com a interface `IClassifier`, é o método responsável por classificar uma amostra de entrada (a partir do seu vetor de descritores) em uma espécie existente na base de dados. É neste método que, efetivamente a implementação das camadas da RNP está contida. O Quadro 43 apresenta a

implementação do método `classify`.

```
01. /**Classifica uma amostra em uma classe*/
02. PLNT_SAMPLE_ID PNN::classify(const std::vector<PLNT_FEAT_DATA> &sample) const
03. {
04.     //copia vetor de samples e normaliza ele
05.     SampleValues input(sample.size());
06.     for (size_t i = 0; i < input.size(); ++i) {
07.         input[i] = sample[i].value;
08.     }
09.     this->normalizeFeatures(input);
10.     //precalcula o fator de smooth usado no gaussiano.
11.     //o gaussiano usado é exp(- [u / 2*sigma^2] ),
12.     //então pré-calcula -2*sigma^2
13.     double smooth = -2 * (_sigma * _sigma);
14.
15.     //classe com maior valor de likelihood para a sample
16.     PLNT_SAMPLE_ID resultClass = 0;
17.     double maxLikelihood = std::numeric_limits<double>::min();
18.     //percorre cada classe
19.     for (std::map<PLNT_SAMPLE_ID, SampleValueTable>::const_iterator it =
                                        _data.begin(); it != _data.end(); ++it)
20.     {
21.         //matriz com samples da classe
22.         const SampleValueTable &samples = it->second;
23.         //somatória das PDF's das samples da classe
24.         double pdfSum = 0;
25.         //percorre cada sample
26.         for (SampleValueTable::const_iterator sampIt = samples.begin();
                                            sampIt != samples.end(); ++sampIt) {
27.             //calcula distância entre as features
28.             double distance = 0;
29.             for (size_t j = 0; j < sampIt->size(); ++j) {
30.                 double val = sampIt->at(j) - input.at(j);
31.                 distance += (val * val);
32.             }
33.             //exponencia e adiciona na soma da classe
34.             pdfSum += exp(distance / smooth);
35.         }
36.         //normaliza entre o número de samples da classe
37.         pdfSum /= samples.size();
38.         //verifica se é o que possui maior likelihood
39.         if (pdfSum > maxLikelihood) {
40.             resultClass = it->first;
41.             maxLikelihood = pdfSum;
42.         }
43.     }
44.     //retorna classe em que foi classificado o input
45.     return resultClass;
46. }
```

Quadro 43 - Método de classificação de amostras

3.3.2 Operacionalidade da implementação

Em virtude da natureza do projeto (API disponibilizada como uma *Dynamic Link Library* (DLL) com funções exportadas no padrão C), um projeto de demonstração das funcionalidades da API foi desenvolvido utilizando a linguagem C# da plataforma .NET, ilustrando a possibilidade de integração da API com outras linguagens.

Na aplicação de demonstração o usuário tem a possibilidade realizar as seguintes tarefas:

a) criação de uma nova base de dados de plantas: indicar as características das amostras que serão levadas em consideração, o classificador utilizado no processo de decisão da espécie e os parâmetros de execução relevantes para as características e o classificador;
b) adição de uma amostra de folha na base de dados: vincular a amostra à uma espécie de planta, tendo a possibilidade de registrar essa nova espécie de planta durante o processo de catalogação. É possível informar um valor textual arbitrário e vinculá-lo à amostra, assim como definir a imagem da amostra como a representação padrão para a espécie.
c) classificação de uma amostra: classificar uma planta de acordo com as espécies registradas na base de dados.

As seções seguintes ilustram a operacionalidade de cada um dos itens acima, onde cada um possui uma tela específica.

3.3.2.1 Criação de nova base de dados

O formulário para a criação de uma nova base de dados pode ser acessado a partir do menu superior da aplicação, selecionando a opção “Funções” e em seguida “Nova base de dados”, como exibido na Figura 32.

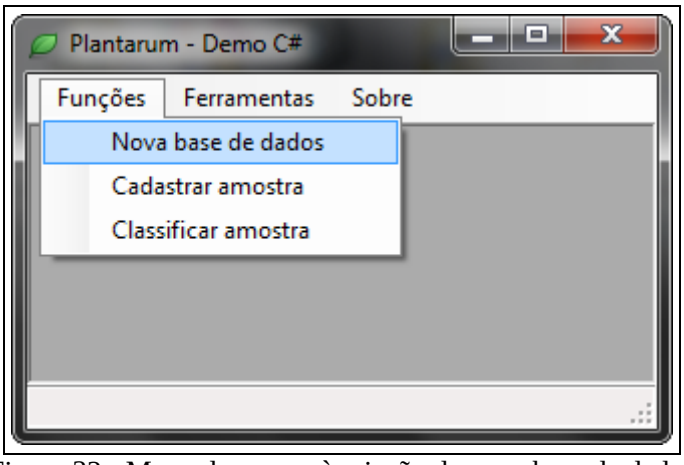


Figura 32 - Menu de acesso à criação de nova base de dados

Após a seleção do menu em questão, a tela de criação de base de dados é exibida, como ilustra a Figura 33.

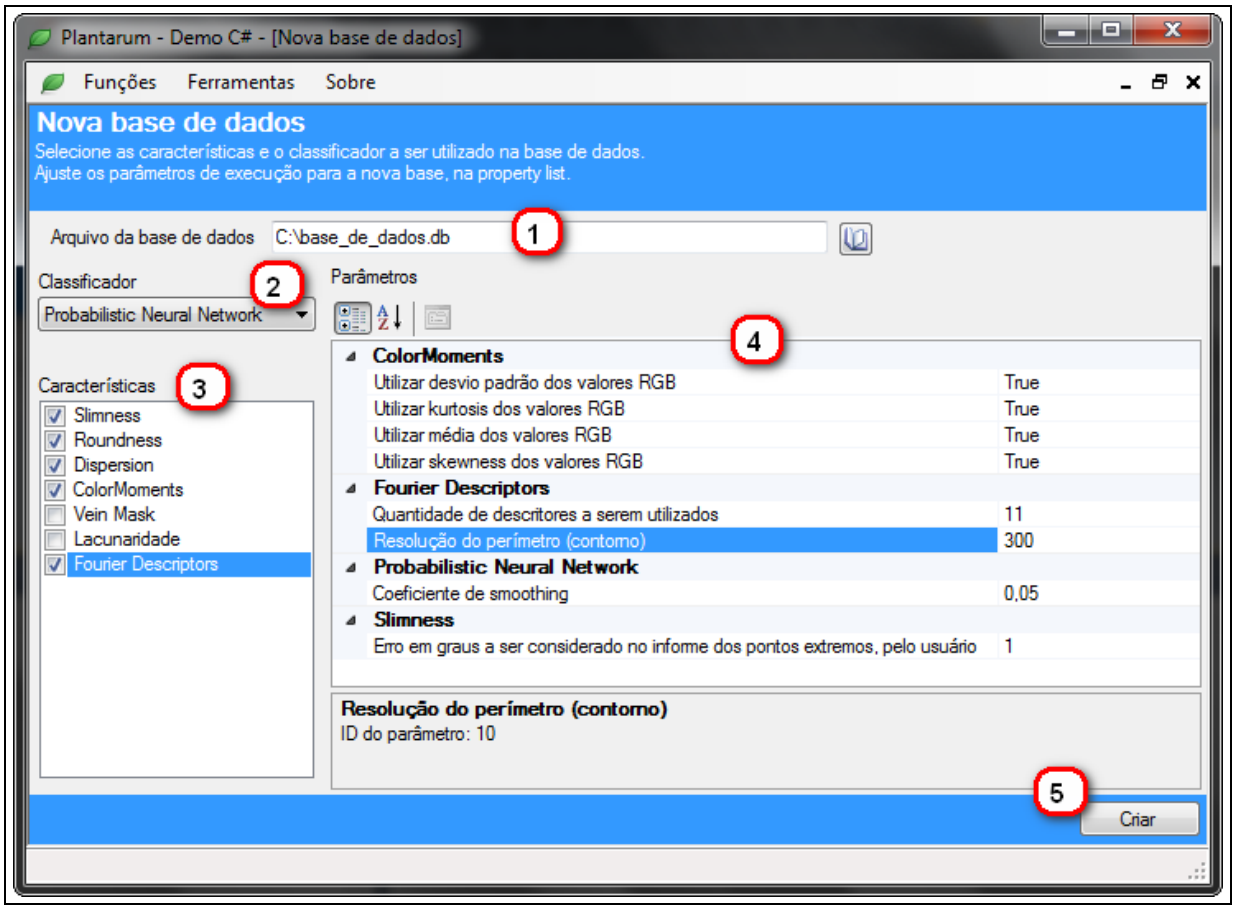


Figura 33 - Tela de criação de nova base de dados

No formulário de criação, o usuário informa o caminho do arquivo da base de dados a ser criado (item 1).

Na lista de seleção destacada no item 2, o usuário pode selecionar o tipo de classificador utilizado para determinar as espécies de plantas na base de dados em criação.

O item 3 compreende a listagem de características disponibilizadas pela API. O usuário pode selecionar as que serão utilizadas para discriminar as amostras presentas na nova base de dados.

Parâmetros de configuração do classificador e características selecionados são exibidos na lista de propriedades assinalado pelo item 4. Antes da criação da base de dados, o usuário pode configurar quais serão os valores considerados para cada parâmetro.

Após informar o classificador, características e parâmetros desejados para a base de dados, o usuário pressiona o botão “Criar” (destacado como item 5) e a API efetua a criação da nova base de dados, com todas as tabelas e configurações necessárias para a execução da API.

Ao final do processo de criação da base de dados, uma mensagem de sucesso é exibida ao usuário, como ilustrado na Figura 34.

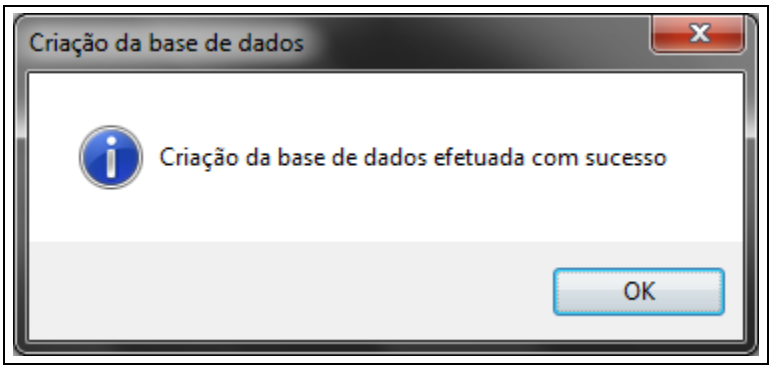


Figura 34 - Mensagem de sucesso após criação da base de dados

3.3.2.2 Inclusão de amostra

Para efetuar a inclusão de uma amostra, o usuário deve selecionar, no menu superior da aplicação, o item “Funções” seguido de “Cadastrar amostra”, como exibido na Figura 35.

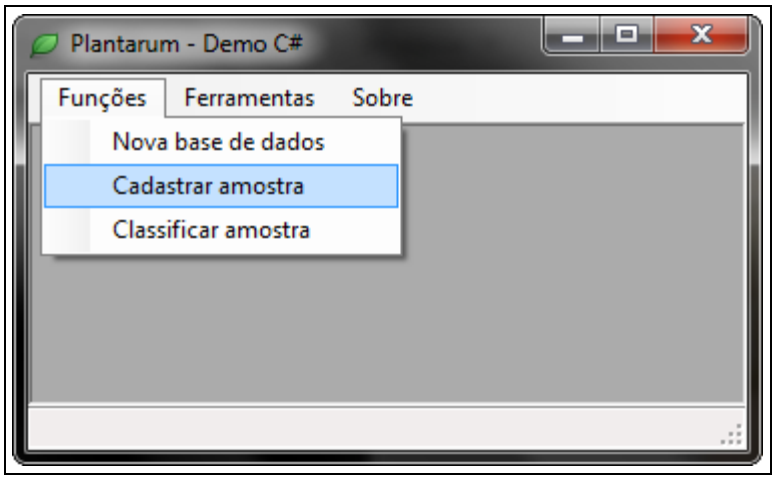


Figura 35 – Menu de acesso ao cadastro de amostras

Depois de efetuada a seleção da opção de menu mencionada, o formulário de inclusão de nova amostra é apresentado ao usuário, porém, inicialmente requisitando o caminho da base de dados onde serão armazenadas as novas amostras, como ilustrado na Figura 36.

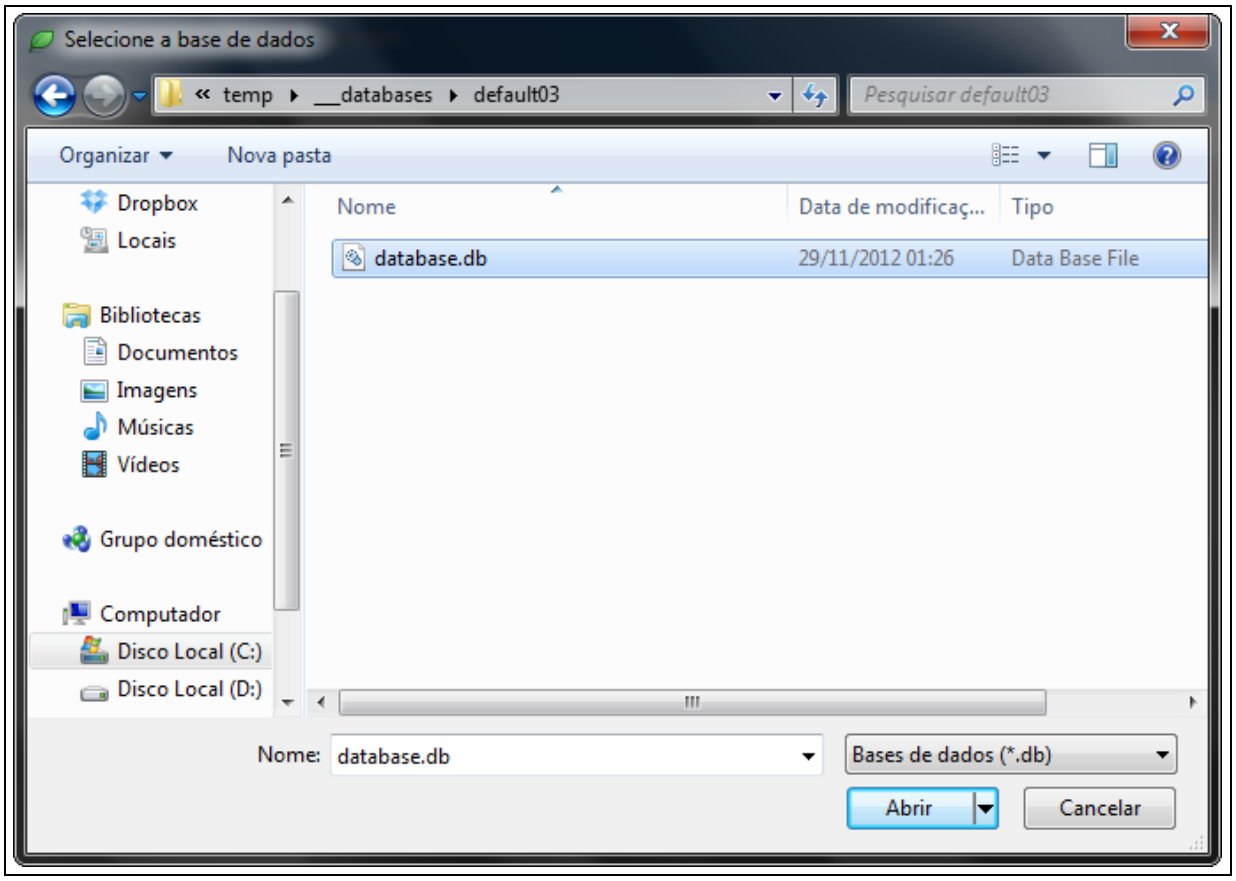


Figura 36 - Seleção de base de dados

Após a seleção da base de dados a ser utilizada para o cadastro das novas amostras de folhas, o formulário de cadastro de amostras é apresentado, como ilustrado na Figura 37.

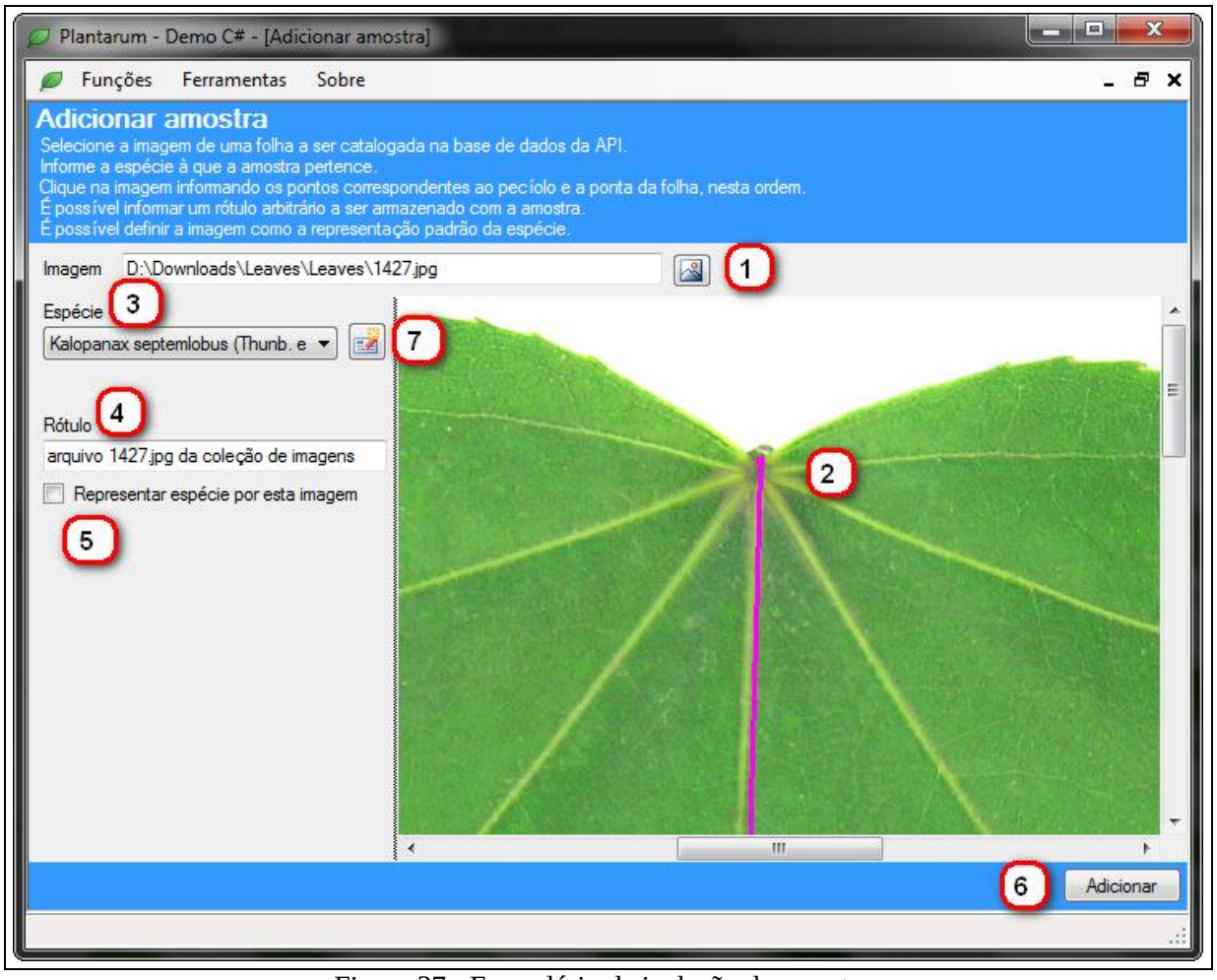


Figura 37 - Formulário de inclusão de amostras

A imagem da folha a ser registrada na base de dados é informada no campo identificado como item 1. A imagem é exibida no painel indicado pelo item 2, onde o usuário deve selecionar na imagem (pressionando com o botão esquerdo do mouse) os pontos representando o pecíolo e a ponta da folha.

A seleção da espécie da amostra é feita a partir da seleção de uma das espécies existentes na base de dados, apresentadas na lista de seleção marcada como o item 3.

Para cada amostra registrada na base de dados é possível vincular um rótulo (informação textual arbitrária, que pode ser utilizada para algum tipo de identificação ou processamento posterior). Tal campo é indicado pelo item 4.

Cada espécie de planta registrada em uma base de dados da API pode possuir uma imagem atrelada, com o intuito de prover uma representação visual da espécie. No campo apresentado como item 5, o usuário pode decidir se a imagem da amostra será utilizada para representar a espécie.

Após a seleção da imagem da amostra, dos pontos do pecíolo e da ponta da folha, da espécie e da decisão da utilização ou não da imagem como representativa da espécie, o

usuário pode pressionar o botão “Adicionar” (item 6) e efetuar o registro das características da amostra selecionada na base de dados selecionada. Uma mensagem de sucesso é apresentada ao final do processo, como ilustrado na Figura 38.

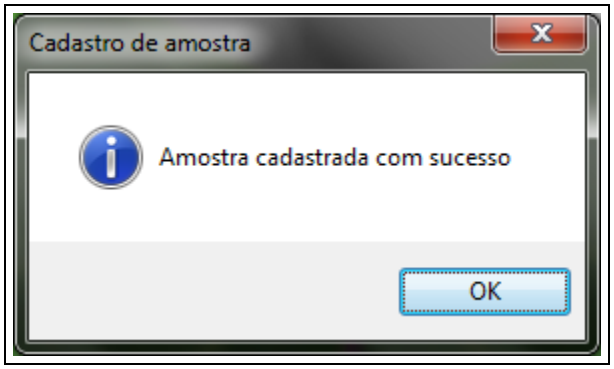


Figura 38 - Mensagem de sucesso do cadastro de amostras

No caso da amostra selecionada possuir uma espécie que ainda não esteja registrada na base de dados selecionada, o usuário tem a opção de efetuar o registro de uma nova espécie de planta, pressionando o botão indicado pelo item 7 na Figura 37. Ao pressionar este botão, uma janela solicitando o nome da nova espécie é apresentada, como visto na Figura 39.

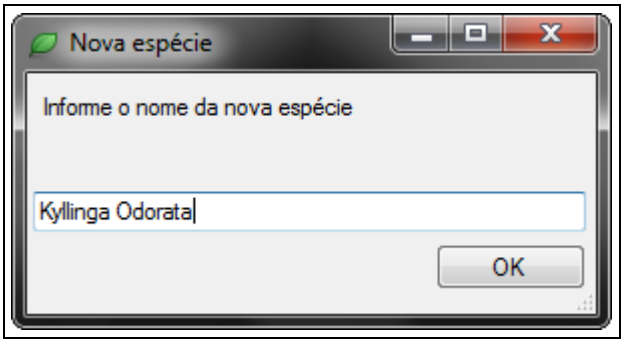


Figura 39 - Formulário para registro de nova espécie

Após informar o nome da nova espécie, o usuário confirma a inclusão do registro pressionando o botão “OK”. No final deste processo, uma mensagem de sucesso é exibida ao usuário, conforme Figura 40.

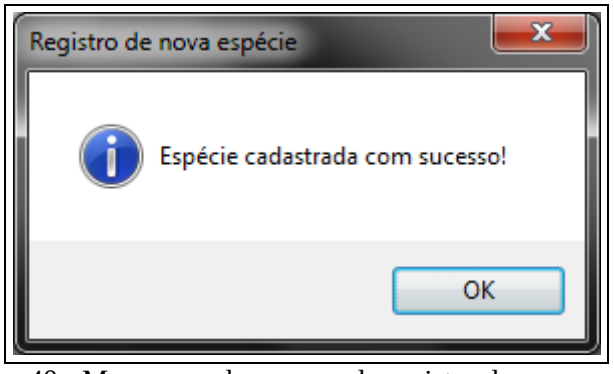


Figura 40 - Mensagem de sucesso do registro de nova espécie

3.3.2.3 Classificação de amostra

Para efetuar a classificação de uma amostra, deve-se acessar a tela de classificação de amostra a partir do menu “Funções” e então “Classificar amostra”, como apresentado na Figura 41.

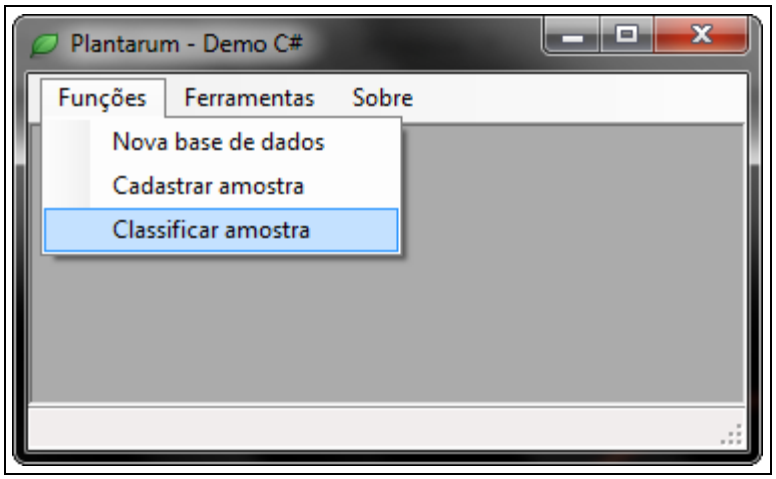


Figura 41 - Menu para acesso à classificação de amostras

Assim como no formulário de cadastro de amostras, a janela para seleção de base de dados é apresentada para o usuário (vide Figura 36). Após a seleção do arquivo de base de dados desejado, o formulário para classificação de amostras é exibido, como ilustrado na Figura 42.

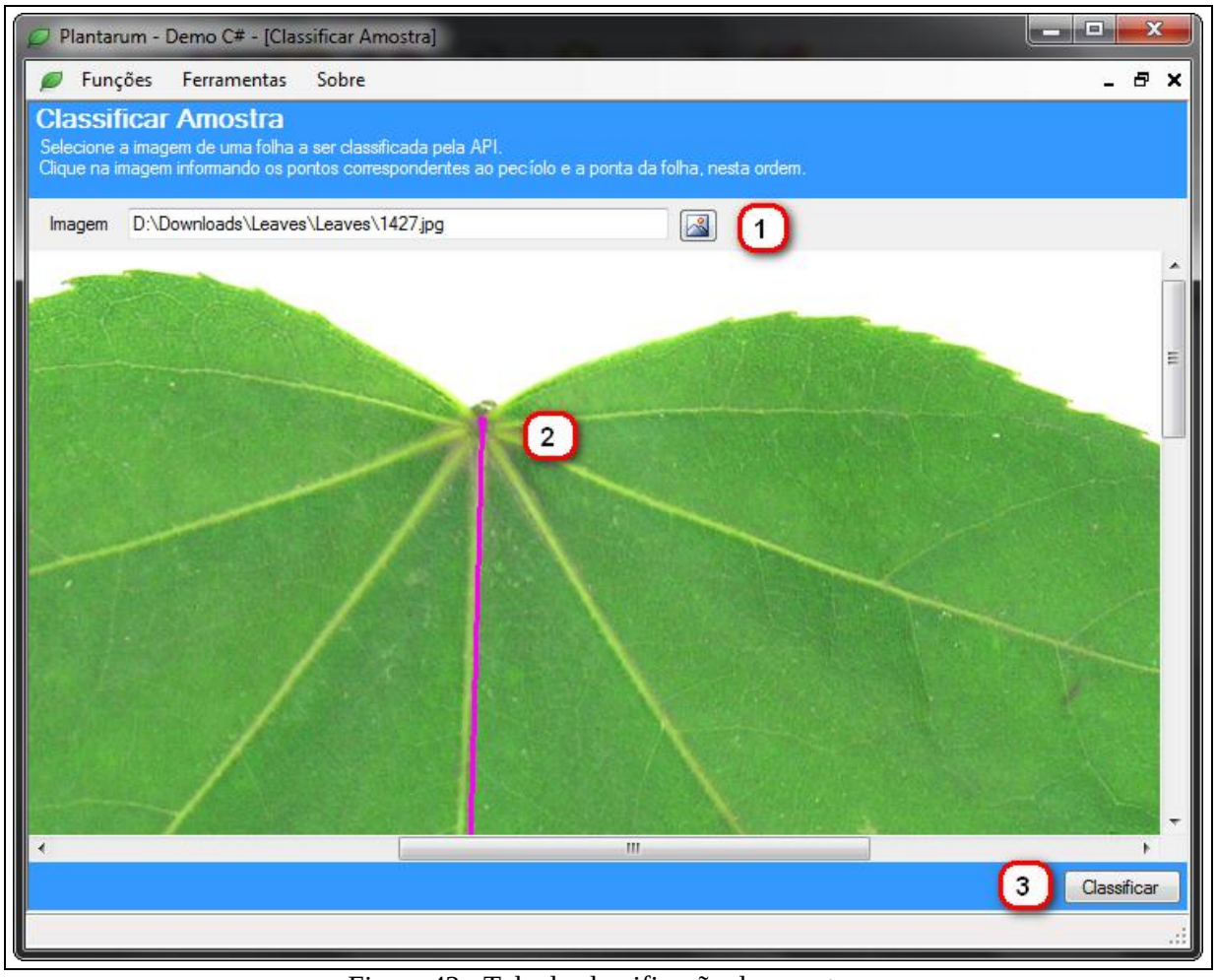


Figura 42 - Tela de classificação de amostras

Na tela de classificação de amostras, a imagem da folha da planta deve ser informada no campo demarcado como item 1 sendo, então, exibida no painel indicado como item 2. Assim como no formulário de adição de novas amostras, o usuário informa os pontos correspondentes ao pecíolo e a ponta da folha, pressionando o com o botão esquerdo do mouse na imagem.

Após a seleção da imagem e dos pontos indicando o pecíolo e a ponta da folha, o processo de classificação da amostra é acionado após o pressionamento do botão “Classificar” (item 3), que efetua a chamada da rotina de classificação da API e apresenta o resultado do processo na janela de resultado exibida na Figura 43.

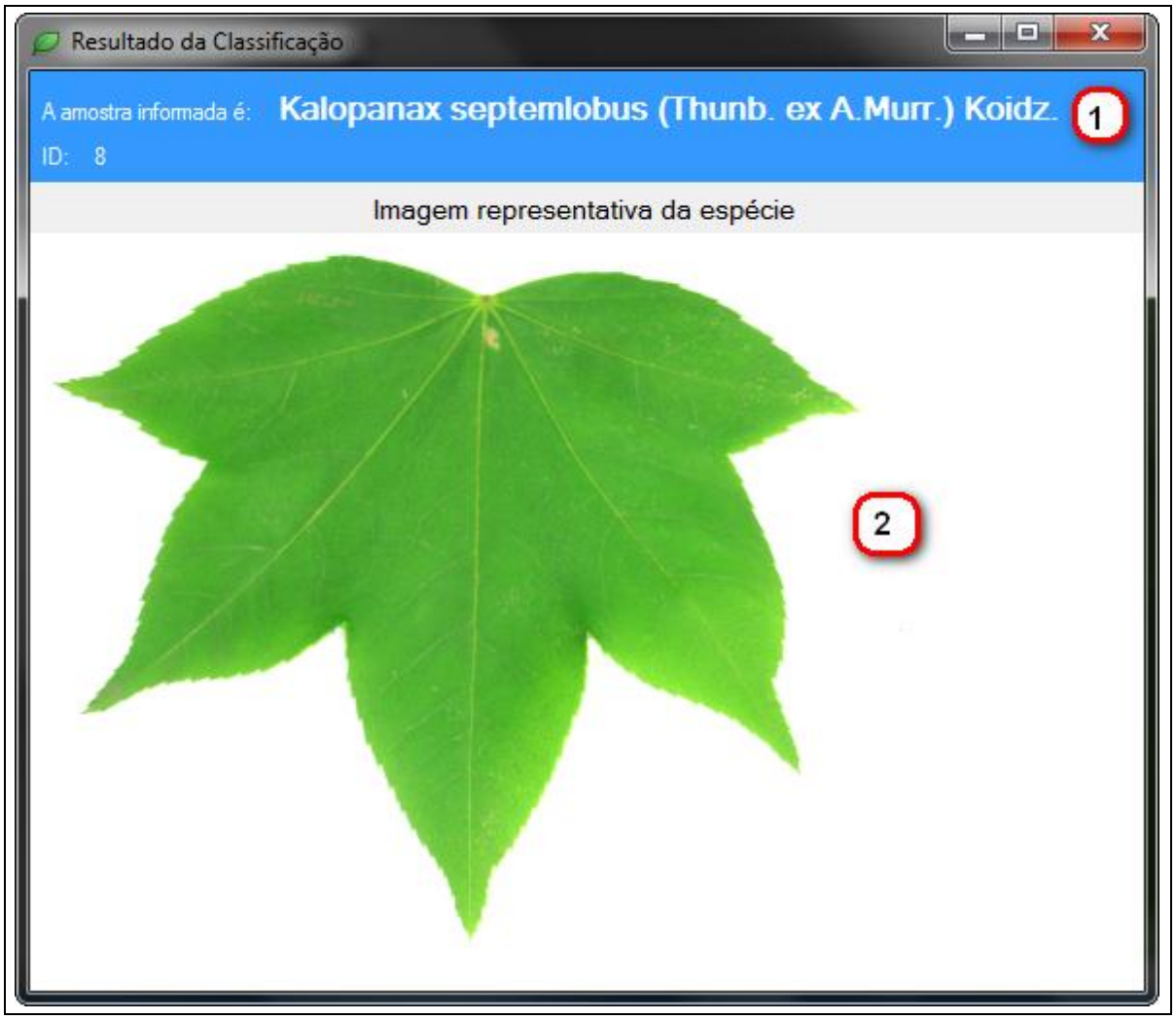


Figura 43 - Tela de resultado de classificação de amostra

Na tela de resultado da classificação o nome e o código identificador da espécie são apresentados no painel superior do formulário, indicado pelo item 1.

Caso uma imagem representativa para a espécie ter sido previamente especificada no momento do registro de amostras, esta é exibida no painel identificado como o item 2.

3.4 RESULTADOS E DISCUSSÃO

Para a validação deste projeto, foi efetuada uma bateria de testes utilizando a coleção de imagens de folhas também empregada por Wu et al. (2007) no desenvolvimento de seu trabalho. Tal coleção de imagens, denominada Flavia *dataset*, é uma coleção composta de imagens de folhas gratuitamente disponibilizada por Wu et al. (2007), no intuito de auxiliar projetos relacionados à classificação de plantas. A coleção contém, atualmente, 32 espécies

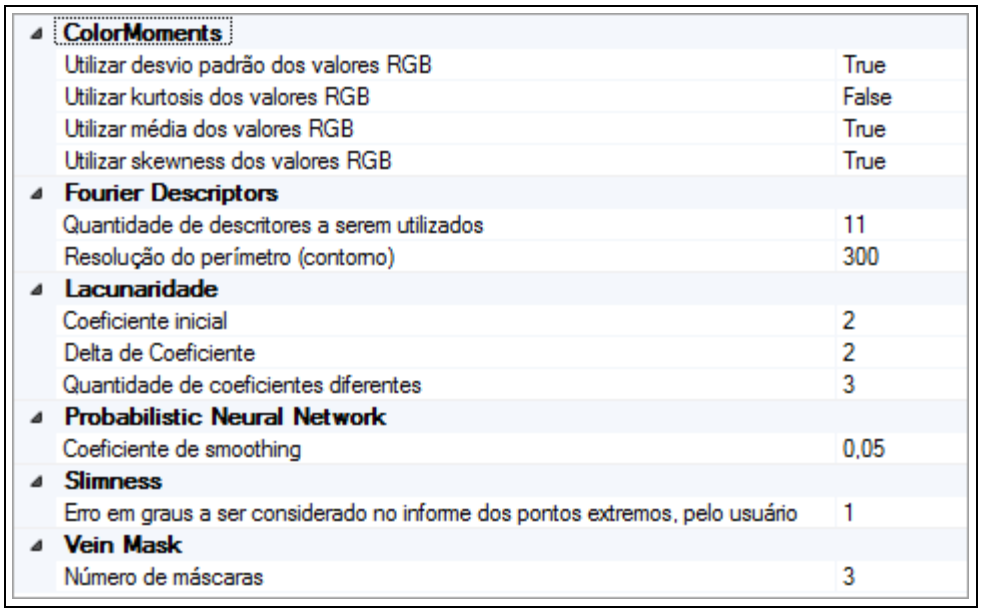
diferentes de de folhas, com uma média de 60 imagens por espécie. A Tabela 1 lista as espécies contidas na coleção de imagens utilizada na validação deste projeto.

Tabela 1 - Espécies de plantas utilizadas na validação da API

ESPÉCIES UTILIZADAS NA VALIDAÇÃO DA API	
Nome da espécie	**Número de amostras**
Acer Palmatum	56
Acer buergerianum Miq.	53
Aesculus chinensis	63
Berberis anhweiensis Ahrendt	65
Cedrus deodara (Roxb.) G. Don	77
Cercis chinensis	72
Chimonanthus praecox L.	52
Cinnamomum camphora (L.) J. Presl	65
Cinnamomum japonicum Sieb.	55
Citrus reticulata Blanco	56
Ginkgo biloba L.	62
Ilex macrocarpa Oliv.	50
Indigofera tinctoria L.	73
Kalopanax septemlobus (Thunb. ex A.Murr.) Koidz.	52
Koelreuteria paniculata Laxm.	59
Lagerstroemia indica (L.) Pers.	61
Ligustrum lucidum Ait. f.	55
Liriodendron chinense (Hemsl.) Sarg.	53
Magnolia grandiflora L.	57
Mahonia bealei (Fortune) Carr.	55
Manglietia fordiana Oliv.	52
Nerium oleander L.	66
Osmanthus fragrans Lour.	56
Phoebe nanmu (Oliv.) Gamble	62
Phyllostachys edulis (Carr.) Houz.	59
Pittosporum tobira (Thunb.) Ait. f.	63
Podocarpus macrophyllus (Thunb.) Sweet	60
Populus - canadensis Moench	64
Prunus persica (L.) Batsch	54
Prunus serrulata Lindl. var. lannesiana auct.	55
Tonna sinensis M. Roem.	65
Viburnum awabuki K.Koch	60
	1907

Para a execução dos testes, a coleção de testes foi dividida de forma que 75% das amostras de cada espécie fossem registradas na base de dados da API e o restante utilizado como coleção a ser classificada.

A parametrização inicial da base de dados utilizada nos testes é baseada em grande parte nos valores utilizados por Kadir et al. (2011), uma vez que seu trabalho apresenta resultados utilizando várias combinações das características também desenvolvidas neste projeto. A Figura 44 exibe os valores iniciais adotados para os parâmetros.



ColorMoments	
Utilizar desvio padrão dos valores RGB	True
Utilizar kurtosis dos valores RGB	False
Utilizar média dos valores RGB	True
Utilizar skewness dos valores RGB	True
Fourier Descriptors	
Quantidade de descritores a serem utilizados	11
Resolução do perímetro (contorno)	300
Lacunaridade	
Coeficiente inicial	2
Delta de Coeficiente	2
Quantidade de coeficientes diferentes	3
Probabilistic Neural Network	
Coeficiente de smoothing	0,05
Slimness	
Erro em graus a ser considerado no informe dos pontos extremos, pelo usuário	1
Vein Mask	
Número de máscaras	3

Figura 44 - Parametrização inicial da base de dados de validação da API

Primeiramente foi realizado um teste utilizando apenas a característica “Descritores de Fourier”, que neste projeto é implementada a partir da transformada unidimensional de Fourier em cima dos pontos do contorno da folha. Os resultados desta série de testes são exibidos na Tabela 2.

Tabela 2 - Resultado da bateria de testes número 1

RESULTADO DOS TESTES DE CLASSIFICAÇÃO (TESTE 1)			
Espécie	**Número de amostras de teste**	**Número de classificações corretas**	**Percentual de acerto**
Acer buergerianum Miq.	14	14	100,00%
Acer Palmatum	14	13	92,86%
Aesculus chinensis	16	6	37,50%
Berberis anhweiensis Ahrendt	17	16	94,12%
Cedrus deodara (Roxb.) G. Don	20	20	100,00%
Cercis chinensis	18	17	94,44%
Chimonanthus praecox L.	13	8	61,54%
Cinnamomum camphora (L.) J. Presl	17	12	70,59%
Cinnamomum japonicum Sieb.	14	3	21,43%
Citrus reticulata Blanco	14	11	78,57%
Ginkgo biloba L.	16	14	87,50%
Ilex macrocarpa Oliv.	13	9	69,23%
Indigofera tinctoria L.	19	17	89,47%
Kalopanax septemlobus (Thunb. ex A.Murr.) Koidz.	13	13	100,00%
Koelreuteria paniculata Laxm.	15	4	26,67%
Lagerstroemia indica (L.) Pers.	16	3	18,75%
Ligustrum lucidum Ait. f.	14	1	7,14%
Liriodendron chinense (Hemsl.) Sarg.	14	12	85,71%
Magnolia grandiflora L.	15	8	53,33%
Mahonia bealei (Fortune) Carr.	14	11	78,57%
Manglietia fordiana Oliv.	13	2	15,38%
Nerium oleander L.	17	12	70,59%
Osmanthus fragrans Lour.	14	1	7,14%
Phoebe nanmu (Oliv.) Gamble	16	13	81,25%
Phyllostachys edulis (Carr.) Houz.	15	8	53,33%
Pittosporum tobira (Thunb.) Ait. f.	16	14	87,50%
Podocarpus macrophyllus (Thunb.) Sweet	15	13	86,67%
Populus - canadensis Moench	16	16	100,00%
Prunus persica (L.) Batsch	14	7	50,00%
Prunus serrulata Lindl. var. lannesiana auct.	14	9	64,29%
Tonna sinensis M. Roem.	17	12	70,59%
Viburnum awabuki K.Koch	15	7	46,67%
	488	326	66,80%

Kadir et al. (2011) apresentam um teste semelhante, utilizando unicamente a *Polar Fourier Transform* (PFT), reportando uma precisão geral de 74,69% na classificação. A diferença considerável na performance da classificação entre o trabalho de Kadir et al. (2011) e este projeto, indica que, individualmente, a variação do algoritmo escolhida por Kadir et al. (2011) é um melhor descritor que a variação escolhida neste projeto.

A Tabela 2 demonstra também que várias espécies apresentaram uma performance muito ruim (abaixo de 50%, destacadas na cor laranja) com a utilização de apenas a

característica “Descritores de Fourier”.

Na segunda série de testes, utilizou-se juntamente com os descritores de Fourier, as características de natureza geométrica: circularidade, dispersão e magreza. O resultado da série de testes é exibido na Tabela 3.

Tabela 3 - Resultado da bateria de testes número 2

RESULTADO DOS TESTES DE CLASSIFICAÇÃO (TESTE 2)			
Espécie	**Número de amostras de teste**	**Número de classificações corretas**	**Percentual de acerto**
Acer buergerianum Miq.	14	14	100,00%
Acer Palmatum	14	13	92,86%
Aesculus chinensis	16	7	43,75%
Berberis anhweiensis Ahrendt	17	16	94,12%
Cedrus deodara (Roxb.) G. Don	20	20	100,00%
Cercis chinensis	18	17	94,44%
Chimonanthus praecox L.	13	9	69,23%
Cinnamomum camphora (L.) J. Presl	17	13	76,47%
Cinnamomum japonicum Sieb.	14	2	14,29%
Citrus reticulata Blanco	14	12	85,71%
Ginkgo biloba L.	16	14	87,50%
Ilex macrocarpa Oliv.	13	10	76,92%
Indigofera tinctoria L.	19	19	100,00%
Kalopanax septemlobus (Thunb. ex A.Murr.) Koidz.	13	13	100,00%
Koelreuteria paniculata Laxm.	15	12	80,00%
Lagerstroemia indica (L.) Pers.	16	16	100,00%
Ligustrum lucidum Ait. f.	14	7	50,00%
Liriodendron chinense (Hemsl.) Sarg.	14	12	85,71%
Magnolia grandiflora L.	15	11	73,33%
Mahonia bealei (Fortune) Carr.	14	10	71,43%
Manglietia fordiana Oliv.	13	5	38,46%
Nerium oleander L.	17	16	94,12%
Osmanthus fragrans Lour.	14	4	28,57%
Phoebe nanmu (Oliv.) Gamble	16	14	87,50%
Phyllostachys edulis (Carr.) Houz.	15	4	26,67%
Pittosporum tobira (Thunb.) Ait. f.	16	14	87,50%
Podocarpus macrophyllus (Thunb.) Sweet	15	15	100,00%
Populus - canadensis Moench	16	16	100,00%
Prunus persica (L.) Batsch	14	9	64,29%
Prunus serrulata Lindl. var. lannesiana auct.	14	10	71,43%
Tonna sinensis M. Roem.	17	14	82,35%
Viburnum awabuki K.Koch	15	8	53,33%
	488	376	77,05%

É possível observar na Tabela 3 um aumento considerável na performance do processo de classificação, uma vez consideradas as características geométricas implementadas na API. As espécies destacadas na cor verde tiveram um aumento em suas taxas de acerto, partindo de

valores abaixo dos 50% e ultrapassando essa faixa. No trabalho de Kadir et al. (2011) um teste com características semelhantes apresentou uma precisão geral de 77,5% no processo de classificação, resultado praticamente semelhante ao obtido na bateria de testes efetuada na API.

O terceiro teste realizado, incorpora características baseadas em indicadores estatísticos da cor da folha. Neste teste, o parâmetro “Utilizar kurtosis dos valores RGB” foi configurado com o valor “True”, habilitando a utilização deste valor estatístico no descritor. O resultado da série de testes é exibido na Tabela 4.

Tabela 4 - Resultado da bateria de testes número 3

RESULTADO DOS TESTES DE CLASSIFICAÇÃO (TESTE 3)			
Espécie	**Número de amostras de teste**	**Número de classificações corretas**	**Percentual de acerto**
Acer buergerianum Miq.	14	14	100,00%
Acer Palmatum	14	14	100,00%
Aesculus chinensis	16	15	93,75%
Berberis anhweiensis Ahrendt	17	17	100,00%
Cedrus deodara (Roxb.) G. Don	20	20	100,00%
Cercis chinensis	18	18	100,00%
Chimonanthus praecox L.	13	12	92,31%
Cinnamomum camphora (L.) J. Presl	17	14	82,35%
Cinnamomum japonicum Sieb.	14	7	50,00%
Citrus reticulata Blanco	14	11	78,57%
Ginkgo biloba L.	16	15	93,75%
Ilex macrocarpa Oliv.	13	11	84,61%
Indigofera tinctoria L.	19	19	100,00%
Kalopanax septemlobus (Thunb. ex A.Murr.) Koidz.	13	13	100,00%
Koelreuteria paniculata Laxm.	15	15	100,00%
Lagerstroemia indica (L.) Pers.	16	16	100,00%
Ligustrum lucidum Ait. f.	14	12	85,71%
Liriodendron chinense (Hemsl.) Sarg.	14	12	85,71%
Magnolia grandiflora L.	15	14	93,33%
Mahonia bealei (Fortune) Carr.	14	14	100,00%
Manglietia fordiana Oliv.	13	13	100,00%
Nerium oleander L.	17	16	94,12%
Osmanthus fragrans Lour.	14	12	85,71%
Phoebe nanmu (Oliv.) Gamble	16	16	100,00%
Phyllostachys edulis (Carr.) Houz.	15	15	100,00%
Pittosporum tobira (Thunb.) Ait. f.	16	15	93,75%
Podocarpus macrophyllus (Thunb.) Sweet	15	15	100,00%
Populus - canadensis Moench	16	15	93,75%
Prunus persica (L.) Batsch	14	13	92,86%
Prunus serrulata Lindl. var. lannesiana auct.	14	12	85,71%
Tonna sinensis M. Roem.	17	15	88,23%
Viburnum awabuki K.Koch	15	15	100,00%
	488	455	93,24%

Como é possível observar na Tabela 4, as características baseadas em estatísticas de cor contribuem de forma significante para o aumento da precisão do processo de classificação, como evidenciado nas espécies destacadas na cor verde. Um grande avanço, em termos de resultado, em relação ao teste anterior é o fato da não ocorrência de espécies com desempenho ruim na classificação (abaixo de 50%). Kadir et al. (2011) apresentam, em seus testes utilizando PFT, características geométricas e estatísticas de cor, uma taxa de acerto de 87,81%.

O quarto teste utiliza o parâmetro “Utilizar kurtosis dos valores RGB” com o valor “False”, desabilitando o uso do valor estatístico *kurtosis* nos descritores de cor. O motivo da remoção deste valor descritor é o fato de que este valor estava reduzindo a precisão de classificação das amostras, no trabalho de Kadir et al. (2011). O resultado dos testes efetuados na API são apresentados na Tabela 5.

Tabela 5 - Resultado da bateria de testes número 4

RESULTADO DOS TESTES DE CLASSIFICAÇÃO (TESTE 4)			
Espécie	**Número de amostras de teste**	**Número de classificações corretas**	**Percentual de acerto**
Acer buergerianum Miq.	14	14	100,00%
Acer Palmatum	14	14	100,00%
Aesculus chinensis	16	15	93,75%
Berberis anhweiensis Ahrendt	17	17	100,00%
Cedrus deodara (Roxb.) G. Don	20	20	100,00%
Cercis chinensis	18	18	100,00%
Chimonanthus praecox L.	13	12	92,31%
Cinnamomum camphora (L.) J. Presl	17	14	82,35%
Cinnamomum japonicum Sieb.	14	7	50,00%
Citrus reticulata Blanco	14	11	78,57%
Ginkgo biloba L.	16	15	93,75%
Ilex macrocarpa Oliv.	13	11	84,61%
Indigofera tinctoria L.	19	19	100,00%
Kalopanax septemlobus (Thunb. ex A.Murr.) Koidz.	13	13	100,00%
Koelreuteria paniculata Laxm.	15	15	100,00%
Lagerstroemia indica (L.) Pers.	16	16	100,00%
Ligustrum lucidum Ait. f.	14	12	85,71%
Liriodendron chinense (Hemsl.) Sarg.	14	12	85,71%
Magnolia grandiflora L.	15	14	93,33%
Mahonia bealei (Fortune) Carr.	14	14	100,00%
Manglietia fordiana Oliv.	13	13	100,00%
Nerium oleander L.	17	16	94,12%
Osmanthus fragrans Lour.	14	12	85,71%
Phoebe nanmu (Oliv.) Gamble	16	16	100,00%
Phyllostachys edulis (Carr.) Houz.	15	15	100,00%
Pittosporum tobira (Thunb.) Ait. f.	16	15	93,75%
Podocarpus macrophyllus (Thunb.) Sweet	15	15	100,00%
Populus - canadensis Moench	16	15	93,75%
Prunus persica (L.) Batsch	14	13	92,86%
Prunus serrulata Lindl. var. lannesiana auct.	14	12	85,71%
Tonna sinensis M. Roem.	17	15	88,23%
Viburnum awabuki K.Koch	15	15	100,00%
	488	455	93,24%

É possível observar que na Tabela 5 os resultados são semelhantes aos do teste anterior, que usava o valor da *kurtosis*. Porém, nos testes efetuados por Kadir et al. (2011), a

ausência do valor da *kurtosis* fez com que a precisão do mecanismo de classificação avançasse para a taxa de acerto de 88,75%. Devido ao resultado desta etapa destacar que o valor da *kurtosis* não agrega uma melhora na performance do processo de classificação, dada a combinação de características utilizadas, ela manteve-se desabilitada nos testes subsequentes com a API.

A quinta série de testes incorpora um novo elemento à combinação de características utilizadas anteriormente: a máscara de nervura. O resultado proveniente desta etapa de teste é exibido na Tabela 6.

Tabela 6 - Resultado da bateria de testes número 5

RESULTADO DOS TESTES DE CLASSIFICAÇÃO (TESTE 5)			
Espécie	**Número de amostras de teste**	**Número de classificações corretas**	**Percentual de acerto**
Acer buergerianum Miq.	14	14	100,00%
Acer Palmatum	14	13	92,86%
Aesculus chinensis	16	15	93,75%
Berberis anhweiensis Ahrendt	17	17	100,00%
Cedrus deodara (Roxb.) G. Don	20	20	100,00%
Cercis chinensis	18	18	100,00%
Chimonanthus praecox L.	13	13	100,00%
Cinnamomum camphora (L.) J. Presl	17	14	82,35%
Cinnamomum japonicum Sieb.	14	10	71,43%
Citrus reticulata Blanco	14	11	78,57%
Ginkgo biloba L.	16	15	93,75%
Ilex macrocarpa Oliv.	13	12	92,31%
Indigofera tinctoria L.	19	19	100,00%
Kalopanax septemlobus (Thunb. ex A.Murr.) Koidz.	13	13	100,00%
Koelreuteria paniculata Laxm.	15	14	93,33%
Lagerstroemia indica (L.) Pers.	16	16	100,00%
Ligustrum lucidum Ait. f.	14	14	100,00%
Liriodendron chinense (Hemsl.) Sarg.	14	14	100,00%
Magnolia grandiflora L.	15	14	93,33%
Mahonia bealei (Fortune) Carr.	14	13	92,86%
Manglietia fordiana Oliv.	13	13	100,00%
Nerium oleander L.	17	17	100,00%
Osmanthus fragrans Lour.	14	12	85,71%
Phoebe nanmu (Oliv.) Gamble	16	16	100,00%
Phyllostachys edulis (Carr.) Houz.	15	14	93,33%
Pittosporum tobira (Thunb.) Ait. f.	16	15	93,75%
Podocarpus macrophyllus (Thunb.) Sweet	15	15	100,00%
Populus - canadensis Moench	16	15	93,75%
Prunus persica (L.) Batsch	14	13	92,86%
Prunus serrulata Lindl. var. lannesiana auct.	14	13	92,86%
Tonna sinensis M. Roem.	17	15	88,23%
Viburnum awabuki K.Koch	15	15	100,00%
	488	462	94,67%

O resultado geral (destacado na cor amarela) obtido neste teste acabou representando a taxa de acerto mais alta dentre as séries registradas no processo de validação da API, alcançando o valor de 94,67% de sucesso nos processos de classificação. Pode-se observar um ganho considerável nos resultados da classificação da espécie *Cinnamomum japonicum Sieb.* (destacada na cor verde), saltando de uma taxa de acerto de 50% para 71,43%.

A série de testes final engloba todas as características disponibilizadas pela API (exceto a *kurtosis*, pelas razões já descritas nos testes anteriores), incorporando ao conjunto de

características presentes no teste anterior mais uma característica de textura: a lacunaridade. O resultado desta bateria de testes é exibido na Tabela 7.

Tabela 7 - Resultado da bateria de testes número 6

RESULTADO DOS TESTES DE CLASSIFICAÇÃO (TESTE 6)			
Espécie	**Número de amostras de teste**	**Número de classificações corretas**	**Percentual de acerto**
Acer buergerianum Miq.	14	14	100,00%
Acer Palmatum	14	13	92,86%
Aesculus chinensis	16	15	93,75%
Berberis anhweiensis Ahrendt	17	16	94,12%
Cedrus deodara (Roxb.) G. Don	20	20	100,00%
Cercis chinensis	18	18	100,00%
Chimonanthus praecox L.	13	13	100,00%
Cinnamomum camphora (L.) J. Presl	17	14	82,35%
Cinnamomum japonicum Sieb.	14	11	78,57%
Citrus reticulata Blanco	14	12	85,71%
Ginkgo biloba L.	16	15	93,75%
Ilex macrocarpa Oliv.	13	12	92,31%
Indigofera tinctoria L.	19	19	100,00%
Kalopanax septemlobus (Thunb. ex A.Murr.) Koidz.	13	13	100,00%
Koelreuteria paniculata Laxm.	15	14	93,33%
Lagerstroemia indica (L.) Pers.	16	16	100,00%
Ligustrum lucidum Ait. f.	14	14	100,00%
Liriodendron chinense (Hemsl.) Sarg.	14	14	100,00%
Magnolia grandiflora L.	15	13	86,67%
Mahonia bealei (Fortune) Carr.	14	13	92,86%
Manglietia fordiana Oliv.	13	13	100,00%
Nerium oleander L.	17	17	100,00%
Osmanthus fragrans Lour.	14	12	85,71%
Phoebe nanmu (Oliv.) Gamble	16	15	93,75%
Phyllostachys edulis (Carr.) Houz.	15	14	93,33%
Pittosporum tobira (Thunb.) Ait. f.	16	15	93,75%
Podocarpus macrophyllus (Thunb.) Sweet	15	15	100%
Populus - canadensis Moench	16	14	87,50%
Prunus persica (L.) Batsch	14	13	92,86%
Prunus serrulata Lindl. var. lannesiana auct.	14	11	78,57%
Tonna sinensis M. Roem.	17	16	94,12%
Viburnum awabuki K.Koch	15	15	100,00%
	488	459	94,06%

O resultado do teste contendo todas as características, apesar de possuir uma performance geral (destacada em azul) inferior à série anterior, fez com que a taxa mínima de precisão registrada alcançasse o valor de 78,57% (tais ocorrências estão destacadas na cor verde). Um fator negativo observado foi a queda significativa na performance da classificação da espécie *Prunus serrulata Lindl. var. lannesiana auct*. O resultado obtido por Kadir et al.

(2011) para um conjunto similar de características reporta uma taxa de acerto de 93,75% no processo de classificação, semelhante ao resultado obtido neste projeto.

O Quadro 44 resume as características e resultados finais dos trabalhos correlatos ao lado dos obtidos por este projeto.

características / trabalhos relacionados	Wu et al. (2007)	Singh, Gupta e Gupta (2010)	Kadir et al. (2011)	Plantarum (2012)
mecanismo de classificação	RNP	SVM-BDT	RNP	RNP
espécies diferentes	32 (Flavia)	32 (Flavia)	32 (Flavia)	32 (Flavia)
características utilizadas	geométricas, nervuras	geométricas, nervuras	geométricas, nervuras, descritores de Fourier, cor, textura	geométricas, nervuras, descritores de Fourier, cor, textura
invariância geométrica	translação, escala, rotação	translação, escala, rotação	translação, escala, rotação	translação, escala, rotação
número de descritores	5	5	62	38
precisão média	90,3%	96%	93,75%	94,06%

Quadro 44 – Comparativo dos trabalhos correlatos com este projeto

A proposta apresentada por Chaki e Parekh (2012) é omitida do Quadro 44, pois, apesar de apresentar uma alta taxa de acerto, a grande diferença (em relação aos demais projetos listados) na quantidade de amostras utilizadas na validação do projeto compromete a imparcialidade de uma comparação de resultados.

Um fator a ser destacado nas implementações de Wu et al. (2007) e Singh, Gupta e Gupta (2010) é a utilização da técnica *Principal Component Analysis* (PCA) para ortogonalizar as características iniciais e selecionar as 5 combinações mais relevantes. Esta técnica ajuda a reduzir o número de descritores passados como vetor de entrada para a RNP e SVM-BDT, aumentando a performance (em termos de processamento) da rede neural e consumindo menos recursos de armazenamento.

Quando compara-se os resultados apresentados por Wu et al. (2007) com os obtidos neste projeto, podemos constatar uma ligeira melhora na taxa de acerto do processo de classificação. Apesar de ambos os trabalhos utilizarem o mesmo tipo de classificador, tal comportamento é possilvelmente proveniente da utilização de mais características de naturezas distintas (baseadas em cor, textura e contorno).

Em comparação ao trabalho apresentado por Kadir et al. (2011), este trabalho apresenta praticamente a mesma taxa de acerto no processo de classificação, porém, utilizando somente uma quantidade de descritores equivalente à cerca de 60% do conjunto utilizado por Kadir et al. (2011), contribuindo para uma melhor performance (em termos de

processamento) da rede neural e consumindo menos recursos de armazenamento.

O classificador utilizado por Singh, Gupta e Gupta (2010) apresenta uma taxa geral de acerto alta, caracterizando uma performance superior à este trabalho, porém, algumas espécies de plantas apresentaram uma taxa de acerto insatisfatória no processo de classificação, como visto na Tabela 8.

Tabela 8 - Resultados obtidos por Singh, Gupta e Gupta (2010)

RESULTADOS DA CLASSIFICAÇÃO	
Espécie	**Percentual de acerto**
Acer buergerianum Miq.	100,00%
Acer Palmatum	80,00%
Aesculus chinensis	100,00%
Berberis anhweiensis Ahrendt	100,00%
Cedrus deodara (Roxb.) G. Don	100,00%
Cercis chinensis	100,00%
Chimonanthus praecox L.	90,00%
Cinnamomum camphora (L.) J. Presl	100,00%
Cinnamomum japonicum Sieb.	80,00%
Citrus reticulata Blanco	100,00%
Ginkgo biloba L.	100,00%
Ilex macrocarpa Oliv.	90,00%
Indigofera tinctoria L.	100,00%
Kalopanax septemlobus (Thunb. ex A.Murr.) Koidz.	90,00%
Koelreuteria paniculata Laxm.	80,00%
Lagerstroemia indica (L.) Pers.	80,00%
Ligustrum lucidum Ait. f.	80,00%
Liriodendron chinense (Hemsl.) Sarg.	80,00%
Magnolia grandiflora L.	90,00%
Mahonia bealei (Fortune) Carr.	100,00%
Manglietia fordiana Oliv.	90,00%
Nerium oleander L.	100,00%
Osmanthus fragrans Lour.	80,00%
Phoebe nanmu (Oliv.) Gamble	100,00%
Phyllostachys edulis (Carr.) Houz.	100,00%
Pittosporum tobira (Thunb.) Ait. f.	0,00%
Podocarpus macrophyllus (Thunb.) Sweet	90,00%
Populus - canadensis Moench	0,00%
Prunus persica (L.) Batsch	40,00%
Prunus serrulata Lindl. var. lannesiana auct.	90,00%
Tonna sinensis M. Roem.	80,00%
Viburnum awabuki K.Koch	0,00%

Fonte: adaptado de Singh, Gupta e Gupta (2010).

Como observado na Tabela 8, são apresentadas 4 espécies com taxas de acerto muito baixas (abaixo de 50%), possivelmente indicando que o classificador utilizado não se apresenta totalmente estável (uma vez que os descritores utilizados são semelhantes ao apresentado por Wu et al. (2007) em seu trabalho).

4 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de uma API para reconhecimento de plantas a partir de imagens de folhas, onde os objetivos principais foram a segmentação da área correspondente ao órgão foliar, extração de características discriminantes das amostras e classificação em uma das espécies previamente registradas na base de dados da API.

A API foi implementada utilizando a linguagem de programação C/C++, fazendo uso das bibliotecas OpenCV e SQLite, auxiliando nas rotinas de processamento de imagens e gerenciamento da base de dados, respectivamente. O projeto de demonstração da API foi implementado utilizando a linguagem C# da plataforma .NET, utilizando-se dos componentes disponibilizados pelo pacote do Windows Forms.

Os resultados obtidos a partir das séries de testes efetuadas com a API mostraram-se satisfatórios, mantendo-se no mesmo patamar dos trabalhos correlatos relacionados, cuja publicação pode ser considerada relativamente recente. Em especial, a taxa de acerto na classificação apresentou-se satisfatória para todas as espécies utilizadas no processo de validação da API, apesar da escolha por simplificar algumas das características descritas nos trabalhos correlatos.

A preferência pela construção da API no formato de uma DLL mostrou-se vantajosa pelo fato da simples integração com outras plataformas: além da aplicação de demonstração desenvolvida na plataforma .NET, durante o estágio de desenvolvimento do projeto outra aplicação, desenvolvida em C++ e utilizada nos testes de execução das características, também integrou facilmente com a API. Tal formato permite, inclusive, a utilização deste projeto em aplicações web, vinculado ao *backend* do aplicativo porém podendo ser consumido por vários usuários, ao longo da internet.

A API desenvolvida pode ser utilizada como base para a implementação de sistemas que auxiliem no reconhecimento e classificação de espécies baseados no registro de características das folhas, sem a necessidade do porte de catálogos impressos ou conhecimento prévio das características discriminantes da espécie.

4.1 LIMITAÇÕES

A API desenvolvida apresenta as seguintes limitações:

a) operado apenas na plataforma Windows;
b) trabalha com o reconhecimento somente de imagens de folhas com alto contraste (folhas com coloração esverdeada disposta em um fundo branco, por exemplo) e devidamente dispostas na imagem (sem galhos, dobradas ou parcialmente omitidas);
c) o processo de classificação não fornece um coeficiente indicando a certeza do resultado fornecido;
d) o resultado fornecido pelo processo de classificação é composto de apenas 1 espécie identificada como similar;
e) o classificador utilizado utiliza o mesmo peso para todos os descritores utilizados, isso faz com que características com maior número de descritores tenham uma influência maior no processo de decisão;
f) é necessária a intervenção do usuário, para o informe dos pontos correspondentes ao pecíolo e à ponta da folha;
g) alguns caminhos de arquivos, devido à funções de bibliotecas utilizadas internamente pela API, são restritos ao *charset* ASCII.

4.2 EXTENSÕES

Algumas sugestões de extensões deste trabalho são listadas a seguir:

a) detecção automática dos pontos correspondentes ao pecíolo e à ponta da folha, eliminando a necessidade da intervenção do usuário no processo de classificação das amostras;
b) disponibilizar, através do classificador, o coeficiente de certeza da decisão efetuada sobre a espécie atribuída à amostra de teste, permitindo o usuário analisar este valor e decidir a confiabilidade do resultado;
c) permitir o processo de classificação retornar uma lista de espécies candidatas à que a amostra classificada pertence e seu coeficiente de certeza, para que, mesmo em

caso de erro de classificação, seja possível o usuário identificar na lista de resultados a espécie correta;

d) implementação de pesos no mecanismo de classificação, permitindo definir características com graus de relevância distintos no processo de decisão da espécie;
e) utilização de PCA para reduzir o número de valores descritores utilizados pela API, aumentando a performance do classificador e diminuindo o espaço necessário para armazenamento desses valores;
f) utilização da *Fast Fourier Transform* (FFT) para suplantar a DFT utilizada no projeto, tornando a computação dos descritores de Fourier mais rápida;
g) implementação de novas características como, por exemplo, um indicador se a amostra está disposta de forma alternada ou oposta (em relação às outras folhas no galho) e a extração das dimensões reais da amostra, a partir do reconhecimento de um gabarito na imagem. Isso visa, à partir da inclusão de novos fatores discriminantes, melhorar a taxa de acerto do processo de classificação;
h) implementação de novos classificadores como, por exemplo, *Support Vector Machines* (SVM) ou uma Rede Neural Perceptron Multicamadas;
i) portabilizar a API para outras plataformas.

REFERÊNCIAS BIBLIOGRÁFICAS

BROWN, Stan. **Measures of Shape: Skewness and Kurtosis / MATH200 (TC3, Brown)**. [S.l], nov. 2012. Disponível em: <http://www.tc3.edu/instruct/sbrown/stat/shape.htm>. Acesso em: 19 dez. 2012.

CHAKI, Jyotismita; PAREKH, Ranjan. Designing an automated system for plant leaf recognition. **International Journal of Advances in Engineering & Technology**, [S.l.], v. 2, n. 1, p. 149-158, Jan. 2012.

COSTA, Luciano F.; CESAR JR., Roberto M. **Shape analysis and classification**: theory and practice. 2nd ed. Boca Raton: CRC Press, 2009.

GONZALES, Rafael C.; WOODS, Richard E. **Digital image processing**. 3rd ed. Upper Saddle River: Pearson Prentice Hall, 2008.

KADIR, Abdul et al. Leaf classification using shape, color, and texture features. **International Journal of Computer Trends and Technology**, [S.l], v. 1, n. 3, p. 225-230, July/Aug. 2011.

KARPERIEN, Audrey. **What is lacunarity?**. [S.l], out. 2010. Disponível em: <http://rsbweb.nih.gov/ij/plugins/fraclac/FLHelp/Lacunarity.htm>. Acesso em: 19 dez. 2012.

PLOTZE, Rodrigo O. **Identificação de espécies vegetais através da análise da forma interna de órgãos foliares**. 2004. 152 f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Paulo.

RUSS, John C. **The image processing handbook**. 6th ed. Boca Raton: CRC Press, 2011.

SINGH, Krishna; GUPTA, Indra; GUPTA, Sangeeta. SVM-BDT PNN and Fourier moment technique for classification of leaf shape. **International Journal of Signal Processing, Image Processing and Pattern Recognition**, [S.l], v. 3, n. 4, p. 67-78, Dec. 2010.

SOUZA, Luiz A. **Morfologia e anatomia vegetal**: célula, tecidos, orgãos e plântula. Ponta Grossa: Ed. da Universidade Estadual de Ponta Grossa, 2003.

WU, Stephen G. et al. A leaf recognition algorithm for plant classification using probabilistic neural network. In: IEEE INTERNATIONAL SYMPOSIUM ON SIGNAL PROCESSING AND INFORMATION TECHNOLOGY, 7th, 2007, Giza. **Proceedings…** [S.l]: IEEE, 2008. p. 11-16.

ZHANG, Dengsheng; LU, Guojun. Study and evaluation of different Fourier methods for image retrieval. **Image and Vision Computing**, [S.l], v. 23, n. 1, p. 33-49, Jan. 2005.