Universidade Estadual de Campinas - UNICAMP
Faculdade de Engenharia Elétrica e de Computação

Ambiente de Suporte ao Ensino de Processamento de Imagens Usando a Linguagem Python

Autor: **Alexandre Gonçalves Silva**
Orientador: **Prof. Dr. Roberto de Alencar Lotufo**

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: **Engenharia de Computação**.

Banca Examinadora

Prof. Dr. Arnaldo de Albuquerque Araújo — ICEx/DCC/UFMG
Prof. Dr. Clésio Luiz Tozzi — DCA/FEEC/UNICAMP

Fevereiro de 2003
Campinas, SP - Brasil

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Si38a

Silva, Alexandre Gonçalves
Ambiente de suporte ao ensino de processamento de imagens usando a linguagem Python / Alexandre Gonçalves Silva.--Campinas, SP: [s.n.], 2003.

Orientador: Roberto de Alencar Lotufo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Processamento de imagens. 2. Educação. 3. Engenharia – Estudo e ensino. 4. Software – Desenvolvimento. 5. Software livre. I. Lotufo, Roberto de Alencar. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Este trabalho consiste no estudo, desenvolvimento e implementação de uma caixa de ferramentas de processamento de imagens usando a linguagem *Python* de programação e o pacote *Numerical Python*. A caixa de ferramentas foi desenvolvida dentro do projeto *Adesso*. Este é uma coleção de ferramentas baseada na tecnologia *XML* que auxilia a programação, a escrita de documentação e a inteface de bibliotecas *C* para plataformas comumente utilizadas. No *Adesso*, havia originalmente suporte às linguagens de programação *MATLAB* e *Tcl/Tk*. Este trabalho o estende de modo que suporte a linguagem *Python*. A caixa de ferramentas de processamento de imagens é disponibilizada como "código aberto", segue a mesma licença de distribuição da linguagem *Python* e do pacote *Numerical*, e é útil ao ensino, pesquisa e desenvolvimento de aplicações finais.

Abstract

This work consists in the study, development and implementation of a toolbox for image processing written in the *Python* programming language and in the *Numerical Python* package. The toolbox was developed using the *Adesso* project. *Adesso* is a collection of tools based on the *XML* technology, which helps the programming, the writing of the documentation and the interfacing of *C* libraries to standard platforms. *Adesso* originally supported *MATLAB* and *Tcl/Tk* programming languages. This work has extended *Adesso* to support the *Python* language. The image processing toolbox is available as "open source", following the same distribution license of the *Python* language and the *Numerical* package. The toolbox is useful for education, research and development of final applications.

À minha avó que, enquanto presente,
nos fazia vibrar com sua mais profunda serenidade.

Agradecimentos

Não poderia deixar de mencionar aqui a figura de meus pais: Olenir e Elenir. A semelhança dos nomes não é maior que a do cuidado que ambos têm para com os filhos. Mesmo correndo o risco de me tornar repetitivo não me canso de lhes agradecer muito. Mais que um sólido alicerce, proporcionaram-me oportunidades! Sem as mesmas, fico imaginando... O que seria então?

Gostaria de externar minha admiração pelo modo como se procedeu a orientação a este trabalho. Agradeço ao Prof. Lotufo por sua competência em trilhar o caminho do bom-senso, criatividade quase que contagiante, capacidade em se desvencilhar dos problemas, e dedicação sempre prontamente dispensada.

Agradeço ao Rubens do CenPRA por solucionar sempre nossas dúvidas quando o assunto era: "Ambiente para Desenvolvimento de Software Científico" (Adesso). Por ser nossa segurança em muitos pontos críticos.

Agradeço também aos meus familiares, aos amigos da pequena grande Apucarana, meu irmão de sangue Giuliano, meus irmãos de pesquisa - André, Francisco, Guilherme, Jane, Marco, Ricardo, Romaric e Wellington - por toda força prestada nesta caminhada. Aos camaradas - Alex, Carlos, Franklin, Ivana, Luiz, Maurício e Rangel - por todas as divertidas "intrigas" que deixaram minha vida mais branda nas horas difíceis.

À minha queria Letícia que, mesmo relutante, acabou por me permitir incluir seu nome aqui. Agradeço suas doces palavras que me confortaram e me fizeram forte antes da defesa...

Agradeço à FAPESP e à FAEP pelo incentivo à pesquisa e, em especial, pelo inestimável suporte a esta realização.

Enfim, agradeço a todos que, de alguma forma, participaram deste trabalho. Uma idéia, um palpite por mais despretensiosos que parecessem, sem dúvida, foram decisivos em particularidades do projeto e até mesmo em capítulos completos deste texto.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

> *"...Costumava se distrair realizando pequenos consertos domésticos: uma bóia de descarga, a bucha de uma torneira, um fusível queimado. Dispunha para isso da necessária habilidade e de uma preciosa caixa de ferramentas em que ninguém mais podia tocar. Aprendi com ele como é indispensável, para a boa ordem da casa, ter à mão pelo menos um alicate e uma chave de fenda..."*
>
> *Como Dizia Meu Pai*, Fernando Sabino

Com a ampliação de formas de aquisição de imagens, poder crescente de processamento dos computadores e aprimoramento de ferramentas computacionais, o processamento digital de imagens se configura entre as principais estratégias para tratar certos problemas em áreas como medicina, biologia, astronomia, automação industrial, engenharia, geologia, agronomia, artes, entre outras. Para imagens médicas, por exemplo, além do tradicional raio-X, há também, hoje em dia, a tomografia computadorizada e a ressonância

magnética que geram imagens com dados e resoluções diversas. O tipo de informação potencial registrada em uma imagem é importante e pode determinar se uma aplicação será ou não bem sucedida. Ainda na área médica, como exemplo de aplicação, pode-se citar a caracterização de tumores. Neste caso, uma ferramenta mais objetiva (desprovida totalmente ou parcialmente de interferência humana) pode auxiliar o médico em seu diagnóstico.

O foco do presente trabalho está na escolha de plataformas de desenvolvimento de *software* científico consideradas promissoras, definição e implementação de um conjunto de algoritmos tradicionais de processamento de imagens, e organização sistemática de toda esta informação de forma a minimizar o custo de manutenção e criação de programas e documentação. Auxiliar o aprendizado dos assuntos referentes à disciplina de "Processamento de Imagens" é o principal resultado, mas o ambiente construído também é apto a ser utilizado em várias instâncias de pesquisa e aplicação.

1.1 Ferramentas Existentes

Existem diversas ferramentas computacionais apropriadas ao processamento de imagens, tanto comerciais como gratuitas. Podemos citar o Khoros, MATLAB [9], Mathematica, VisiLog, PV-WAVE, AVS, IDL, ImageMagik, e o esforço nacional PhotoPixJ [1], entre outras [23].

O DCA-FEEC-Unicamp adotou, no período de 1991 a 1997, a plataforma Khoros para ensino das disciplinas de pós-graduação, para pesquisa e aplicações. O Khoros é um ambiente completo de processamento de imagens consistindo em uma interface de programação visual bastante atrativa para os iniciantes na área. Suas principais vantagens eram o fato de ser gratuito, ter o código fonte disponível, oferecer um sistema de ajuda de autoria de *software*, permitindo sua extensão para novas funções dentro do contexto de caixa de ferramentas (*toolboxes*). Sua adoção pela Unicamp teve uma grande influência em diversos grupos de pesquisa do Brasil e do exterior. No

DCA foram desenvolvidas três *toolboxes*: V3DTOOLS de processamento e visualização tridimensional [19], MMACH de Morfologia Matemática [4] e DIPCOURSE de ensino de processamento de imagens [15, 17]. Este último foi o primeiro curso de processamento de imagens disponível na Internet. Porém, o grupo de desenvolvimendo do Khoros não acompanhou o avanço tecnológico de *software* e, ao mesmo tempo, deixou de ser gratuito. Como conseqüência, o mesmo deixou de ser adotado no DCA como ferramenta preferencial, dando lugar ao MATLAB[1]. Para o MATLAB também já foram desenvolvidas algumas *toolboxes*, sendo uma delas repassada à indústria dentro do projeto Softex [27].

Baseado na experiência de uso de duas plataformas distintas, pode-se dizer que as mesmas apresentam características muito contrastantes, pois os pontos mais fortes de uma são os mais fracos de outra e vice-versa. Como desvantagem do Khoros pode-se citar pouca documentação, certa dificuldade de instalação, dificuldade de criação de novas funções, e de ser dedicado a problemas de processamento de imagens, apesar dos esforços do grupo em considerá-lo um ambiente genérico de desenvolvimento de *software*. Já o MATLAB é um ambiente genérico de computação científica muito difundido nos cursos de Engenharia, tem uma linguagem matricial extremamente simples, compacta, muito próxima da linguagem matemática e possui uma grande comunidade sendo freqüentemente utilizado na descrição de algoritmos de computação científica. Tem como maior desvantagem, entretanto, o seu alto custo. Poucas instituições têm condições de adquiri-lo. Uma outra desvantagem é a falta de ferramentas de autoria de caixas de ferramentas. Apesar de ser muito fácil criar um conjunto de funções voltadas a um problema, não existe nenhum suporte para a geração de documentação ou gerenciamento das implementações.

[1] `http://www.mathworks.com`

1.2 Motivação

É intenção do grupo desenvolver, a longo prazo, um ambiente computacional que contenha o melhor das plataformas Khoros e MATLAB. Neste sentido já existe o esforço associado ao projeto Adesso [21, 7], em cooperação com o CenPRA[2], que consiste de um ambiente de autoria de *software* de computação científica. O estágio atual do projeto, baseado na tecnologia XML [31] contempla o suporte à geração de documentação e de código de interface de bibliotecas computacionais. Havia até então o suporte à interface automática para as plataformas MATLAB e Tcl/Tk [34]. Este trabalho detalha implementações similares feitas com o intuito de se gerar suporte também para a linguagem Python e reforça, desta forma, a generalidade do sistema.

1.3 A Linguagem Python

Recentemente, a linguagem Python [24, 20] vem mostrando um crescimento muito grande, principalmente na comunidade acadêmica, como sendo uma linguagem interpretada bastante moderna, com conceitos de orientação a objetos e extensibilidade muito apurados. A linguagem foi concebida pelo holandês Guido van Rossum e seu desenvolvimento teve início em 1990, no CWI, em Amsterdã. O conjunto de entidades e pessoas ligadas à sua manutenção e aprimoramento, sobretudo via Internet, formam a Python Software Activity (PSA). Python é o nome popular dado a uma grande cobra qualquer, mas o nome da linguagem é inspirado no título do programa humorístico de TV "Monty Python's Flying Circus".

Python apresenta semântica dinâmica, um moderno mecanismo de tratamento de erros e exceções, flexibilidade no tratatamento de argumentos. Possui uma forma eficiente de acesso e reutilização de código com o uso dos

[2] *Centro de Pesquisas Renato Archer - Campinas - SP*

chamados "módulos", coleta de lixo automática, recursos avançados de manipulação de textos, listas e dicionários. Em Python, diferentemente de C++ ou Java, as funções são tratadas como objetos, característica de linguagens de programação funcional como Lisp[3]. Por ser programado em C, ser livremente distribuído e ter código aberto, Python oferece grande portabilidade.

Python, em sua versão 1.5.2, de 1999, integrante de várias distribuições Linux, já incluía mais de 140 módulos nativos. Como ilustração, pode-se destacar os módulos `os` e `sys` para funcionalidades de sistema, `cgi` para programação de páginas dinâmicas para Internet, `ftplib` para montagem de *scripts* para interação com servidores FTP, `gzip` para leitura e escrita de arquivos comprimidos, `math` para utilização de funções matemáticas, `re` para busca de texto com expressões regulares, `string` para operações com *strings*, `time` para obtenção de hora atual e conversão de formatos de data, `xmllib` para interpretação de arquivos em formato XML, `sockets`, manipulação de arquivos de entrada e saída. Inúmeros outros módulos, entre administração de sistemas, opções diversas de interfaces gráficas com usuário (GUI), *scripting* para Internet, programação de banco de dados, programação científica, inteligência artificial, gráficos, CORBA, podem ser encontrados a partir do site oficial[4], quase todos livres e gratuitos. Em especial, há um pacote de suporte à computação científica denominado Numerical[5] [3].

O Numerical também segue a linha "Open Source", tem parte de suas funções (críticas em desempenho) programadas em C, e estende o Python de forma simples e direta, implementando um eficiente modelo de matriz multidimensional. Os módulos do Numerical foram projetados sob forte influência da família de linguagens APL, Basis, MATLAB, FORTRAN, S e S+, entre outras. Desta forma, está se tornando muito atrativo o uso do Python, associado a este pacote numérico e ao ambiente Adesso de autoria de *software* científico. Acredita-se ser possível dar um grande passo no sentido de se obter

[3] `http://www.lisp.org`
[4] `http://www.python.org`
[5] `http://www.pfdubois.com/numpy`

um sistema que agregue as vantagens dos sistemas Khoros e MATLAB.

1.4 Objetivos

O objetivo principal deste trabalho foi o desenvolvimento de uma caixa de ferramentas para processamento de imagens que possa ser utilizada em ensino e pesquisa. Para atingir este objetivo, o sistema Adesso, que serviu de suporte para o desenvolvimento da caixa de ferramentas, foi estendido para suportar a linguagem Python, tanto na geração de código como na geração de documentação. Adicionalmente, foi desenvolvido um sistema de entrega via Internet, de relatórios e correção automática de programas feitos em Python para apoio ao uso da caixa de ferramentas em cursos de processamento de imagens. O objetivo final é ter um conjunto de ferramentas totalmente gratuito, de fácil instalação, multiplataforma e, sobretudo, de rápida aprendizagem e com programação simplificada. E ainda, baseado em Python, uma linguagem genérica bem projetada que oferece suporte nativo às mais diversas aplicações.

1.5 Estruturação da Dissertação

Este trabalho está organizado da seguinte forma. O Capítulo 2 ilustra o modelo de programação de algortimos de computação científica em linguagens de *scripting*. O Capítulo 3 descreve as ferramentas de desenvolvimento e a forma como se dá a produção de *software*. Neste capítulo, se encontra uma das três contribuições principais do presente trabalho: o desenvolvimento de folhas de estilos para que o sistema Adesso pudesse suportar a linguagem Python. O Capítulo 4 mostra os sistemas gerados para o suporte ao ensino de processamento de imagens e os principais resultados obtidos. Neste capítulo, se encontra a contribuição principal do trabalho: as diversas funções e demonstrações constituindo o pacote de auxílio ao ensino de

processamento de imagens. A outra contribuição, também neste capítulo é o sistema *on line* completo de entrega, correção, de estatística e gerenciamento de exercícios Python/C/C++ de alunos. Por fim, conclusões são apresentadas no Capítulo 5.

Capítulo 2

Modelo de Programação

Para cada linguagem de programação, é natural que haja uma série de vantagens e desvantagens [22]. Deve-se analisar os propósitos de uma dada aplicação para adoção de uma ferramenta em particular. Uma linguagem de sistema pode ser suficientemente eficiente conforme o grau de otimização que os programas sofrem na compilação. Considerando eficiência, a programação em linguagem *assembly* seria uma ótima opção mas demanda muito tempo e paciência na codificação. Por outro lado, programas extremamente simples, em poucas linhas e com eficiência computacional compatível ao assembly podem ser elaborados em linguagens de mais alto nível.

Uma ferramenta computacional genérica para processamento de imagens genérica usualmente oferece dois níveis de programação, associando facilidade de implementação e desempenho. Procura-se ter então uma linguagem interpretada de *scripting* servindo de interface a uma biblioteca de alto desempenho escrita em uma linguagem de mais baixo nível como C/C++ ou Fortran. A Figura 2.1 ilustra esta arquitetura. Porém, normalmente a demanda de ferramentas diversas de processamento de imagens é muito grande, sendo praticamente impossível oferecer um sistema completo ao usuário.

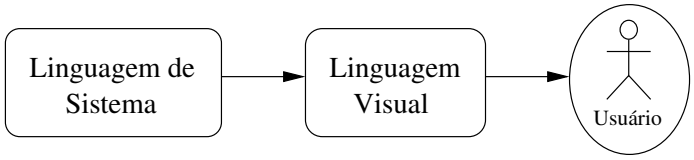


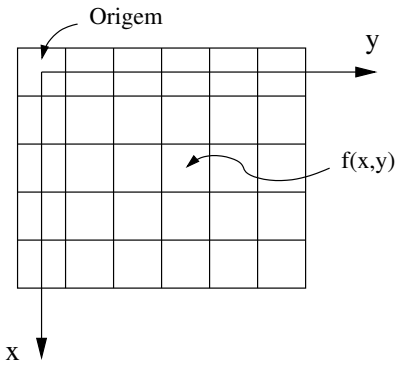
Figura 2.1: Níveis de programação de uma ferramenta usual de processamento de imagens.

2.1 Processamento de Imagens

As técnicas de processamento de imagens, desenvolvidas ao longo do tempo, apresentam duas facetas principais: uma de melhoramento da qualidade das informações, facilitando a observação humana; a outra de permitir a extração de dados, de forma automática ou semi-automática, por uma máquina. Uma das primeiras aplicações, nos anos 20, foi o melhoramento de imagens de jornal transmitidas via cabo submarino entre Londres e New York, além da introdução do sistema Bartlane que reduziu o tempo de transporte das imagens, através do oceano Atlântico, de mais de uma semana para menos de três horas [12]. Com o aprimoramento computacional (aquisição, armazenamento, capacidade de processamento, comunicação e exibição) e o desenvolvimento de novos algoritmos, o tratamento digital de imagens sobrevoa cada vez mais novas áreas da ciência, tornando-se, hoje em dia, muitas vezes, procedimento padrão e até mesmo transparente ao usuário.

2.1.1 Representação de Imagens

Uma imagem digital monocromática é uma função bidimensional de intensidade da luz $f(x, y)$, onde x e y denotam as coordenadas espaciais e o valor de f, em qualquer ponto (x, y), é proporcional ao brilho (ou nível de cinza) da imagem naquele ponto (*pixel*). A Figura 2.2 ilustra a representação de uma imagem. A Figura 2.2(a) mostra a convenção adotada, neste texto e na programação Python, para a direção e sentido dos eixos x e y. A Fi-

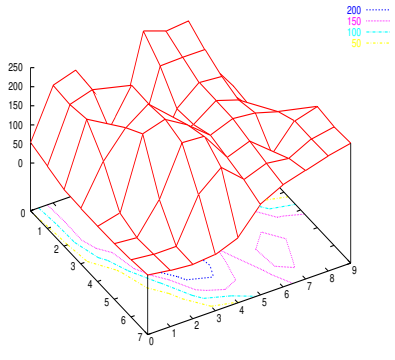


(a) Representação da função $f(x, y)$, origem, direção e sentido adotados para os eixos x e y da imagem.

(b) Exemplo de imagem digital em níveis de cinza (cada *pixel* tem intensidade entre 0 e 255).

54	183	202	112	20	242	221	227	91	14
38	179	197	102	13	197	199	198	69	7
23	187	140	234	24	131	179	174	39	13
20	123	190	232	117	136	178	159	48	20
20	33	221	238	202	155	177	171	154	137
21	15	94	223	219	143	165	110	176	193
25	18	18	133	184	131	153	145	175	186
31	22	17	22	45	112	143	159	177	189

(c) Ilustração dos valores dos níveis de cinza da região da orelha do câmera



(d) Representação do perfil topográfico da região da orelha do câmera.

Figura 2.2: Representação de imagens digitais em níveis de cinza.

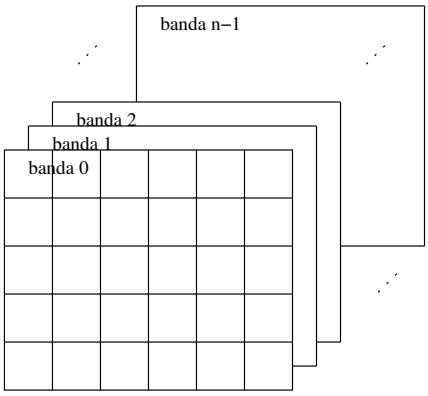
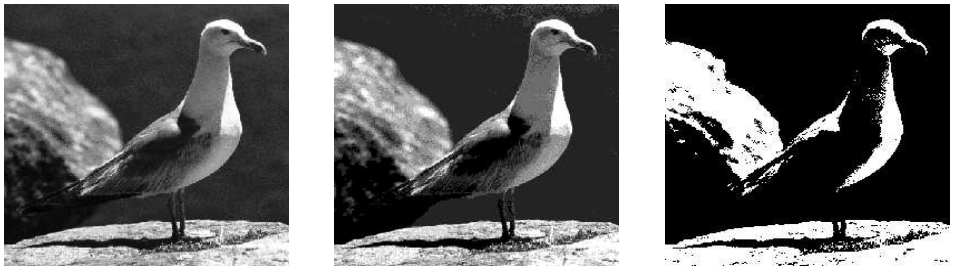


Figura 2.3: Representação de imagens digitais multibanda.

gura 2.2(b) exemplifica uma imagem digital com a indicação de seleção de uma pequena região da orelha do câmera. Cada *pixel* da imagem, neste exemplo, é representado por um inteiro sem sinal de 1 byte $(8 - bit)$, ou seja, cada *pixel* pode assumir um valor entre 0 e $2^8 - 1$. A Figura 2.2(c) ilustra os valores dos níveis de cinza da região selecionada e a Figura 2.2(d), o perfil topográfico, sendo os vértices do gráfico uma representação dos *pixels* da imagem nesta região.

Uma imagem multibanda consiste em se ter, para cada *pixel* (x, y), um vetor $(a_0, a_1, \ldots, a_{n-1})$, onde cada elemento a_k, $k \in [0, n-1]$, apresenta um nível de cinza. A Figura 2.3 ilustra esta representação. Uma imagem colorida é multibanda com $k \in [0, 2]$. Há diversos modelos para representar imagens coloridas, muitos deles baseado em luminância (associado ao brilho da luz), matiz (associado ao comprimento de onda dominante) e saturação (associado ao grau de pureza da matiz). Neste trabalho, as imagens coloridadas são armazenadas e exibidas no modelo RGB ou, de outra forma, $a_1 = R$ (banda vermelha), $a_2 = G$ (banda verde) e $a_3 = B$ (banda azul). A maioria das cores visíveis pode ser representada como uma combinação destas três cores primárias. Normalmente, são utilizados três inteiros sem sinal, um para cada banda, ou 3 *bytes* $(24 - bit)$ para representar cada *pixel*.



(a) Imagem original. (b) 8 intensidades. (c) 2 intensidades.

Figura 2.4: Imagem visualizada em diferentes mapas de cores.

Pode-se dizer que a representação de cada *pixel* por um inteiro sem sinal de $8-bit$, em geral, é superior ao total de níveis que a visão humana pode distinguir. Isto é verificado, por exemplo, na Figura 2.4. Apesar de um drástica redução dos 256 níveis de cinza da imagem da Figura 2.4(a) para apenas os 8 níveis distintos (0, 36, 72, 109, 145, 182, 218 e 255) da Figura 2.4(b), o resultado visual não apresenta grandes diferenças. A percepção humana, no entanto, é mais apurada quando se trata de imagens coloridas. Já na Figura 2.4(c), o resultado é crítico por se ter apenas dois níveis (0 e 255) para descrever a imagem original. Ainda neste exemplo, uma forma simples de compressão da imagem original seria a representação de cada *pixel* em $3 - bit$ para o primeiro caso e $1 - bit$ para o segundo.

Para efeito de programação, verifica-se que a representação de imagem se ajusta a uma representação comum de matriz bidimensional ou tridimensional de números, ou seja, com profundidade 1 em caso de imagens em níveis de cinza ou profundidade 3 (ou três matrizes justapostas) em caso de imagens coloridas.

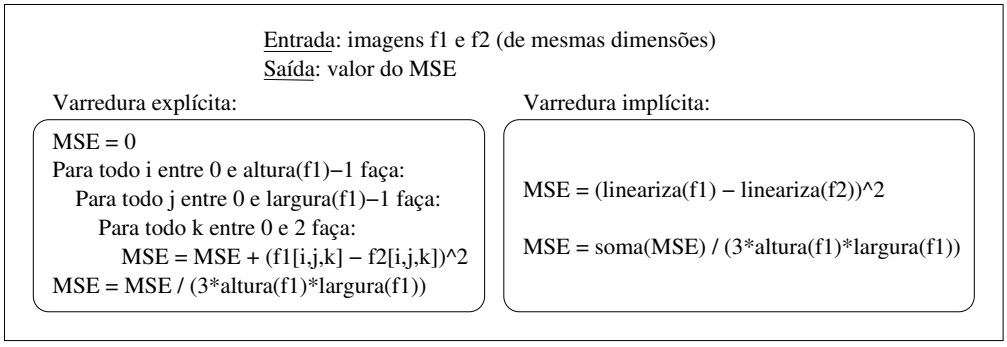


Figura 2.5: Algoritmos de varredura explícita e implícita para o cálculo do MSE.

2.2 Modelo de Programação

Linguagens interpretadas como Python ou MATLAB têm excelente suporte a operações matriciais (condição esperada em um sistema de processamento de imagens) e são apropriadas para construção de protótipos, pois nelas, em geral, o tempo de desenvolvimento de programas é consideravelmente mais curto. Porém, muitas vezes, não se ajustam a implementações que necessitam de iteração excessiva.

A Tabela 2.1 compara os tempos para o cálculo do MSE (Mean Square Error ou Erro Médio Quadrático), entre duas imagens coloridas (256x256 *pixels*), em diferentes linguagens. Os algoritmos são descritos na Figura 2.5 e as implementações podem ser vistas no Apêndice A. A equação adotada é

$$MSE = \frac{1}{3N_h N_w} \sum_{i=0}^{N_h-1} \sum_{j=0}^{N_w-1} \sum_{k=0}^{2} \left[(f_1(i,j,k) - f_2(i,j,k))^2 \right]$$

onde f_1 e f_2 podem ser consideradas imagens de entrada, no modelo RGB, de mesma dimensão N_hxN_wx3. Os testes notados por "explícita" ilustram a ineficiência em se usar laços iterativos de varredura explícita em linguagens interpretadas. Python implementa um código intermediário otimizado tor-

	Tempos
MATLAB (matricial)	$0,038s$
C	$0,051s$
Python (matricial)	$0,136s$
Java	$0,198s$
Python (explícita)	$4,473s$
MATLAB (explícita)	$14,628s$

Tabela 2.1: Comparação de tempos médios (Sun Ultra 60) para o cálculo do MSE entre duas imagens coloridas 256x256.

nando este tipo de implementação consideravelmente mais rápido em relação ao MATLAB. Java, dita semi-interpretada por alguns autores, pela produção de *bytecode* portável, normalmente utiliza um vetor de inteiros (4 *bytes*) na representação de imagens. Cada byte determina um canal (*Alpha*, *Red*, *Green*, *Blue*). A manipulação desta estrutura usando deslocamento de bits é bastante eficiente mas, ainda assim, o tempo é quase 4 vezes superior à implementação em C. Os testes notados por "matricial" se aproveitam da existência de operadores matriciais das linguagens de *scripting* em questão. Os operadores necessários na programação do MSE são basicamente: soma de todos elementos de uma matriz, subtração de matrizes (*pixel* a *pixel*) e quadrado de matrizes (*pixel* a *pixel*). Observa-se o ganho obtido no uso destas funcionalidades. Neste caso, Python supera Java e a implementação MATLAB é até mais eficiente que a adotada em C.

2.3 Desempenho

Esta seção introduz um modelo de programação para que se possa contornar o problema de desempenho em linguagens de *scripting* como Python e MATLAB, de forma mais direta, ou seja, sem a necessidade de construção de uma API (vide Subseção 3.2.4), sempre que houver a necessidade de implementação de um programa para manipulação de grandes conjuntos de dados (como uma imagem).

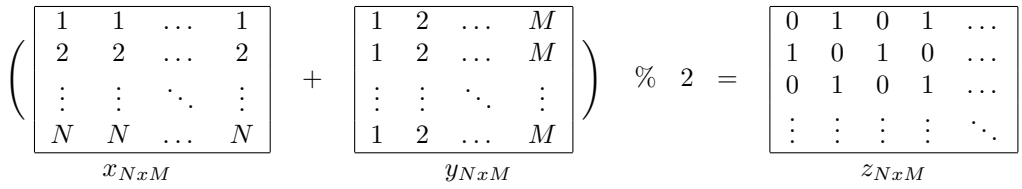


Figura 2.6: Exemplo para se gerar uma imagem em forma de xadrez por manipulação matricial.

Dada a facilidade de programação, é interessante que a maioria dos algoritmos solicitados em uma disciplina de processamento de imagens seja implementada diretamente em linguagens interpretadas de forma eficiente. Para isto, deve-se evitar, sobretudo, programas com iteração explícita. A idéia é proceder indiretamente toda iteração necessária, sempre que possível, através de manipulações matriciais. Um exercício inicial, neste sentido, poderia ser a geração de uma imagem 2-D, na qual os elementos justapostos se alternam entre 0 e 1 em ambas direções. Há diversos procedimentos para se elaborar esta imagem em forma de xadrez. Pensando em termos de operações matriciais pode-se propor a soma entre duas matrizes de malha de índices seguida pelo cálculo do resto da divisão inteira por dois:

$$z(i,j) \;=\; (x(i,j) \;+\; y(i,j)) \;\%\, 2$$

onde $x(i,j) = i+1$ e $y(i,j) = j+1$, para $\forall i \in [0, N-1]$ e $\forall j \in [0, M-1]$.

A Figura 2.6 ilustra esta operação. Observe que tanto o operador "+" (soma) como o operador "%" (resto da divisão inteira) devem estar presentes na linguagem e se encarregar de fazer o cálculo *pixel* a *pixel*. As matrizes x e y podem ser facilmente e eficientemente construídas (`Numeric.indices` em Python; `meshgrid` em MATLAB). Desta mesma forma, pode-se sintetizar uma imagem de círculo, logaritmo, cossenóide 2-D, entre outros exemplos.

Outra abordagem pode ser considerada quando há operações de vizinhança dada uma máscara de convolução. A seguir, tem-se a equação ado-

Máscara	Teste 1	Teste 2
3x3	$22,41s$	$0,05s$
5x5	$61,23s$	$0,13s$
7x7	$119,07s$	$0,26s$

Tabela 2.2: Tempos calculados (Pentium 4-1.5GHz) para duas implementações diferentes de um filtro de média sobre uma imagem de 256x256.

tada para convolução periódica bidimensional:

$$(f *_{(N,M)} h)(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} f(i,j)h(mod(x-i,N), mod(y-j,M))$$

onde $mod(x,N) = x - N\lfloor \frac{x}{N} \rfloor$, sendo as imagens f e h periódicas de dimensão NxM.

Neste caso, é mais eficiente criar uma imagem aumentada auxiliar e transladar f em todas as posições da máscara h, calculando a soma acumulada do produto destas translações por cada *pixel* da máscara. A Figura 2.7 ilustra este algoritmo. A Tabela 2.2 mostra tempos calculados para duas implementações, em Python, de um filtro por convolução. No Teste 1, é feita uma varredura *pixel* a *pixel* com a utilização de quatro laços iterativos (dois para a imagem e dois para a máscara). Já, no Teste 2, são feitas translações na imagem e são usados dois laços iterativos apenas para a máscara. Observa-se a diferença expressiva de tempo entre os métodos. Em se tratando do uso de linguagens interpretadas, é importante ter em mente este modelo de programação, no qual um operador, sempre que houver, possa ser aplicado a um grande conjunto de dados em uma única vez. Na verdade, este ganho em eficiência vem justamente do fato destes operadores serem implementados em linguagens de mais baixo nível.

Entrada: imagens f1 e f2
Saída: imagem g

```
rows  = altura(f), cols  = largura(f)
hrows = altura(h), hcols = largura(h)

dr1 = inteiro((hrows−1)/2), dr2 = hrows−dr1
dc1 = inteiro((hcols−1)/2), dc2 = hcols−dc1

// Criacao de uma matriz p aumentada:
p = concatena_vertical(
   concatena_vertical(f[{−dr2+1..fim},{0..fim}], f), f[{0..dr1},{0..fim}] )
p = concatena_horizontal(
   concatena_horizontal(p[{0..fim},{−dc2+1..fim}], p), p[{0..fim},{0..dc1}] )

g = cria_matriz_zeros(rows,cols)
Para todo r entre 0 e hrows−1:
   Para todo c entre 0 e hcols−1:
      g = g + h[hrows−r−1,hcols−c−1] * p[{r..rows+r},{c..cols+c}]
```

Figura 2.7: Algoritmo de convolução com redução de iterações.

Capítulo 3

Ambiente de Desenvolvimento

A computação científica se caracteriza pelo uso de computadores para investigar modelos matemáticos em ciências e engenharia, além de algumas das ciências sociais como economia. Sua importância vem sendo impulsionada pelo desenvolvimento de computadores modernos poderosos e ferramentas de alta performance, de preferência, fáceis de serem utilizadas e modificadas. É desejável também que a linguagem de desenvolvimento, apesar de fundamentalmente aplicada a computação científica, seja de propósito geral, agregando funcionalidades de rede, interface gráfica, entre outras, de forma a não impossibilitar a expansão do *software* produzido. Além disto, é importante que se tenha uma boa documentação publicada na Internet ou impressa para que a comunidade beneficiada primeiramente se identifique com o produto e, após isto, passe a divulgá-lo e aprimorá-lo através de contribuições. A documentação deve tipicamente conter um manual de referência das ferramentas, um tutorial e um conjunto de demonstrações. O manual de referência deve conter a descrição sintática do comando ou função, exemplos ilustrativos, equações matemáticas, referências bibliográficas e descrição do algoritmo utilizado. É imprescindível também haver correspondência da documentação com as novas versões do *software*.

O paradigma de níveis de programação da Figura 2.1 se aplica especial-

mente bem ao desenvolvimento de software científico onde a implementação de algoritmos quase sempre exige a utilização de linguagens de programação de sistemas. A confecção de aplicações, entretanto, pode ser feita a partir de linguagens de *scripting* ou programação visual com grandes ganhos de produtividade e facilidades de depuração. Python é uma plataforma que disponibiliza interfaces de programação bem definidas para a incorporação de extensões escritas principalmente em C/C++. A criação destas extensões é um processo mecânico que se torna muito trabalhoso à medida em que se tem muitos componentes. Este tipo de situação incentiva o uso da prática "cópia e cola" pois a maioria dos programas possuem um gabarito a ser seguido. É comum criar um novo componente a partir da cópia de um componente similar e fazer as substituições necessárias. Esta prática é uma das maiores dificuldades encontradas por equipes de desenvolvimento de *sofware* científico. A prática de "cópia e cola" aumenta consideravelmente o número de linhas de programas, é uma fonte inerente de erros e faz com que o sistema venha a apresentar uma manutenção muito cara para ser atualizado na tentativa de seguir as tendências tecnológicas. Automatizar estas gerações sistemáticas de código e documentação é um ponto extremamente valioso na produção de *software* científico.

Neste capítulo, será feita uma breve apresentação de mecanismos de funcionamento e sintaxe da linguagem Python e de seu pacote numérico. Será dada uma visão geral das ferramentas escolhidas inicialmente para visualização de imagens e gráficos, elementos importantes na análise de resultados. Será apresentado o PIL[1] como ferramenta de leitura e gravação de vários formatos de arquivos de imagem. O ambiente Adesso de desenvolvimento de *software* será introduzido e será detalhada a primeira contribuição significativa deste trabalho com o suporte feito, para Python, para o processo de geração automática de código e documentação. Serão vistos ainda: a estruturação XML adotada, o processo de transformação, a geração de código, de

[1] *Python Imaging Library*

wrapper e *setup*, além da geração de documentação.

3.1 Programação em Python

Python é uma linguagem suficientemente genérica, capaz de ser aplicada a uma expressiva quantidade de problemas, superior a de outras linguagens no mesmo nível, como por exemplo, Awk, Perl ou Tcl. E ainda assim, apresenta um sintaxe bastante atraente pela elegância e simplicidade. Estas características proporcionam, em geral, redução de esforço na manutenção de código, desenvolvimento de aplicações de forma mais rápida que o de linguagens como C, C++ ou Java. O código-fonte resultante normalmente também é menor basicamente por três motivos: os tipos de dados de elevado nível, em Python, podem traduzir expressões complexas em comandos bastante simples; Python é uma linguagem não-declarativa, ou seja, não é necessário definir variáveis ou argumentos antes de utilizá-los e os mesmos podem assumir tipos diferentes conforme o fluxo do programa (tipagem dinâmica); um bloco de comandos, em Python, é marcado simplesmente pela indentação, forçando o desenvolvedor a manter um código visualmente limpo. Por estes motivos também, um programa Python se adapta facilmente na descrição de algoritmos por se apresentar, em geral, quase como um pseudo-código.

As estruturas de dados implementadas, em Python, dão boa flexibilidade ao usuário. Em C, estruturas equivalentes de mesmo rigor de otimização, custaria bom tempo de implementação. Outra característica importante é que, além destas estruturas, há um grande conjunto nativo de bibliotecas implementadas em C (*built-in*), para Python, que torna o processo de compilação/correção/re-compilação totalmente descartado (a menos que se queira estender a linguagem com uma API). Com isto, há um ganho em produtividade. A Tabela 3.1 apresenta um quadro resumo com as operações mais comuns sobre alguns destes tipos de objetos *built-in*. Em Python, além destas estruturas, funções, módulos, métodos, classes, código compilado também

são encarados como objetos.

3.1.1 Pacote Numerical

O pacote Numerical [3] é um conjunto de extensões para Python que oferece várias funcionalidades para manipulação de conjuntos de objetos chamados *arrays* que, por sua vez, podem ter qualquer número de dimensões (multidimensional). A vantagem destas extensões é permitir que se possa processar grandes volumes de dados (números) de forma tão rápida quanto ao que é feito, com os mesmos resultados, em linguagens compiladas. Além disto, também é oferecido suporte simples, com boa documentação, para construção de API's. A maior parte do código do Numerical foi escrita por Jim Hugunin, enquanto aluno de graduação do MIT. O trabalho então passou a ser mantido por um grupo de pesquisadores do laboratório LLNL da Universidade da Califórnia. O Numerical apresenta uma comunidade de desenvolvimento bastante ativa e vem, atualmente, sofrendo otimizações com o propósito de que o pacote passe a incorporar a distribuição padrão do Python.

Para exemplificar algumas funcionalidades do Numerical, volta-se agora ao exemplo da Seção 2.3, no qual se queria construir uma imagem em forma de xadrez. A Figura 3.1 mostra o conteúdo de um módulo com sete funções (`xadrez1`, `xadrez2`, ... `xadrez7`) que determinam resultados idênticos. O princício de funcionamento das diferentes versões são: em (1) é implementada a varredura explícita com preenchimento de valores pixel a pixel; em (2) é utilizada a função `fromfunction` do `Numeric`, na qual, pode-se definir uma função auxiliar que trabalhe com um índice em particular, representando o cálculo que é repetido para todos os demais índices conforme a dimensão dada; em (3) e (4) são construídas duas matrizes com malha de índices na horizontal e vertical (conforme a Figura 2.6), em (5) é criada uma pequena matriz 2x2 com elementos 0 e 1 na primeria linha, 1 e 0 na segunda linha, sendo a mesma replicada em ambas direções até a dimensão dada; em (6)

	Operações comuns	Exemplos de sintaxe
Números	inteiro normal	`>>> 1234`
	inteiro longo	`>>> 99999999L`
	ponto flutuante	`>>> 1.23, 3.14e-10`
	octal e hexadecimal	`>>> 0177, 0x9ff`
	complexo	`>>> 2+1j, complex(2,1)`
Strings	string vazia	`>>> s1=''`
	aspas	`>>> s2="Fulanos's name"`
	indexação e *slicing*	`>>> s2[i], s2[i:j]`
	concatenação e replicação	`>>> s1+s2, 3*s2`
	formatação	`>>> "Nome do %s" % 'Fulano'`
	aspas tripla	`>>> bloco="""..."""`
Listas	lista vazia	`>>> L1=[]`
	4 elementos: índices 0...3	`>>> L2=[0,1,2,3]`
	sub-listas aninhadas	`>>> ['abc',['def','ghi']]`
	indexação e *slicing*	`>>> L2[i], L2[i:j]`
	concatenação e replicação	`>>> L1+L2, 3*L2`
	métodos (inclusão)	`>>> L2.append(novo_valor)`
	exclusão	`>>> del L2[k], L2[i:j]=[]`
	associação de sub-lista	`>>> L2[i:j]=[1,2,3]`
	lista de inteiros	`>>> range(4), xrange(0,4)`
Dicionários	dicionário vazio	`>>> d1={}`
	3 elementos	`>>> d2={'pera':2,'uva':1,'maca':3}`
	aninhamento	`>>> d3={'cesta':{'uva':1,'banana':2}}`
	indexação	`>>> d2['maca'], d3['cesta']['uva']`
	métodos	`>>> d2.has_key('maca'), d2.keys()`
Tuplas	tupla vazia	`>>> ()`
	1 elemento	`>>> t1=(0,)`
	4 elementos	`>>> t2=(0,1,2,3)`
	aninhamento	`>>> t3=('abc',('def','ghi'))`
	indexação, *slicing*	`>>> t1[i], t1[i:j]`
	concatenação e replicação	`>>> t1+t2, 3*t2`

Tabela 3.1: Quadro resumo de algumas estruturas *built-in*.

1
```
def xadrez1(s):
    """Varredura explicita"""
    from Numeric import zeros
    a = zeros(s)
    for i in range(s[0]):
        for j in range(s[1]):
            a[i,j] = (i+j) % 2
    return a
```

2
```
def xadrez2(s):
    """Usando 'fromfunction'"""
    from Numeric import fromfunction
    def elementos(i,j):
        return (i+j) % 2
    a = fromfunction(elementos, s)
    return a
```

3
```
def xadrez3(s):
    """Usando 'indices'"""
    from Numeric import indices
    (i,j) = indices(s)
    return (i+j) % 2
```

4
```
def xadrez4(s):
    """Usando 'iameshgrid'"""
    from ia636 import iameshgrid
    (i,j) = iameshgrid(range(s[1]), range(s[0]))
    return (i+j) % 2
```

5
```
def xadrez5(s):
    """Usando 'iatile'"""
    from ia636 import iatile
    from Numeric import array
    return iatile(array([[0,1],[1,0]]), s)
```

6
```
def xadrez6(s):
    """Usando slices  """
    from Numeric import zeros
    a = zeros(s)
    a[ ::2,1::2] = 1
    a[1::2, ::2] = 1
    return a
```

7
```
def xadrez7(s):
    """Usando 'resize'"""
    from Numeric import arange, array, resize
    r = arange(s[1]) % 2
    linha = array([r, 1-r])
    a = resize(linha, s)
    return a
```

```
def xadrez_teste(s):
    """Testes das funcoes com shape s = (height,width)"""
    from time import time
    funlist = (xadrez1,xadrez2,xadrez3,xadrez4,xadrez5,xadrez6,xadrez7)
    print 'Tempos das funcoes para imagem de', s, '\n'+50*'-'
    for fun in funlist:
        t1, f, t2 = time(), fun(s), time()
        print fun.__name__, fun.__doc__, '\t->', '%.4f' %(t2-t1), 'segundos'
```

Figura 3.1: Módulo com sete funções diferentes para se gerar uma mesma imagem em forma de xadrez.

é criada uma matriz de zeros, assim como em (1), porém, desta vez é feito uso de *slices* para acessar e preencher todos elementos que devem receber o valor 1; finalmente, em (7), são construídas as duas primeiras linhas da imagem xadrez seguida do redimensionamento que se deseja. Observa-se que são feitas importações de funções do módulo `Numeric`. Este é mais comumente usado mas, no Numerical, há também os módulos `FFT`, `LinearAlgebra`, `MLab`, `RandomArray`, entre outros. As funções `iameshgrid` e `iatile` usadas, respectivamente, no `xadrez4` e `xadrez5` são provenientes do módulo `ia636` (vide Seção 4.1) que é um dos resultados deste trabalho. Adiantando, `iameshgrid` é similar ao `Numeric.indices`, porém aceita que os valores da malha sejam criados a partir de seqüências quaisquer, incluindo números fracionários por exemplo; e `iatile` tem a função de replicar uma matriz na horizontal e na vertical até que se atinja uma dada dimensão. Ainda no módulo da Figura 3.1, tem-se uma função encarregada de testar todas as demais, chamada `xadrez_teste`. Ao importar este módulo no intepretador Python (em um Athlon XP 1.4GHz), e para uma imagem de 1024x1024, é obtido o seguinte resultado:

```
>>> xadrez_teste((1024,1024))

Tempos das funcoes para imagem de (1024, 1024)
-----------------------------------------------
xadrez1 Varredura explicita     -> 7.1467 segundos
xadrez2 Usando 'fromfunction'   -> 0.3545 segundos
xadrez3 Usando 'indices'        -> 0.3540 segundos
xadrez4 Usando 'iameshgrid'     -> 0.2373 segundos
xadrez5 Usando 'iatile'         -> 0.0547 segundos
xadrez6 Usando slices           -> 0.0543 segundos
xadrez7 Usando 'resize'         -> 0.0236 segundos
```

Para ilustrar, em C, o tempo médio de varredura explícita de pixels (análogo ao `xadrez1`) foi de `0.0118 segundos` e a replicação de linhas (análogo ao `xadrez7`) foi de `0.0034 segundos`. Pode-se dizer, porém, que o uso de linguagens matriciais gera código, muitas vezes, mais eficiente até mesmo que o de linguagens de mais baixo nível (como visto na Tabela 2.1).

Esta afirmação só é válida considerando um mesmo resultado gerado a partir de soluções de programação em cada ambiente. No caso de operadores matriciais, por exemplo, não há informação explícita de como as iterações ocorrem internamente e, portanto, não há comparação de implementações idênticas. O que pretende-se dizer, com isto, é que sempre é possível programar em C mais eficientemente que em Python. Mas os operadores implementados em Python geralmente oferecem um grau de otimização dificilmente alcalçado em implementações despreocupadas em C. Uma observação interessante a partir desta experimentação em linguagens de níveis diferentes é que a prática de programação na forma matricial pode induzir a se escrever em C de forma mais eficiente. No exemplo do xadrez visto, a melhor solução obtida, para confirmar esta tese, faz uso de características das CPUs modernas que implementam a cópia de vetores em *hardware* (`memcpy` em C).

3.1.2 Interface Gráfica

A exibição de imagens é possível através de módulos como `wxPython`, `PyGTK`, `PyQt`, apenas para citar alguns. Optou-se pelo uso do módulo `Tkinter`, uma interface GUI[2] padrão do Python, orientada a objetos, para funcionalidades do pacote Tk [34]. Esta escolha se deu primeiramente por se tratar de um módulo nativo da linguagem, ou seja, sem necessidade de instalação de pacotes extras para este fim, e principalmente por se tratar de uma interface bastante utilizada em aplicações já desenvolvidas no sistema Adesso. O porte para Python fica, portanto, facilitado em muitos casos. A ferramenta implementada em `Tkinter` para mostrar uma imagem na tela vem sendo aprimorada pelo aluno de iniciação científica Guilherme Mazzela [11]. Quanto a exibição de gráficos, existem várias ferramentas que podem ser localizadas a partir do *site* oficial do Python. Foi escolhido o *Gnuplot*[3] [35] por ser bastante poderoso, portável, distribuído sob licença GNU-GPL, e ser

[2] *Graphical User Interface*
[3] `http://www.gnuplot.info`

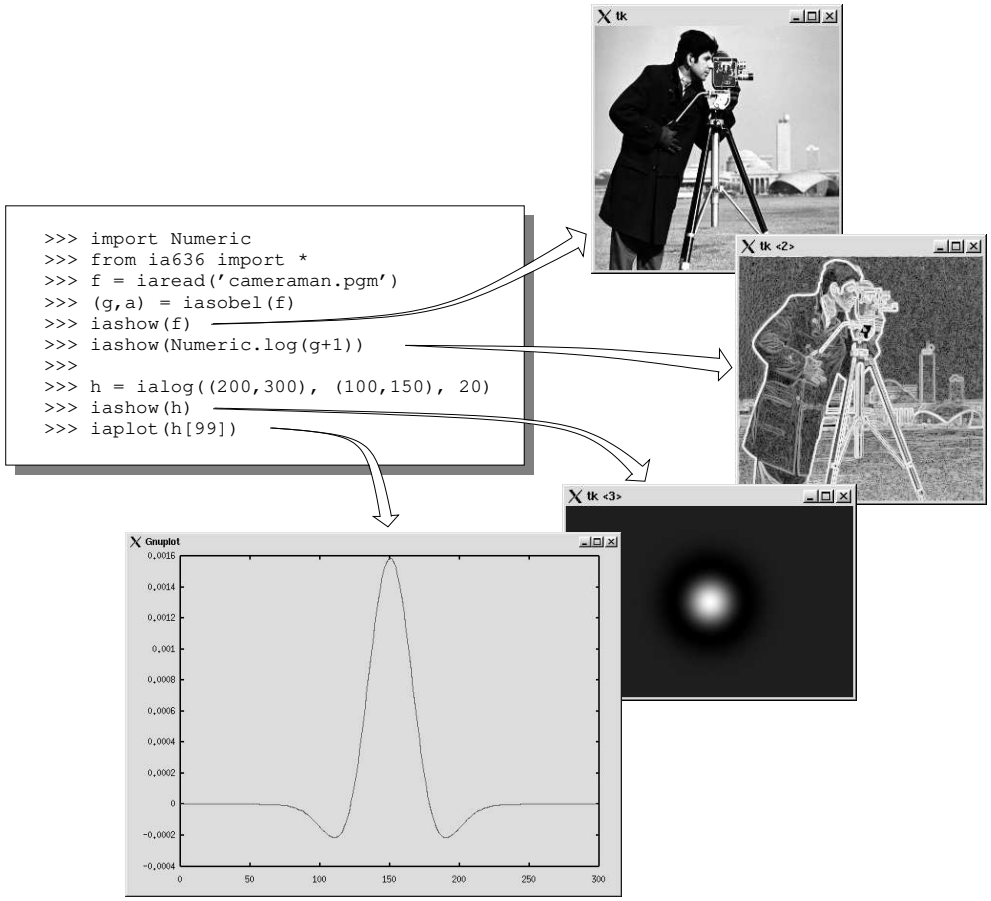


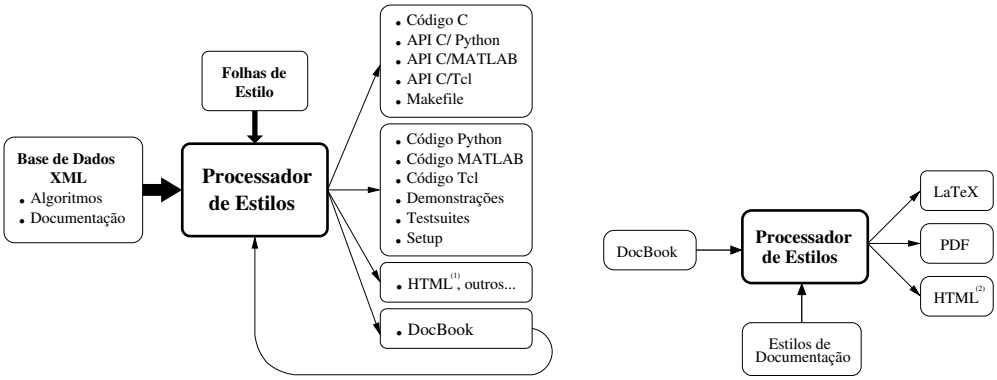
Figura 3.2: *Script* de teste de exibição de imagem e gráfico.

utilizado em vários sistemas de computação científica como, por exemplo, o Octave [8], uma opção gratuita compatível com o MATLAB. A interface *Gnuplot.py*[4] é escrita em Python puro, sendo simples de ser instalada em diversas plataformas. A Figura 3.2 mostra um *script* no ambiente do interpretador e as janelas geradas a partir do uso de funções de visualização implementadas (`iashow` para exibição de imagens e `iaplot` para gráficos). Neste exemplo, é feita a exibição de uma imagem original f, do logaritmo sobre o resultado da aplicação de uma filtro de Sobel em f. Depois, há a sintentização de uma imagem 2-D do Laplaciano da Gaussiana h e, por fim, a exibição do gráfico resultante da extração da centésima linha (a central) de h.

3.1.3 Formatos de Arquivo

O uso dos mais diversos formatos de arquivos de imagens é possível através da extensão PIL [18]. Este pacote, de código aberto e livremente distribuido, talvez seja a referência mais comum entre os *softwares*, em Python, que necessitam fazer leitura e escrita de imagens. Há uma lista discussão no *site* oficial da linguagem (`image-sig@python.org`) justamente para discutir as questões referentes ao suporte de imagens em Python e, sobretudo, as questões relativas ao pacote PIL. O PIL também inclui funcionalidades básicas de tratamento de imagens, porém além de sua arquitetura não ser facilmente visível ao usuário, é preferível utilizar funções normais de matrizes multidimensionais pois, desta forma, o aluno tem uma leitura melhor do código-fonte podendo, com isto, reforçar o entendimento das equações de um curso nesta área. De outra forma, o trabalho consistiria apenas de um conjunto de funções específicas programadas para resolver problemas de processamento de imagens. Neste mesmo caminho, quando a ferramenta escolhida é o MATLAB, procura-se evitar o uso da *toolbox* de processamento de imagens, mas sim os recursos inerentes a imagens de processamneto de

[4] `http://gnuplot-py.sourceforge.net`



(a) Geral. (b) Documentação.

Figura 3.3: Esquema de transformação do Sistema Adesso.

matrizes.

3.2 O Sistema Adesso

O Adesso [21, 7] é um projeto conjunto entre a Fundação CenPRA e a FEEC-Unicamp que explora o modelo de programação baseado em componentes reutilizáveis. Esta abordagem fornece suporte ao desenvolvimento de componentes e sua integração a diversas plataformas de programação científica. O Adesso configura-se em uma base de dados de algoritmos representada em XML [31] e em um conjunto de ferramentas de transformação (*stylesheets*) para a geração de código, documentação e empacotamento. A Figura 3.3(a) mostra o esquema de como esta transformação ocorre. Há uma base de dados XML, de entrada, contendo todas as informações pertinentes à caixa de ferramentas tais como algoritmos C, Python e diversos campos de documentação. Há um conjunto de folhas de estilo, cada qual para uma saída específica, contendo instruções para geração de código-fonte C ou Python, *testsuites*, arquivos de configuração e instalação (*setup.py*) e DocBook [33].

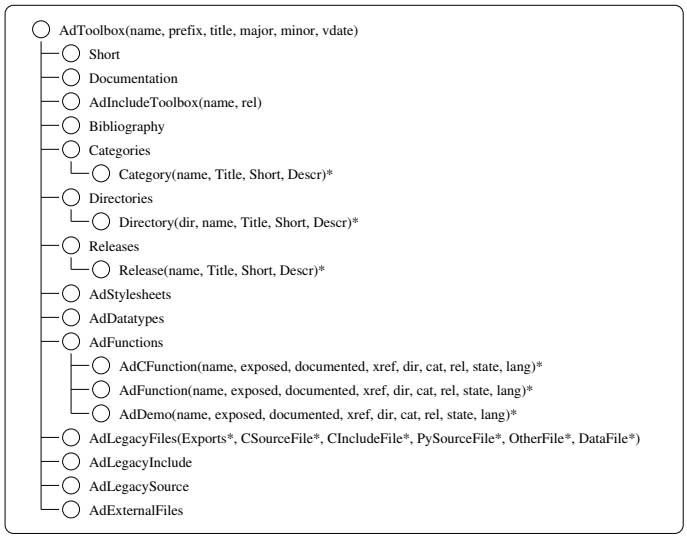


Figura 3.4: Estrutura XML do Adesso.

Este último formato é estruturado em XML, e facilita a geração de LaTeX, PDF[5], HTML[6], entre inúmeras outras possibilidades de documentação, dada uma folha de estilo específica de transformação. A Figura 3.3(b) ilustra este esquema de transformação para documentação. O processador de estilos do Adesso se encarrega de ler as duas entradas (base de dados em XML e a folha de estilo) e gerar a saída solicitada.

3.2.1 Estruturação em XML

Os arquivos XML do sistema Adesso formam uma base de dados contendo algoritmos e suas descrições, parâmetros de entrada e saída e suas descrições, exemplos, equações, testes, indicações de dependências, entre outros campos. Estas informações são totalmente armazenadas em texto ASCII, no qual, as marcações definem a estrutura em forma de árvore apresentada na Figura 3.4.

[5] *Portable Document Format*
[6] *HyperText Markup Language*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AdFunction name="iacos" state="alpha" lang="matlab python">
 ├─ <Dependencies><Module name="Numeric">cos sin pi</Module></Dependencies>
 ├─ <Seealso>iabwlp iacircle iacomb iagaussian ialog iaramp iarectangle</Seealso>
 ├─ <Short>Create a cossenoidal image.</Short>
 │
 ├─ <Documentation>
 │   ├─ <Descr name="Description">Generate a cosenosoid image of size s with amplitude 1, period T,
 │   phase phi and wave direction of theta. The output image is a double array.</Descr>
 │
 │   ├─ <Descr name="Examples">
 │   │   ├─ <Mcode>f = <Self/>([5 10], 6, 0)</Mcode>
 │   │   ├─ <Par/>
 │   │   ├─ <Mcode>f = <Self/>([128 256], 100, pi/4);
 │   │   │       iashow(f);
 │   │   │       plot(f(1,:));
 │   │   │       showfig(f(1,:));</Mcode>
 │   │   ├─ <PYcode>import Numeric
 │   │   │       f = <Self/>((5,10), 6, 0)
 │   │   │       print f</PYcode>
 │   │   ├─ <Par/>
 │   │   └─ <PYcode>f = <Self/>([256,256], 100, Numeric.pi/4)
 │   │           iashow(f)
 │   │           (g,d) = iaplot(f[0])
 │   │           showfig(f[0])</PYcode></Descr>
 │
 │   └─ <Descr name="Equation">
 │       └─ <Eq>f(x,y) &amp;amp; = &amp;amp; sin( 2\pi (f_x x + f_y y) + \phi) \\
 │                 f_x &amp;amp; = &amp;amp; \frac{cos(\theta)}{T} \\
 │                 f_y &amp;amp; = &amp;amp; \frac{sin(\theta)}{T}</Eq></Descr></Documentation>
 │
 ├─<Return name="f" type="mmIMAGE" output="yes" constraint="IMAGE"/>
 ├─<Args>
 │   ├─ <Arg name="s" type="mmIMAGE" dir="in" constraint="IMAGE" optional="no" default="" hidden="no">
 │   │   └─ <Descr>size: [rows cols].</Descr></Arg>
 │   ├─ <Arg name="t" type="mmIMAGE" dir="in" constraint="IMAGE" optional="no" default="" hidden="no">
 │   │   └─ <Descr>period: in pixels.</Descr></Arg>
 │   ├─ <Arg name="theta" type="mmDOUBLE" dir="in" constraint="DOUBLE" optional="no" default="" hidden="no">
 │   │   └─ <Descr>spatial direction of the wave, in radians. 0 is a wave on the horizontal direction.</Descr></Arg>
 │   └─ <Arg name="phi" type="mmDOUBLE" dir="in" optional="no" default="0" hidden="no">
 │       └─ <Descr>phase</Descr></Arg></Args>
 │
 ├─<Source lang="matlab"><Code><![CDATA[
 │     cols=s(2); rows=s(1);
 │     [x y] = meshgrid(0:cols-1,0:rows-1); freq = 1/t;
 │     fcols = freq * cos(theta); frows = freq * sin(theta);
 │     f = cos(2*pi* (fcols*x + frows*y) + phi);]]></Code></Source>
 │
 ├─<Source lang="python"><Code><![CDATA[
 │     cols, rows = s[1], s[0]
 │     (x, y) = iameshgrid(range(cols),range(rows)); freq = 1./t
 │     fcols = freq * cos(theta); frows = freq * sin(theta)
 │     f = cos(2*pi*(fcols*x + frows*y) + phi)]]></Code></Source>
 │
 ├─<Testsuite><Code lang="matlab"><![CDATA[...]]></Code><Code lang="python"><![CDATA[...]]></Code>
 │     <ExternalFiles/></Testsuite>
 ├─<Platforms>windows linux sunos</Platforms>
 └─<ExternalFiles/></AdFunction>
```

Figura 3.5: Exemplo de estrutura XML para descrição de uma função.

Entre parênteses aparecem os atributos do elemento em questão na árvore XML e, algumas vezes, para simplificar, seus filhos (escritos em maiúsculo). Os elementos notados por asterisco (*) indicam que pode haver mais filhos do mesmo tipo na árvore. A distinção, neste caso, é dada pelos atributos e conteúdo de tais elementos. A Figura 3.5 mostra um exemplo, sem o conteúdo dos *testsuites*, de um código XML para a descrição de uma função (`AdFunction`). Observa-se, exceto os trechos onde há código-fonte, que as informações podem ser compartilhadas para documentação de mais de uma linguagem, no caso, Python e MATLAB. A implementação de folhas de estilo para interpretar esta nova estrutura XML com suporte a múltiplas linguagens foi uma contribuição deste trabalho.

3.2.2 Processo de Transformação

O processador de estilos do Adesso é baseado em um processo de substituições de texto inspirado na linguagem XSL[7] [32]. A implementação é feita com uso do pacote tDOM [14] para Tcl (linguagem de programação do Adesso). O tDOM possibilita que se trabalhe com um *parser* XML e árvore DOM[8] [30], além da linguagem de *query* XPath [29]. Com este conjunto, passa-se a ter a representação de um documento XML em memória, garantindo, desta forma, o acesso rápido às marcações e atributos que podem ser localizados assim como se localiza um arquivo em um sistema de diretórios comum, também estruturado em forma de árvore. O exemplo de folha de estilo apresentado no Apêndice B ilustra a aplicação de alguns dos comandos do Adesso descritos a seguir:

sty:apply - implementa a varredura de um conjunto de elementos da árvore especificados por uma expressão XPath, transferindo o fluxo de processamento de uma folha de estilo à um `template` em particular;

[7] *Extensible Style Language*
[8] *Document Object Model*

sty:foreach - idêntico ao **sty:apply**, exceto por não fazer busca por `template`;

sty:value - retorna o *string* de um texto, atributo ou lista de atributos conforme a expressão XPath fornecida;

sty:if - avalia se uma expressão XPath resulta verdadeiro, procedendo ou não a substituição solicitada;

sty:iif - similar ao **sty:if**, mas avalia uma expressão Tcl em vez de uma expressão XPath;

sty:par - realiza tratamento de parâmetros ou operações aritméticas simples;

sty:call - retorna o conteúdo de uma macro definida;

sty:tag - gera marcações XML/HTML. Exemplo: `[sty:tag body]` produz `<body>`;

sty:save - grava em arquivo todas as substituições geradas;

sty:copy - copia arquivos quaisquer;

sty:logIt - exibe mensagens para comentar o fluxo de execução.

3.2.3 Geração de Código

Através do conteúdo registrado em XML, representado nas Figuras 3.4 e 3.5, é possível gerar automaticamente código-fonte em uma dada linguagem. Folhas de estilos para Python e MATLAB foram implementadas com esta finalidade. Para exemplificar esta transformação, restringe-se a solução adotada ao Python. As demais linguagens de *scripting* seguem modelo semelhante de formação de código.

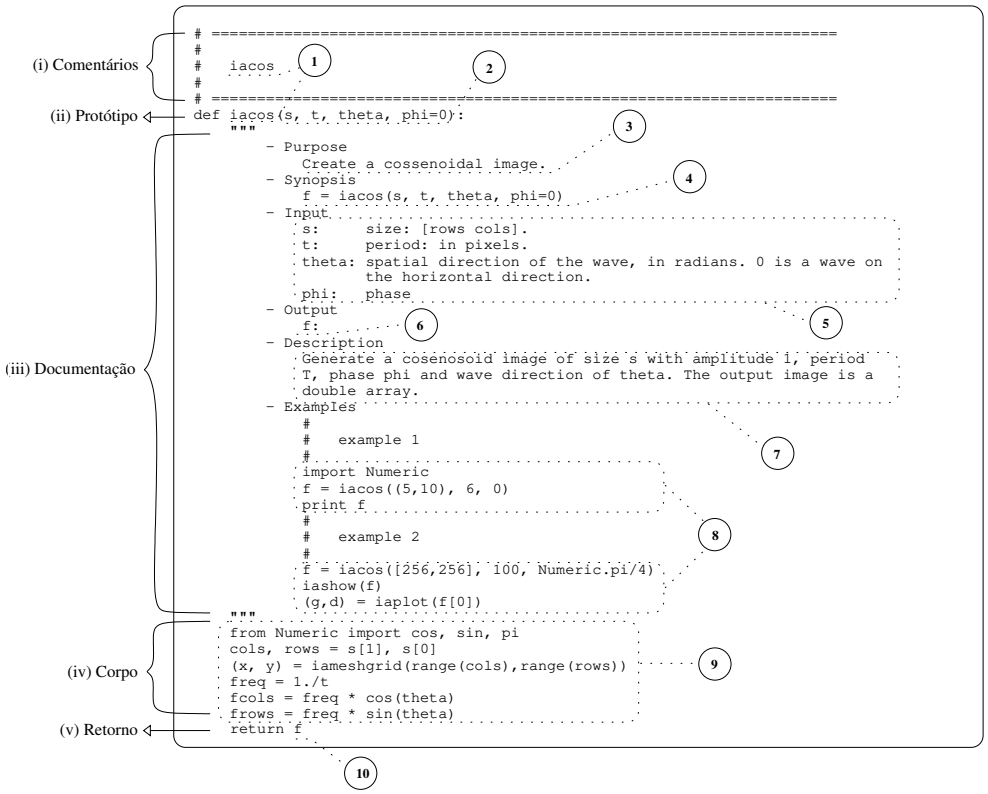


Figura 3.6: Exemplo de código-fonte Python gerado automaticamente.

O Apêndice B mostra o corpo principal (sem a declaração das diversas macros usadas) da folha de estilo para codificação em Python. A caixa de ferramentas, a ser gerada, foi definida como sendo formada por três grandes módulos: de funções em si (`ia636`); de demonstrações ou lições (`ia636demo`); de conjuntos de testes chamados *testsuites* (`ia636test`).

Geração de Funções

Uma função Python pode ser dividida em 5 partes: (i) comentários de programação através do sustenido (#); (ii) protótipo ou cabeçalho definido pela palavra reservada `def`; (iii) documentação, entre aspas tripla (`"""`),

que poderá ser consultada no interpretador com uso do comando `help` do Python; (iv) corpo do código com todas as instruções da linguagem para implementação do procedimento; e (v) descrição dos objetos a serem retornados definidos pela palavra reservada `return`. A Figura 3.6 mostra um código de função gerado automaticamente, indicando esta estrutura. Os campos numerados de 1 a 10 foram extraídos dos documentos XML de entrada: (1) atributo *name* da marcação XML `AdFunction`; (2) listagem de todos argumentos, dados pelo atributo *name* da marcação `Arg`, separados por vírgula, com a associação de seus respectivos valores padrão, dados pelos atributos *optional* e *default*; (3) descrição curta da finalidade da função, dada pelo texto da marcação `Short` (filho de `AdFunction`); (4) cópia da indicação (2), com o acréscimo da atribuição às variáveis de retorno, dada pelo atributo *name* da marcação `Return`; (5) listagem novamente dos nomes dos argumentos seguidos de suas descrições, dadas pelo texto da marcação `Descr` (filho de `Arg`); (6) de novo, a listagem das variáveis de retorno, agora seguidas por suas descrições; (7) descrição mais detalhada dos objetivos e funcionamento da função, dada pelo texto da marcação `Descr`, cujo atributo *name* é "Description"; (8) textos dos *scripts* contidos nas marcações `PYcode` para exemplificar a forma de uso da função; (9) código propriamente dito proveniente da marcação `Source`, cujo atributo *lang* é definido como "python"; (10) nome das variáveis de retorno separadas por vírgula, caso haja mais de uma.

Este processo de construção se repete para todas as funções da base de dados XML. Todo o código gerado desta forma constitui o módulo `ia636`.

Geração de Demonstrações

Demonstrações são funções diferenciadas, sem parâmetro de entrada e com retorno nulo (None), estruturadas pela marcação `AdDemo`. Como filhos desta, há uma seqüência de marcações `Slide`, cada qual, responsável por registrar um trecho de uma explicação sobre um algoritmo ou um trecho de

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<AdDemo name="iadftmatrixexamples" state="alpha" lang="matlab python">
  |
  |---<Dependencies/>
  |---<Seealso/>
  |---<Short>Demonstrate the kernel matrix for the DFT Transform.</Short>
  |---<Documentation/>
  |
  |---<Source>
  |     |---<Slide title="Kernel images generated">
  |     |     |---<Descr>Imaginary and real parts of the DFT kernel.</Descr>
  |     |     |---<Code lang="matlab">A = iadftmatrix(128);
  |     |     |       Aimag = imag(A); Areal = real(A);
  |     |     |       iashow(Aimag);
  |     |     |       iashow(Areal);</Code>
  |     |     `---<Code lang="python">A = iadftmatrix(128)
  |     |             Aimag, Areal = A.imag, A.real
  |     |             iashow(Aimag)
  |     |             iashow(Areal)</Code></Slide>
  |     |
  |     `---<Slide title="Three first lines">
  |           |---<Descr>Three first lines from imaginary and real parts
  |           |       of the kernel matrix. Observe the increasing
  |           |       frequencies of the senoidals (imaginary
  |           |       part) and cossenoidals (real part).</Descr>
  |           |---<Code lang="matlab">x = 0:127;
  |           |       plot(x,Aimag(1,:), x,Aimag(2,:), x,Aimag(3,:));
  |           |       showfig(x,Aimag(1,:), x,Aimag(2,:), x,Aimag(3,:));
  |           |       plot(x,Areal(1,:), x,Areal(2,:), x,Areal(3,:));
  |           |       showfig(x,Areal(1,:), x,Areal(2,:), x,Areal(3,:));</Code>
  |           `---<Code lang="python">
  |                   g1,i1 = iaplot(Aimag[0,:]); g1,i2 = iaplot(Aimag[1,:]); g1,i3 = iaplot(Aimag[2,:])
  |                   g1.plot(i1,i2,i3)
  |                   showfig(i1,i2,i3)
  |                   g2,r1 = iaplot(Areal[0,:]); g2,r2 = iaplot(Areal[1,:]); g2,r3 = iaplot(Areal[2,:])
  |                   g2.plot(r1,r2,r3)
  |                   showfig(r1,r2,r3)</Code></Slide></Source>
  |
  |---<Platforms>windows linux sunos</Platforms>
  `---<ExternalFiles/></AdDemo>
```

Figura 3.7: Exemplo de estrutura XML para descrição de uma demonstração.

uma lição sobre algum tópico de processamento de imagens. A Figura 3.7 ilustra um exemplo desta organização.

Todas as demonstrações constituem o módulo `ia636demo`. As funções geradas ilustram várias saídas, numéricas ou gráficas (pelo `print`, `iashow` ou `iaplot`), passo a passo. A Figura 3.8 mostra o código gerado para a estrutura XML exemplificada na Figura 3.7. As indicações feitas são: (1) atributo *name* de `AdDemo`; (2) texto da marcação `Short` (filho de `AdDemo`); (3) texto da marcação `Descr` (filho de `Slide`); (4) texto da marcação `Code`, cujo atributo *lang* é "python", que contém o *script* de execução. Observe que em (4), o *script* é visualizado na tela pelo comando `print` e depois executado. Neste exemplo, o *slide* 1 exibe duas imagens (`Aimag` e `Areal`). Ao seu término, é pedido para que se pressione "return" para continuar. Inicia-se o próximo *slide* que, neste caso, exibe dois gráficos. Na Figura 3.9, pode-se ver as saídas desta função quando a mesma é chamada no interpretador.

Figura 3.8: Código-fonte de demonstração gerado automaticamente.

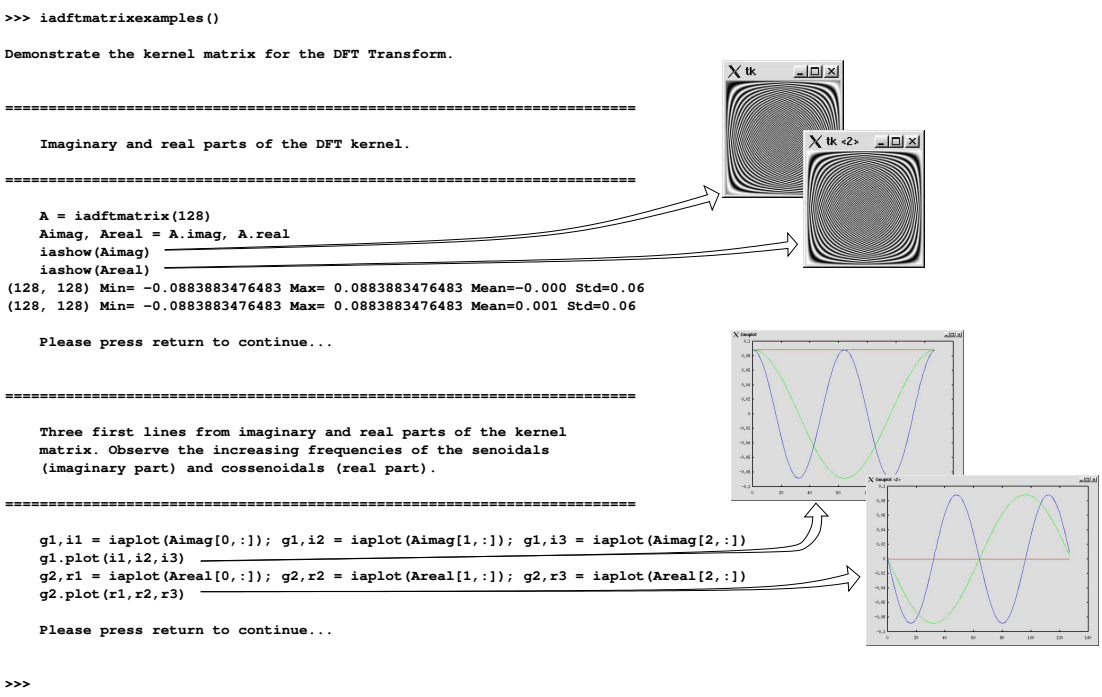


Figura 3.9: Exemplo de saída gerada quando na execução de uma demonstração.

Geração de *Testsuites*

Os *testsuites* são exemplos que comparam as saídas das funções com valores considerados corretos, gerado na maioria das vezes, analiticamente. A Figura 3.10 mostra, à esquerda, um trecho XML que organiza o tratamento dos *testsuites* e, à direita, um trecho do resultado da transformação, o módulo `ia636test`, feita pelo processador de estilos. Os campos numerados são: (1) e (2), respectivamente, os atributos *name* e *title* de `AdToolbox`; (3) atributo *name* de `AdFunction`, seguido da descrição curta da função em `Short` (filho de `AdFunction`); (4) texto de `Code` (filho de `Testsuite`), cujo atributo *lang* é "python".

Ao importar o módulo `ia636test`, obtém-se como resultado o teste de todas funções implementadas da caixa de ferramentas. A seguir, são mostradas as primeiras linhas desta saída:

```
>>> import ia636test

Testsuite ia636 -- Toolbox ia636

======================================================================
iabwlp - Low-Pass Butterworth frequency filter.
======================================================================
testing iabwlp
1

======================================================================
iacircle - Create a binary circle image.
======================================================================
testing iacircle
1
1
1

======================================================================
iacos - Create a cossenoidal image.
======================================================================
testing iacos
1
...
```

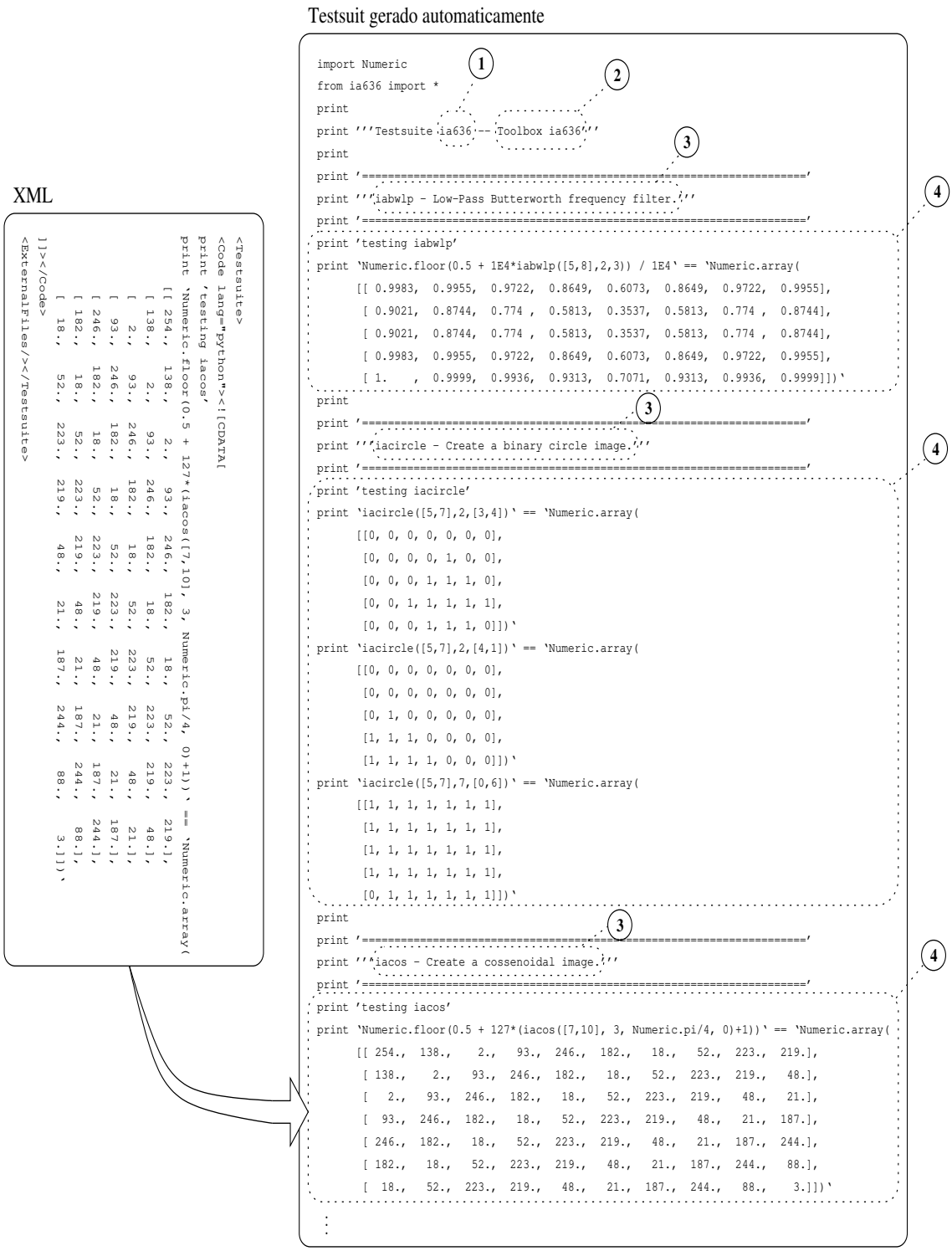


Figura 3.10: Exemplo de *testsuite* gerado automaticamente.

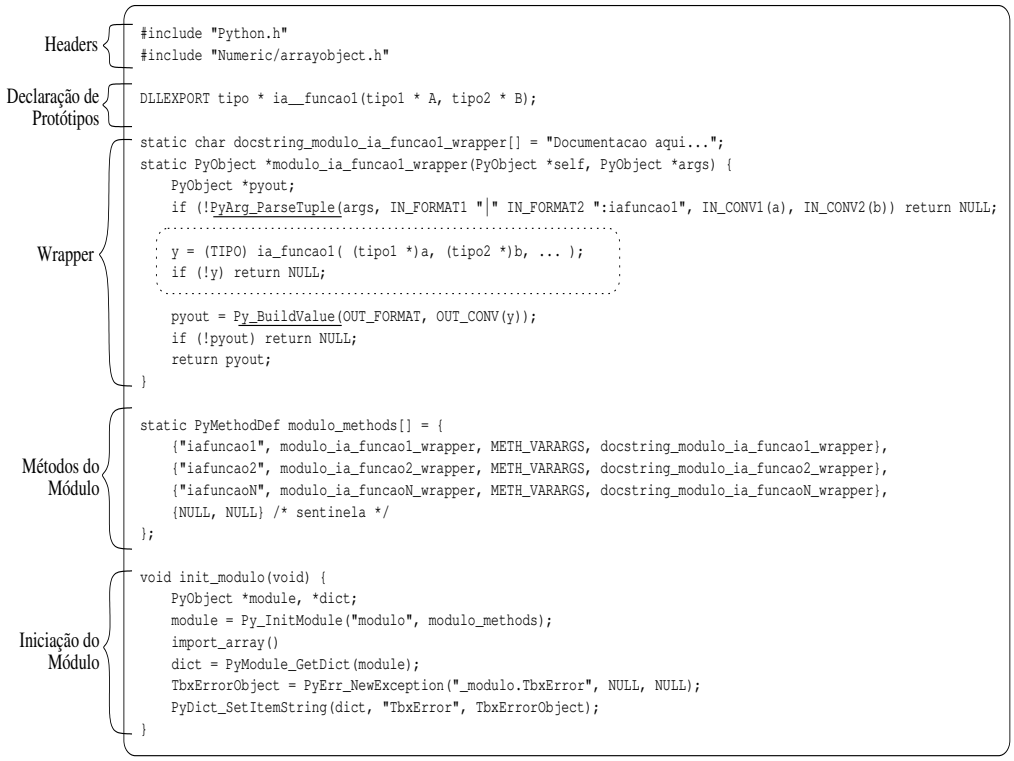


Figura 3.11: Modelo de construção de um módulo *built-in*.

Com esta funcionalidade em mãos, espera-se cobrir o máximo de casos de erro de implementações e detectar rapidamente qualquer alteração de resultado quando há atualização da caixa de ferramentas.

3.2.4 Geração de *Wrapper*

Para que uma função C/C++ possa ser utilizada no ambiente Python é preciso seguir algumas regras de construção de módulos *built-in*. A documentação do Python e do Numeric para implementação de uma API[9] é bastante detalhada e oferece vários exemplos. Basicamente é preciso des-

[9] *Application Programmer's Interface*

crever um envoltório, conhecido como *wrapper*, na chamada das funções C/C++. Além disto, devem ser especificados os arquivos de declaração principais (`Python.h` e `Numeric/arrayobject.h`), ser declarados os protótipos, os métodos e, por fim, a caixa de ferramentas deve ser iniciada. A Figura 3.11 exemplifica a construção de um módulo chamado "modulo" e descreve a construção de um *wrapper* para uma função "funcao1". Em Python, o envoltório pode ser implementado com as funções `PyArg_ParseTuple` e `Py_BuildValue`. A primeira se encarrega de transformar os argumentos do tipo `PyObject` fornecidos, quando na chamada da função pelo interpretador, para o tipos definidos, em C, da função, além de verificar se o número de argumentos de entrada está correto, considerando também os parâmetros opcionais. A segunda faz exatamente o contrário, pegando a saída da função para transformá-la em um objeto Python, hábil a ser utilizado no interpretador.

Este processo é construído seguindo uma sintaxe criteriosa e bem definida. Isto proporciona que tanto o *wrapper* como todo o módulo *built-in* possa ser gerado automaticamente pelo Adesso. Em XML, as funções C são definidas pela marcação `AdCFunction`. Exceto pela inexistência do atributo "lang", esta estrutura é a mesma que foi vista para código Python puro (`AdFunction`). No entanto, aqui é essencial que se registre informações de configuração, de descrição de dependências, de localização de bibliotecas, de definição de constantes e macros, para que haja sucesso na compilação. Um aspecto importante desta automação é o mapeamento de tipos de dados (*typemaps*). *Typemaps* são trechos de código C a serem incluídos no *wrapper* em suas diversas seções: (i) declaração de variáveis para cada marcação XML `Arg` e `Return`; (ii) conversão de tipos para cada `Arg` e verificação do número de argumentos (`PyArg_ParseTuple`); (iii) chamada da função base; (iv) conversão de tipos para a saída (`Py_BuildValue`). A Figura 3.12 ilustra uma parte da folha de estilo responsável pelo mapeamento do tipo inteiro. Observe as transformações indicadas pelas setas que vão do *typemap* para o código *wrapper* gerado. Aqui optou-se por usar a função `PyInt_FromLong`

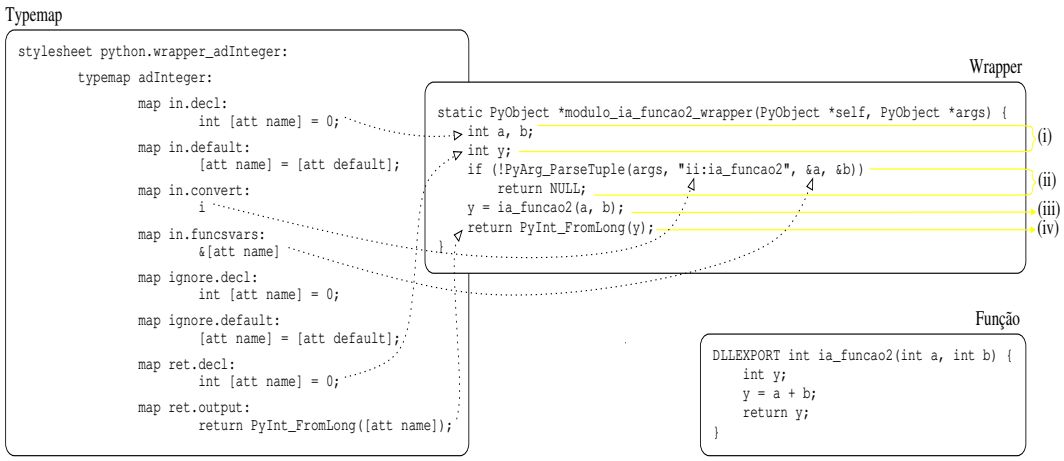


Figura 3.12: Exemplo de *typemap* para o *wrapper* de uma função de soma de dois inteiros.

específica para inteiros ou inteiros longos, mas também poderia ser utilizado `Py_BuildValue`. Ainda é mostrada a função base no canto inferior direito (`ia_funcao2`) da figura.

As conversões (ii) e (iv) são feitas diretamente quando se trata de tipos pré-definidos (`string`, `integer`, ponteiro para `integer`, `float`, ponteiro para `float`, entre outros), porém, podemos tratar vetores, imagens ou qualquer outra estrutura. Nestes casos, felizmente, é possível fazer chamadas às funções de conversão. Foram implementados os seguintes mapeamentos:

- Transformação de um tipo objeto Python em um vetor de inteiros, e vice-versa:
  `int adpython2intvector(PyObject *arg, int **vector)`
  `PyObject *adintvector2python(int *vector)`

- Transformação de um tipo objeto Python em um vetor de reais (double), e vice-versa:
  `int adpython2dblvector(PyObject *arg, double **vector)`
  `PyObject *addblvector2python(double *vector))`

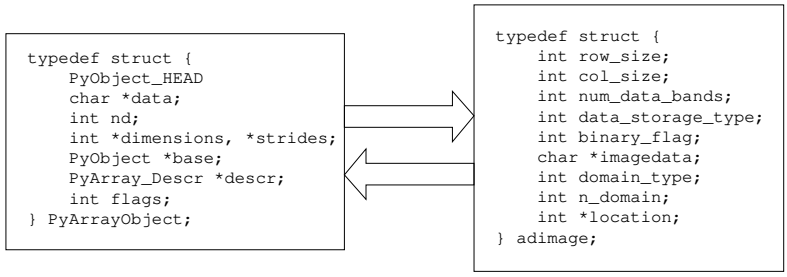


Figura 3.13: Estruturas `PyArrayObject` e `adimage`.

- Transformação de um tipo objeto Python em um tipo opaco, e vice-versa:
  `int adpython2pointer(PyObject *arg, void **ptr))`
  `PyObject *adpointer2python(void *ptr)`)
  O tipo opaco é o mapeamento *default* feito pelo gerador de código caso o tipo tratado não esteja previamento definido no Adesso.

- Transformação de um tipo objeto Python (`PyArrayObject`) em um tipo imagem (`adimage`), e vice-versa:
  `int adpython2adimage(PyObject *arg, adimage **img)`
  `PyObject *adadimage2python(adimage *img)`)
  A Figura 3.13 mostra as estruturas de `PyArrayObject` e `adimage`. É preciso mapear os campos destes dois tipos entre si.

Finalizados estes ajustes, é a vez da folha de estilo agir para se gerar código-fonte de API Python/C. Optou-se por incluir todas as funções C e respectivos *wrappers* em um único arquivo, facilitando desta forma, o próximo passo que é o de compilação. A folha de estilo implementada instrui a geração do módulo *buit-in* incluindo, na ordem, os seguintes trechos de código: (1) *headers* do Python e Numeric; (2) chamadas a funções que promovem alocação de memória com suporte a depuração; (3) código das funções de mapeamento de tipos; (4) código principal das bibliotecas

da caixa de ferramentas; (5) *wrappers* para as funções em (4); (6) especificação de todos os *wrappers* na variável de métodos do módulo (do tipo `PyMethodDef`); (7) especificação da função de iniciação do módulo (`void init_nome_do_módulo(void)`).

Este processo sistemático automatizado fortalece a consistência da sintaxe, aumenta a imunidade a erros comuns da programação manual, além de possibitar que uma biblioteca, em C, possa se portada em Python com um mínimo de esforço. O trabalho para portar uma biblioteca existente consiste agora basicamente em se descrever os protótipos das funções (argumentos de entrada e saída) e incluir (ou referenciar) o corpo de cada código na base de dados XML. Não exige, portanto, ser feito por um programador.

3.2.5 Geração de *Setup*

Uma vez que o código Python puro e o código API Python/C foram gerados, a intenção agora é empacotá-los para que se tenha um produto final, de tal modo que seja facilmente instalado em qualquer plataforma. Nas versões mais recentes do Python, há suporte para empacotamentos deste tipo através do módulo `distutils`[10] (em versões mais antigas, este módulo pode ser incorporado). Os pacotes Python comumente trazem em sua distribuição um *script* chamado `setup.py`. Nele, são utilizados métodos do `distutils` que, a partir basicamente de referências a localizações dos arquivos Python e bibliotecas C/C++ (caso haja) da caixa de ferramentas gerada, determinam: (i) a criação de código intermediário (`bytecode`) Python, verificando, neste tempo, se há algum erro de sintaxe; (ii) a compilação das API's ligando a bibliotecas necessárias do Python automaticamente. Caso não se tenha qualquer implementação em C, o passo (ii) é desconsiderado. Em um *console* padrão do sistema operacional, usa-se comumente o `setup.py` de três formas:

1. `python setup.py build` - para construção dos passos (i) e (ii), cita-

[10] *Python Distribution Utilities*

```
from distutils.core import setup, Extension

my_headers     = ['modulo.h']
my_ext_modules = Extension( '_module',
                            define_macros = [('EXEMPLO', None)],
                            include_dirs  = incdir1 + incdir2 + incdirN,
                            library_dirs  = libdir1 + libdir2 + libdirN,
                            libraries     = ['lib1', 'lib2', 'libN'],
                            sources       = ['module_wrapper.c'] )
my_py_modules  = ['modulo', 'modulodemo', 'modulotest']
my_data_files  = ['data/img1.tif', 'data/img2.jpg', 'data/imgN.ppm']

setup ( name        = "modulo",
        version     = "1.0",
        description = "Descricao aqui...",
        author      = "Autor aqui...",
        py_modules  = my_py_modules,
        ext_modules = my_ext_modules,
        headers     = my_headers,
        data_files  = my_data_files )
```

Figura 3.14: Modelo de construção do `setup.py`.

dos anteriormente, em diretório temporário, além da cópia de todos arquivos de dados (como imagens) e módulos auxiliares.

2. `python setup.py install` - para execução do item 1, caso ainda não tenha sido feito, e instalação do pacote no ambiente Python instalado.

3. `python setup.py bdist` - para criação de distribuição binária, na plataforma em questão, a fim de deixar o pacote pronto para ser distribuído e posteriormente instalado.

A Figura 3.14 mostra um modelo de construção do *script* `setup.py`. Uma folha de estilo foi implementada com o intuito de se gerar este arquivo automaticamente. Os parâmetros de entrada de `Extension` e `setup` são especificados extraindo-se as informações registradas em XML.

3.2.6 Geração de Documentação

A geração de documentação consiste em se descrever todos os componentes do pacote computacional. Alguns requisitos são considerados importantes neste processo: (i) descrição das funções em si, citando sua finalidade, sintaxe e parâmetros de entrada e saída; (ii) determinação de exemplos e as

saídas que devem ser produzidas quando os mesmos são interpretados pela linguagem em uso; (iii) estabelecimento de fórmulas, sempre que possível, pois tornam a descrição da implementação mais precisa - saber qual é o comportamento do algoritmo implementado é um ponto crítico de *software* científico (por exemplo: é muito vago dizer que se trata de uma melhoria de contraste sem uma equação); (iv) listagem de seu algoritmo ou código-fonte, e funções associadas.

Uma folha de estilo para documentação foi implementada considerando-se inicialmente que os arquivos XML de entrada passem pelo gerador de estilos e sejam diretamente transformados em documentos HTML (caminho para se obter HTML$^{(1)}$ da Figura 3.3). A novidade desta folha de estilo, em relação a de geração de código-fonte, é a necessidade de se gerar ilustrações didáticas com o objetivo de facilitar a explicação do funcionamento do *software* produzido. Por exemplo, quando há a marcação de exemplo na descrição da função (`PYcode` para Python, ou `Mcode` para MATLAB) o Adesso invoca o interpretador Python que executa o *script* definido nesta marcação XML, gerando as saídas correspondentes aproveitadas no conteúdo da documentação. Há também a marcação de equações (`Eq`) onde é a vez do interpretador LaTeX ser invocado, gerando uma figura de fórmulas. São basicamente quatro tipos de saídas: (1) em forma de texto simples ou numéricas; (2) imagens; (3) gráficos; (4) equações. A Figura 3.15 mostra uma página HTML produzida automaticamente e um esquema das transformações para estas quatro saídas.

Para agilizar a chamada ao interpretador, foi utilizada uma interface com o Python chamada Tclpython [10]. Desta forma, é possível usar todo o ambiente Python instalado em linhas de comando Tcl que é a linguagem do Adesso. Pode-se executar então os *scripts*, em `PYcode`, e mapear as saídas diretamente em variáveis Tcl. Assim, quando na passagem por um comando `print`, no *script* propriamente dito ou dentro das funções chamadas, é registrada a saída texto a partir do re-direcionamente da saída padrão no Python (`sys.stdout`) para uma variável. Já na passagem por um `iashow` é dado um

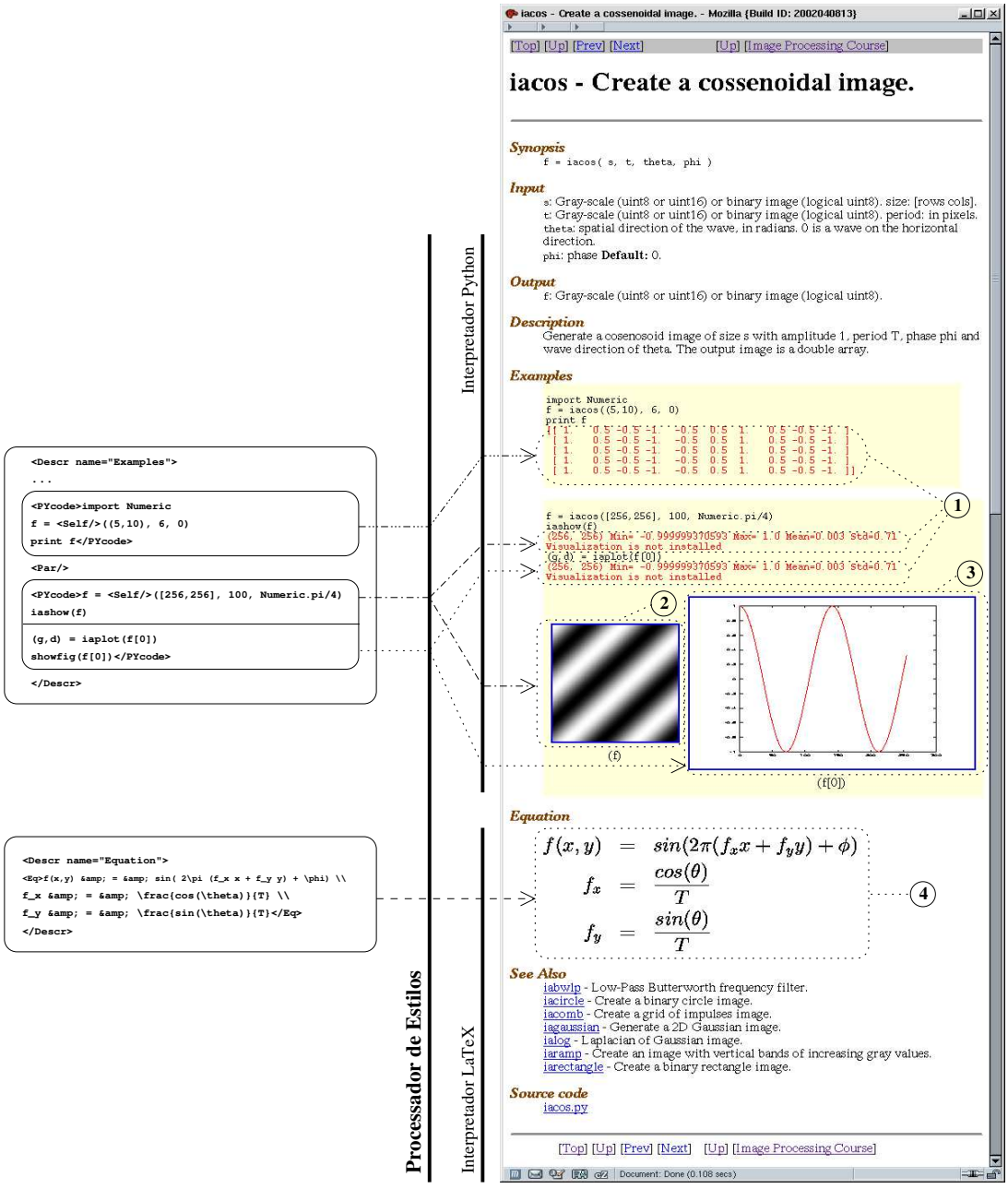


Figura 3.15: Documentação HTML gerada automaticamente.

tratamento diferenciado no Tcl. Neste caso, o interpretador Python aberto passa a gravar, em arquivo (usando o próprio `iawrite` da caixa de ferramentas) a imagem processada em memória. Algo similar acontece quando se detecta um `iaplot` (mais especificamente, um `showfig` para que se reconheça as saídas do `iaplot` localizado imediatamente antes). A folha de estilo agora utiliza as funcionalidades do *gnuplot* para gravar as figuras em formato PS[11]. Feito isto, o Tcl passa a chamar o *software* Ghostscript para conversão do PS criado em qualquer outro formato reconhecível por um navegador de Internet.

Observe que é necessário manter apenas informações textuais para gerar uma documentação ilustrada. Evidentemente que também é mantido um conjunto básico de imagens fotográficas mas todas as figuras, referenciadas na documentação, são criadas durante a sua geração. A importância da geração das figuras desta forma é manter consistência entre a documentação e a última versão das implementações.

Em relação às demonstrações (`AdDemo`), assim como foi gerado o código `ia636demo`, a folha de estilo de documentação HTML também se encarrega de montar páginas ilustrativas contendo os *slides* para cada uma destas funções especiais.

DocBook

DocBook é um forma aprimorada de automatização da documentação, mais genérica que a mostrada anteriormente onde, em vez de se gerar diretamente um formato específico (como HTML$^{(1)}$ da Figura 3.3(a)), produzem-se documentos XML intermediários com marcações padronizadas [33, 21]. Imagens, gráficos e equações são gerados em um primeiro momento e suas localizações devidamente referenciadas nos arquivos XML/DocBook. A grande vantagem desta abordagem fica evidente quando se deseja o produto final (HTML$^{(2)}$, PDF, LaTeX, ou outro formato, conforme a Figura 3.3(b)). Agora

[11] *PostScript*

basta ter uma única folha de estilo de interpretação do DocBook, dado um produto final em particular, para todas as linguagens. A consistência citada anteriormente assume um sentido ainda mais amplo, pois agora há solidez entre as documentações de uma mesma caixa de ferramentas para diferentes linguagens.

Capítulo 4

Resultados

Neste capítulo, serão exibidas as funções e lições da caixa de ferramentas implementadas, será apresentado um sistema criado para correção automática de programas Python de alunos via Internet, será mostrado o resultado de uma avaliação de alunos feita ao ambiente de suporte ao ensino de processamento de imagens usando Python e, por fim, será ilustrada uma comparação de desempenho em relação à plataforma MATLAB anteriormente utilizada.

4.1 Caixa de Ferramentas

De posse do sistema Adesso, ajustado ao desenvolvimento em Python, foi possível criar todas as funções, lições, demonstrações e testes da caixa de ferramentas para processamento de imagens, além de gerar toda a documentação correspondente [26, 2]. São mais de 60 algoritmos implementados distribuídos pelos módulos `ia636`, `ia636demo` e `ia636test`. O prefixo destes nomes vem da sigla da disciplina de Visão Computacional da Unicamp. Todas as funções implementadas também recebem prefixo "ia" para facilitar a identificação. Espera-se que este pacote sempre incorpore novas funcionalidades. Uma descrição geral de seu estado atual é feita a seguir. A

documentação mais detalhada pode ser vista nos Apêndices C e D.

Foram implementadas diversas lições:

- Geração de imagens sintéticas diversas (`iagenimages`);
- Função de transformação de contraste (`iait`);
- Equalização de histograma (`iahisteq`);
- Teorema da convolução (`iaconvteo`);
- Técnica de casamento de padrão (`iacorrdemo`);
- Demonstração do espectro de Fourier para imagens sintéticas simples (`iadftexamples`);
- Propriedade de escala da Transformada de Discreta de Fourier e caracterização da forma da matriz núcleo desta transformação (`iadftscaleproperty` e `iadftmatrixexamples`);
- Decomposição de imagens em ondas primitivas 2-D (`iadftdecompose`);
- Interpolação de imagens aumentadas (`iamagnify`);
- Restauração pela filtragem inversa (`iainversefiltering`);
- Tranformada de Hotelling (`iahotelling`);
- Ilustração do método de seleção de thresholding de Otsu (`iaotsudemo`).

Foram implementadas diversas funções divididas nos seguintes tópicos:

- **Criação de imagem:** filtro passa-baixas de Butterworth (`iabwlp`), círculo (`iacircle`), impulsos (`iacomb`), cossenoidal 2-D (`iacos`), gaussiana 2-D (`iagaussian`), Laplaciano da Gaussiana (`ialog`), bandas verticais com intensidade crescente (`iaramp`) e retângulo (`iarectangle`);

- **Manipulação e informação:** recorte do retângulo mínimo de uma imagem (`iacrop`), conversão de índices lineares para x-y e vice-versa (`iaind2sub` e `iasub2ind`), criação de malha 2-D de índices (`iameshgrid`), negação (`ianeg`), normalização de intensidades (`ianormalize`), acréscimo de borda (`iapad`), seleção de uma sub-imagem (`iaroi`) e replicação matricial até uma nova dimensão (`iatile`);

- **Arquivo de imagens:** leitura (`iaread`) e gravação (`iawrite`);

- **Manipulação de constraste:** transformação de intensidades (`iaapplylut`), seleção de mapa de cores (`iacolormap`);

- **Processamento de cor:** true color RGB para imagem de índices e tabela de cores (`iatcrgb2ind`), conversão RGB-HSV (`iargb2hsv` e `iahsv2rgb`) e conversão RGB-YCbCr (`iargb2ycbcr` e `iaycbcr2rgb`);

- **Manipulações geométricas:** transformação afim (`iaffine`), transformação e escala de corpo rígido 2-D (`iageorigid`), translação periódica (`iaptrans`) e redimensionamento (`iaresize`);

- **Transformações:** Transformada Discreta de Cossenos (`iadctmatrix` e `iadct`), Transformada Discreta de Fourier (`iadftmatrix` e `iadft`), Transformada Wavelet de Haar (`iahaarmatrix` e `iahwt`), Transformada de Hadamard (`iahadamardmatrix` e `iahadamard`) e, respectivamente, suas inversas (`iaidct`, `iaidft`, `iaihwt` e `iaihadamard`), além de funções de translação de espectro da componente zero (`iafftshift` e `iaifftshift`) e verificação de simetria conjugada (`iaisdftsym`);

- **Filtragem:** filtro do Laplaciano da Gaussiana (`ialogfilter`), contornos de imagens binárias (`iacontour`), convolução linear e periódica 2-D (`iaconv` e `iapconv`), detecção de borda Sobel (`iasobel`) e filtro de variância (`iavarfilter`);

- **Técnica de thresholding automático:** Thresholding de Otsu (`iaotsu`);
- **Medidas:** histograma (`iahistogram` e `iacolorhist`), rotulação (`ialabel`), reconstrução de componente conexo (`iarec`) e cálculo do MSE/PSNR/correlação de Pearson entre duas imagens (`iastat`);
- **Visualização:** exibição da transformada de Fourier (`iadftview`), exibição de linhas de nível de uma imagem em níveis de cinza (`iaisolines`), exibição de uma imagem rotulada com cores aleatórias (`ialblshow`), exibição de gráficos (`iaplot`), exibição de imagens (`iashow` e `iagshow`) e exibição de superfícies 3-D (`iasplot`);
- **Aproximações de meio-tom:** *dithering* ordenado (`iadither`) e difusão de erro de Floyd-Steinberg (`iafloyd`);
- **Funções de suporte:** documentação de uma função (`iahelp`), exibição do código-fonte de uma função (`iatype`), exibição de mensagens de erro (`iaerror`) e devolução do conjunto único ordenado, sem repetição, a partir de um vetor ou matriz de entrada (`iaunique`).

A Figura 4.1 mostra o IDLE[1] Python e exemplos simples de uso da caixa de ferramentas implementada.

Dada uma base de dados XML no Adesso, acrescentar o suporte a outra linguagem significa basicamente incluir as informações de sintaxe em trechos onde há código-fonte, além de implementar as folhas de estilo para a interpretação do novo contexto. As demais informações como parâmetros de entrada e saída, descrições gerais das funções e de seus parâmetros, são compartilhadas. Esta característica facilita a manutenção de uma mesma caixa de ferramentas para múltiplas linguagens. Um resultado importante é que as mesmas funcionalidades apresentadas aqui, para a caixa de ferramentas `ia636`, estão disponíveis também para a linguagem MATLAB. Python e

[1] *Integrated DeveLopment Environment*

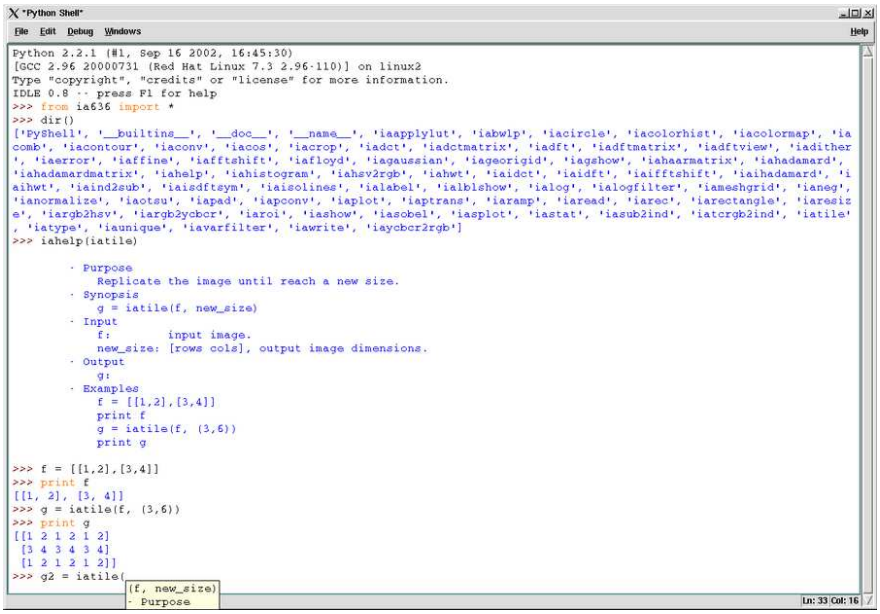


Figura 4.1: Exemplo de uso da caixa de ferramentas no IDLE Python.

MATLAB seguem, via de regra, uma certa sistemática de intercambeamento das sintaxes tornando esta manutenção perfeitamente plausível. A Tabela 4.1 mostra algumas diferenças de sintaxe, de operações usuais em processamento de imagens, entre as linguagens Python e MATLAB.

Toda a caixa de ferramentas, inclusive código-fonte, pode ser obtida a partir da página do projeto do curso de processamento de imagens em Python [13]. Neste site também podem ser conferidos: o programa do curso, os pontos levantados em uma lista de discussão, referências bibliográficas, manual de instalação do ambiente de programação, e todas as atividades recentes entregues pelos alunos.

Operações comuns	Numerical Python	MATLAB
Número complexo	`>>> 1j`	`>> j`
Matriz n-dimensional	`>>> a = zeros((N1,N2,...,Nn))`	`>> a = zeros(N1,N2,...,Nn)`
Matriz quadrada	`>>> a = zeros((N,N))`	`>> a = zeros(N)`
Criação de vetor	`>>> a = zeros(N)`	`>> a = zeros((1,N))`
Linearização de matriz	`>>> ravel(a)`	`>> a(:)`
Dimensão de matriz	`>>> a.shape`	`>> size(a)`
Primeiro elemento	`>>> a[0]`	`>> a(1)`
Último elemento	`>>> a[-1]`	`>> a(end)`
Slicing	`>>> a[inicio:fim+1:passo]`	`>> a(inicio+1:passo:fim+1)`
Retipagem	`>>> a.astype('b')`	`>> uint8(a)`
Redimensionamento	`>>> reshape(a,new_shape)`	`>> resize(a,new_size)`
Transposição	`>>> transpose(a)`	`>> a'`
Multiplicação, potenciação	`>>> a*b, a**b`	`>> a.*b, a.^b`
Multiplicação de matrizes	`>>> matrixmultiply(a,b)`	`>> a*b`
Concatenação de matrizes	`>>> concatenate((a,b))`	`>> [a;b]`

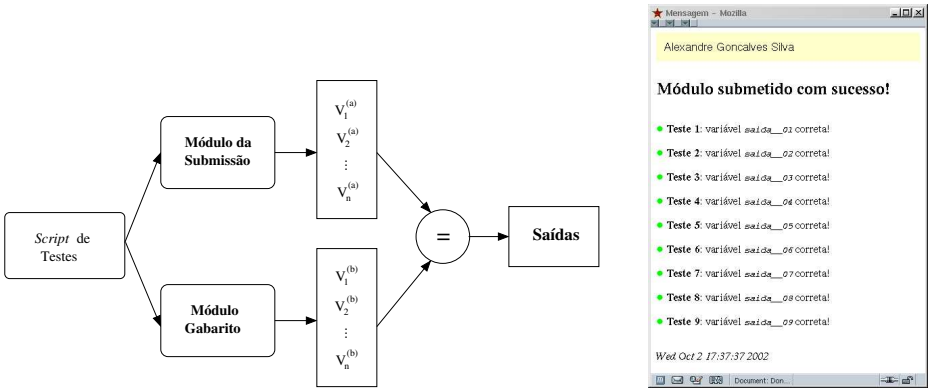
Tabela 4.1: Algumas diferenças de sintaxe entre Numerical Python e MATLAB.

4.2 Corretor Automático

Com auxílio da caixa de ferramentas construída e das funcionalidades nativas do Python para implementação de CGI[2] foi possível desenvolver um sistema para processar entregas de exercícios do alunos, em Python, pela Internet. Em uma página de submissão, é solicitada a entrega de um módulo (um arquivo) com os protótipos de todas as funções precisamente definidos por um enunciado. Optativamente também pode ser entregue um relatório (em formato TXT, PS, PDF ou HTML). Este sistema se divide em duas modalidades:

1. **Correção.** Vários testes, dado por um *script* elaborado pelo professor, são feitos sobre o módulo entregue e os resultados obtidos são comparados com os resultados do processamento do "Módulo Gabarito". A Figura 4.2 ilustra este processo. A caixa com o símbolo "=" se encarrega de comparar todas as n variáveis (n testes) em memória, V_1, V_2, $\ldots V_n$, definidas pelo "*Script* de Testes", entre os resultados da submissão $V_i^{(a)}$ e do gabarito $V_i^{(b)}$, $i \in [1, n]$. As saídas são os resultados

[2] *Common Gateway Interface*



(a) Esquema do corretor automático.

(b) Exemplo de interface das "saídas".

Figura 4.2: Esquema do corretor automático na modalidade de "correção".

das comparações. É exibido, para cada variável, se o valor está correto, incorreto ou se houve falha. O interesse aqui é verificar a exatidão do algoritmo do aluno.

2. **Visualização.** Nesta modalidade, não é necessária a implementação de um gabarito. São feitos testes sobre o módulo submetido, da mesma forma, e uma página HTML, com saídas numéricas ou imagens, é gerada automaticamente e armazenada como resultados de cada aluno, conforme especificações feitas no *script* de testes. A Figura 4.3 ilustra o ambiente de interação inicial de submissão dos exercícios e alguns resultados gerados neste sentido. O interesse aqui poderia ser, entre outros, o de verificar a qualidade ou forma de uma imagem produzida por um algoritmo, um índice de correlação ou de erro.

   Para o professor há ainda uma opção de consulta detalhada dos resultados que pode ser acessada a qualquer tempo (os alunos não têm esta permissão antes do prazo de entrega). Além do código-fonte, relatório,

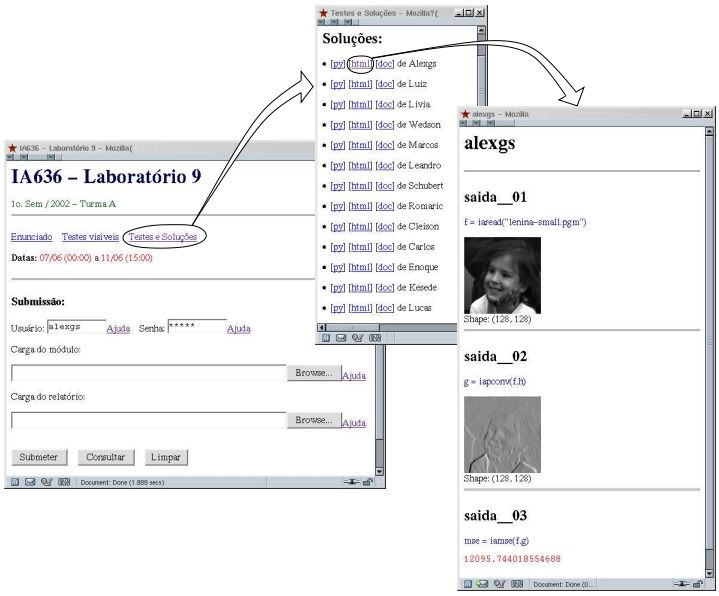


Figura 4.3: Ambiente de submissão e visualização de resultados.

resultados ilustrativos em HTML ou histórico de testes realizados, o professor também pode visualizar os tempos de execução de cada função submetida, aluno por aluno, permitindo, desta forma, uma análise de eficiência das implementações. O sistema ainda permite a re-submissão de todos os módulos entregues, caso se queira estabelecer novos testes, e produção de um arquivo compactado de todos os resultados.

O corretor agregou recentemente o recurso de construção *on line* de *wrappers*, a partir de um *script* Python desenvolvido pelo Rubens C. Machado do CenPRA. Com algumas regras de programação bem documentadas pelo Rubens é possível que os alunos também enviem programas em C. O corretor, neste caso, recebe o código, realiza a compilação, exibe os resultados da mesma e, se não houver erro, cria um módulo com as funções *built-in* do aluno que, a partir deste momento, passam a ser tratadas normalmente em qualquer das modalidades do corretor.

4.3 Avaliação

O ambiente de programação, incluindo o uso da caixa de ferramentas e o sistema de correção automática, foi utilizado no curso de Visão Computacional (IA636) da pós-graduação da Unicamp, sendo que, pela primeira vez, o Prof. Roberto Lotufo experimentou o uso exclusivo da linguagem Python. Diversas atividades foram feitas [13]: (1) Síntese de Imagem em Forma de Xadrez; (2) Ferramentas de Manipulação de Imagens; (3) Síntese de Imagens e Histograma; (4) Síntese e Combinação de Imagens e Modelos de Cor; (5) Transformações Geométricas; (6) Convolução Linear, Verificação de Conjugado Simétrico, Manipulação Geométrica; (7) Síntese de Imagens, Filtro Gaussiano e Propriedades de Transformadas; (8) Transformada de Hotelling; (9) Codificação JPEG; (10) Projeto de Contagem de Laranja. Ao final do curso foi proposto um questionário *on line* de avaliação do ambiente de suporte (sem que uma mesma máquina pudesse responder mais de uma vez). De treze alunos matriculados, doze questionários foram respondidos. Os resultados são mostrados na Tabela 4.2.

Ainda três comentários gerais foram feitos pelos alunos:

(1) "Considero que, por estar em fase de implementação, o sistema de corretor automático ainda precisa de alguns ajustes (embora uma boa parte dos problemas encontrados seja derivada dos próprios problemas da linguagem Python)"

(2) "O ambiente gera uma situação interativa entre alunos e professor, o que torna a cadeira mais dinâmica e mais do que ficar entre quatro paredes de uma sala de aula. O importante é aplicar e conhecer como as coisas funcionam e não só ficar na teoria. Essa é a vantagem desse curso."

(3) "A linguagem python/numeric é uma ferramenta simples para ser utilizada com processamento de imagens, devido a maneira com que ela trabalha com matrizes. O processo de aprendizado da linguagem é fácil e intuitivo."

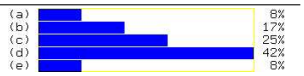
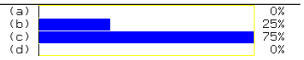
	Perguntas e número de respostas por alternativa	Proporção
1	Quanto tempo aproximadamente você levou para instalar o Python e todos os pacotes necessários ao funcionamento da toolbox IA636? (a) Em poucos minutos: **8**; (b) Em poucas horas: **2**; (c) Em cerca de um dia: **0**; (d) Em cerca de uma semana: **1**; (e) Em mais de uma semana: **1**	
2	Quanto tempo aproximadamente você levou para se habituar à sintaxe Python e Numerical? (a) Menos de duas semanas: **5**; (b) Entre duas semanas e um mês: **5**; (c) Mais de um mês: **2**	
3	Como foi a programação Python/Numeric no início do curso? (a) Muito difícil: **2**; (b) Difícil: **0**; (c) Normal: **7**; (d) Fácil: **1**; (e) Muito fácil: **2**	
4	Como foi a programação Python/Numeric no final do curso? (a) Muito difícil: **1**; (b) Difícil: **2**; (c) Normal: **3**; (d) Fácil: **5**; (e) Muito fácil: **1**	
5	Como você situa o Python/Numeric como ferramenta para um curso de processamento de imagens? (a) Não é adequada: **0**; (b) É pouco adequada: **5**; (c) É adequada: **3**; (d) É muito adequada: **4**	
6	Qual a avaliação geral sobre as implementações da toolbox IA636? (a) Ruins: **0**; (b) Razoáveis: **3**; (c) Boas: **8**; (d) Excelentes: **1**	
7	Qual a sua avaliação geral sobre a toolbox IA636? (a) Ruim: **0**; (b) Razoável: **3**; (c) Boa: **9**; (d) Excelente: **0**	
8	Qual sua avaliação geral sobre o sistema de correção automática? (a) Não é adequado: **0**; (b) É pouco adequado: **4**; (c) É adequado: **3**; (d) É muito adequado: **5**	
9	Qual sua avaliação geral sobre o ambiente de suporte ao curso IA636? (a) Ruim: **1**; (b) Razoável: **0**; (c) Bom: **8**; (d) Excelente: **3**	

Tabela 4.2: Avaliação do ambiente de suporte a processamento de imagens.

Considerando que o corretor foi desenvolvido durante o curso, onde os próprios alunos serviram como testadores do sistema desenvolvido, estes resultados foram considerados bastante satisfatórios.

4.4 Morfologia Matemática

O suporte a criação de API Python/C feito no sistema Adesso (vide Subseção 3.2.4) permitiu que a caixa de ferramentas de Morfologia Matemática [4], integralmente programada em C, pudesse ser diretamente portada em Python. Estas ferramentas já eram utilizadas nas plataformas Khoros, Tcl e MATLAB. Os alunos do curso de Morfologia Matemática (IA870) da pós-graduação da Unicamp, segundo semestre de 2002, passaram a usá-la também em Python. A vantagem mais imediata talvez venha da agilidade proporcionanda pelo ambiente de suporte a processamento de imagens implementado, ou da flexibilidade da própria linguagem. Em Tcl, por exemplo, qualquer operação matricial não prevista pela caixa de ferramentas, necessita ser implementada em C. MATLAB, apesar de excelente ao que se propõe, é fechada e dificulta a implementação de sistemas, por exemplo, para Internet. Com Khoros, há complicações parecidas.

Para ilustrar o funcionamento do `morph`, nome dado ao módulo principal API gerado, é descrito na Figura 4.4 uma aplicação de segmentação de microestruturas do concreto utilizado na construção civil. A imagem original de microscopia eletrônica por varredura (SEM) e algumas das saídas da função `concrete` apresentada podem ser visualizadas à direita. São estas: (i) anidros - compostos do cimento não hidratados; (ii) agregados - outros compostos não pertecentes ao cimento como como areia, pedra ou escória; e (iii) compostos hidratados do cimento. Esta implementação resultou em um artigo aceito no SIBGRAPI 2002 [25]. As funções de morfologia utilizadas são: (1) `mmhistogram` - cálculo eficiente de histograma de uma imagem; (2) `mmasf` - filtragem alternada sequencial; (3) `mmneg` - negativo de uma ima-

```
def concrete(file='concrete.pgm', hm=20):
    from Numeric import nonzero, ravel, sum
    from ia636   import iaread, iashow
    from morph   import mmhistogram, mmasf, mmneg, mmseline,\
                 mmwatershed, mmbinary, mmintersec, mmframe,\
                 mmthreshad, mmareaopen, mmgradm, mmregmin,\
                 mmhbasin, mmcwatershed, mmareaclose, mmsubm
    f  = iaread(file)

    # Segmentacao dos Anidros
    hf   = mmhistogram(f)
    hasf = mmasf(mmneg(hf),'co',mmseline(8,90))
    ws   = mmwatershed(hasf)
    wsf  = mmintersec(mmneg(mmbinary(-mmframe(ws,0,20))), ws)
    kth  = nonzero(ravel(wsf))
    aux  = mmthreshad(f,kth,255)
    anhy = mmareaopen(aux,20)

    # Segmentacao dos Agregados
    grad = mmgradm(f)
    ma   = mmregmin(mmhbasin(grad,hm))
    wsa  = mmcwatershed(grad,ma)
    a1   = mmareaopen(mmneg(wsa),300)
    a2   = mmareaclose(a1,50)
    aggr = mmsubm(a2,anhy)

    # Componente Hidratado
    hydcomp = mmsubm(f,a2)

    # Fracao de Area
    (h,w) = f.shape
    aanhy = 1.*sum(ravel(anhy))/(w*h)
    aaggr = 1.*sum(ravel(aggr))/(w*h)

    return (anhy, aanhy, aggr, aaggr, hydcomp)
```

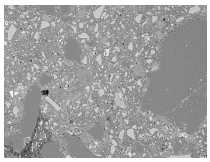
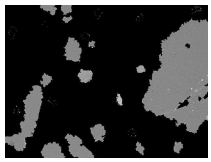
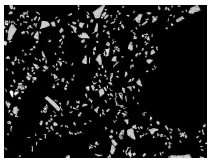


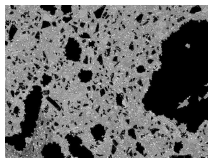
Imagem original (f)



Agregados (aggr)



Anidros (anhy)



Compostos hidratados (hydcomp)

Figura 4.4: Aplicação de segmentação de microestruturas do concreto.

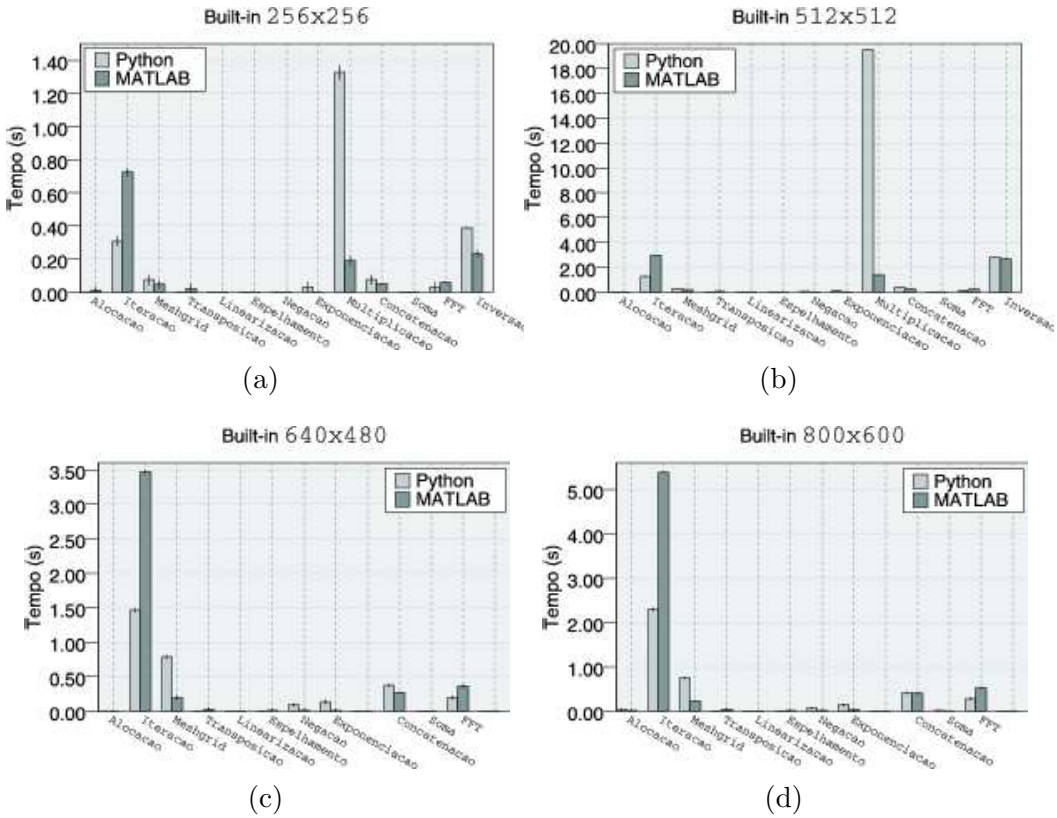
gem; (4) `mmseline` - criação de elemento estruturante em forma de linha; (5) `mmwatershed` - detecção de linhas divisoras de água [5, 28]; (6) `mmbinary` - conversão de uma imagem em níveis de cinza em uma imagem binária; (7) `mmintersec` - intersecção de imagens; (8) `mmframe` - criação de um quadro ao redor de uma imagem; (9) `mmthreshad` - *thresholding* adaptativo; (10) `mmareaopen` - abertura por área; (11) `mmgradm` - gradiente morfológico; (12) `mmregmin` - mínimo regional; (13) `mmhbasin` - remoção de bacias com contraste menor que um certo valor; (14) `mmcwatershed` - detecção de linhas divisoras de água por marcadores; (15) `mmareaclose` - fechamento por área; (16) `mmsubm` - subtração de duas imagens com saturação.

4.5 Python x MATLAB

Nesta seção, são feitos alguns testes em que são medidos os tempos médios de execução (em um Athlon XP 1.4GHz), sob mesmas condições, de funções diretas *built-in* e também da caixa de ferramentas implementada `ia636`. Cada teste é repetido sobre imagens diferentes imagens, de 256x256, 512x512,

640x480 e 800x600. A Figura 4.5 se refere aos tempos calculados para as seguintes funções *built-in*: alocação de memória, iteração para construção de uma imagem sintética, uso de malha de índices também para construção da mesma imagem sintética, transposição matricial, linearização, espelhamento na horizontal e vertical, negação (255 menos os valores da imagem criada), potenciação de todos elementos da imagem, multiplicação de matrizes, concatenação na horizontal e vertical (aumento de 4 vezes), soma de todos os elementos da matriz linearizada, cálculo da Transformada Rápida de Fourier (FFT), e inversão de matriz. Para imagens de 640x480 e 800x600 não se aplica a multiplicação ou inversão de matrizes entre si. Observa-se que, como já foi dito, o uso de iterações sobre estruturas matriciais é mais eficiente em Python, sendo, em geral, menos da metade do tempo obtido em MATLAB. Outra consideração a fazer é o desempenho da multiplicação de matrizes. Neste caso, a implementação do Numerical é bastante ineficiente em relação a do MATLAB. Os demais testes se mostram equilibrados, bastante rápidos e, exceto na inversão de matriz, sempre inferiores a 1s de execução.

A Figura 4.6 compara os tempos de execução de algumas das funções `ia636` implementadas. Os testes mais críticos aqui são `iadct` e `iadft` que se utilizam de multiplicação de matrizes, ponto fraco da implementação do Numerical vista anteriormente. Para o cálculo da DFT existe a opção imediata *built-in* bastante eficiente em ambas linguagens. No caso da DCT, o ideal seria a implementação do algoritmo e *wrapper* em C. Outra função em evidência é a `iaotsu`. Nesta é preciso calcular o histograma da imagem, implementação crítica em linguagens interpretadas por necessitar de uso de comandos de iteração. Outra função bastante iterativa é a `ialabel`. Sua implementação em Python, no entanto, utilizou-se, na maior parte do código, de estruturas nativas do Python, demonstrando-se bastante eficiente (destaque à diferença obtida pelo `ialabel` na Figura 4.6(d)). As demais funções apresentam execuções relativamente eficientes tanto para Python quanto para MATLAB.



(a) (b)

(c) (d)

Figura 4.5: Comparações de desempenho de funções *built-in* entre Python e MATLAB.

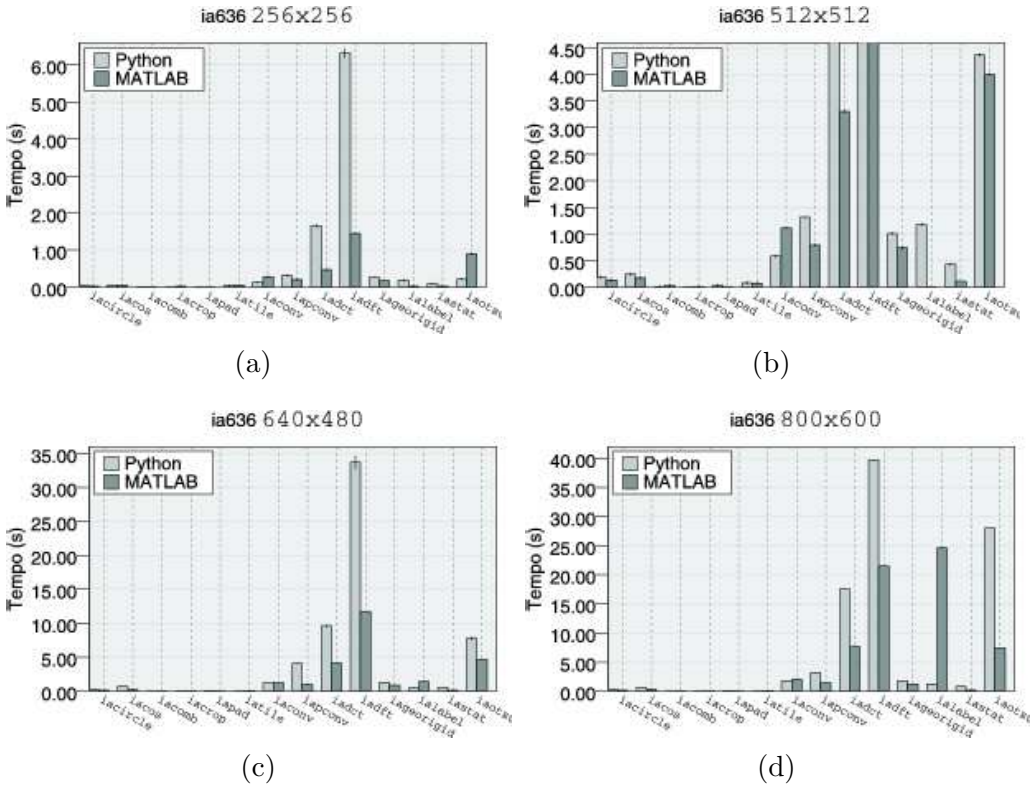


Figura 4.6: Comparações de desempenho de funções `ia636` entre Python e MATLAB.

A conclusão imediata deste trabalho é que a implementação nativa do Python, em termos de eficiência, é superior a do MATLAB. Já a implementação dos módulos do pacote Numerical é, em geral, inferior a do MATLAB. A equipe de desenvolvimento do Numerical está reescrevendo o código e incluindo mais funções em baixo nível com o propósito de que o pacote, agora chamado de *numarray*, mais otimizado, passe a integrar futuras distribuições Python. Enquanto isto, vale citar alguns detalhes que comprometem ou melhoram o desempenho da implementação de *scripts* pelo usuário. Por exemplo, para os parâmetros de entrada, é comum que se aceite listas do Python ou *arrays* do Numeric e, independentemente da entrada, se faça a retipagem para *array*. Isto é custoso para imagens grandes. Aconselha-se usar a função `Numeric.asarray` em vez de `Numeric.array` pois, neste caso, não há alocação de mais memória caso a entrada já seja um *array*. Em MATLAB, tal problema não se aplica pois é utilizado apenas um tipo de estrutura. Outra questão, que ajuda consideravelmente no desempenho do Numerical, é a utilização de referência a elemento da matriz na forma A[i][j][k] em vez de A[i,j,k]. As funções numéricas (seno, tangente, logaritmo, potenciação, ...) do módulo nativo `math` do Python também são mais eficientes que as do módulo `Numeric`, porém não podem ser aplicadas a conjunto de dados, a menos que sejam usadas iterações ou comandos como `map` (implementa iteração implícita). O tratamento usual de erros e exceções é outra característica que faz com que o desempenho em Python fique comprometido em relação ao MATLAB mas, por outro lado, aumenta a flexibilidade do código. Todas estas considerações, no entanto, perdem sentido quando há uma mesma implementação em baixo nível para Python e MATLAB. É o caso da caixa de ferramentas de morfologia matemática. Para *wrappers* a eficiência é comparável ao assembly da máquina.

Capítulo 5

Considerações Finais

5.1 Conclusões

Python é uma linguagem genérica podendo, portanto, ser usada em vários domínios de aplicação. Existem diversos módulos que vêm juntos à sua distribuição (de funções matemáticas, de manipulação de *strings* e tempo, de utilização de sistemas operacionais e de arquivos, de compactação, criptografia, de suporte SGML, CGI, HTTP, FTP, Telnet, apenas para citar alguns) e inúmeros outros que podem ser incorporados. Como exemplo, pode-se destacar o poderoso PyOpenGL (interface encapsulada da implementação OpenGL). Este módulo vem sendo utilizado com sucesso por grande parte do alunos do curso de pós-graduação de Computação Gráfica da FEEC-Unicamp. O pacote Numerical, em especial, se adapta perfeitamente ao propósito deste trabalho que é o de manipulação matricial de grandes conjuntos de dados, de forma eficiente, para o processamento de imagens.

O sistema Adesso separa nitidamente conteúdo de apresentação simplificando consideravelmente o esforço de manutenção e gerenciamento da informação de uma caixa de ferramentas, um processo bastante trabalhoso devido ao elevado número de componentes (que tende a aumentar). A adoção da metodologia de estruturar a informação e usar geração de código e docu-

mentos de forma automatizada traz inúmeros benefícios sendo os principais: diminuição do número de linhas de informação mestre, responsável tanto pelo programa como sua documentação geradas, aumentando imunidade a erros (tipicamente introduzidos em situações de "cópia e cola"), diminuindo espaço de armazenamento de cópias de segurança; consistência na interface com o usuário, visto que o código resultante é gerado automaticamente, sendo inerentemente sistemático por todo o sistema; informação facilmente adaptável para gerar código e documentação de acordo com os avanços tecnológicos exigidos pelo mercado; maior imunidade a erros também na criação de um empacotamento e distribuição do aplicativo, permitindo uma consistência na configuração desejada.

A caixa de ferramentas desenvolvida é capaz de suprir boa parte das necessidades básicas quando na implementação de um exercício de processamento de imagens. A grande quantidade de exemplos da documentação gerada auxilia fortemente o aprendizado da linguagem Python em si e, principalmente, dos algoritmos tradicionais de tratamento de imagens. As codificações utilizando tanto o Numerical Python quanto o MATLAB revelam-se, em geral, implementações diretas das fórmulas estudadas nos livros.

O corretor automático mostrou-se uma ferramenta eficaz no auxílio à avaliação de exercícios de programação, além de permitir a visualização e comparação das soluções pelos códigos entregues e resultados gerados.

A principal vantagem do Python sobre o Tcl é prover um sistema eficiente direto de manipulação de matrizes. Em Tcl qualquer operação neste sentido necessita de implementações em C. Python também permite que todo o trabalho feito possa ser utilizado de diversas maneiras, desde aplicações para Internet como em sistemas *stand alone* (com chamadas ao interpretador embutidas). Com MATLAB, tarefas assim, em geral, não têm suporte explícito e dependem dos termos de registro do *software* para serem realizadas. Linguagens como Octave [8] e Scilab [6] que, à primeira vista, poderiam ser opções imediatas, gratuitas e compatíveis ao MATLAB, demonstram-se bas-

tante dedicadas a problemas de computação científica, são menos flexíveis em relação ao Python, e suas comunidades não apresentam uma atividade tão intensa de desenvolvimento como a observada nas atualizações, melhoramentos constantes e inclusão de novas funcionalidades da linguagem Python. Por ser baseado em Python, o ambiente de suporte a processamento de imagens implementado é totalmente disponibilizado de forma gratuita e modificações de qualquer espécie podem ser feitas a qualquer momento conforme a necessidade. O fácil acesso às instalações, a flexibilidade de programação em função do desempenho requerido, os resultados da aplicação prática sentido na exposição de opiniões pelos alunos, indicam que o presente trabalho passou de fato a ser uma ferramenta auxiliar no aprendizado de processamento de imagens, permitindo uma visualização rápida dos resultados esperados teoricamente. O mesmo pode perfeitamente ser utilizado em atividades de pesquisa e construção de aplicações finais.

5.2 Trabalhos Futuros

Como trabalho futuro, poderíamos citar a criação de folhas de estilo, dentro do sistema Adesso, para facilitar a geração de páginas dinâmicas na Internet. Poder-se-ia ter um ambiente *on line* de testes de funções, onde, a priori, é desnecessária a instalação da caixa de ferramentas (o interpretador seria transparente do lado do servidor). A Figura 5.1 exibe um protótipo neste sentido construído manualmente para o cálculo do filtro de média, onde o aluno pode escolher a imagem de entrada e a dimensão da máscara. A saída é então visualizada ao lado da imagem original. Todo este processo depende simplesmente de um navegador de Internet. O formulário com as opções de escolha poderia ser montado conforme a documentação de tipos registrada na base de dados XML.

Um trabalho possível seria o de implementar um sistema de programação visual similar ao Khoros ou Simulink. Assim como na descrição encadeada de

Figura 5.1: Cálculo *on line* do filtro de média usando CGI/Python e a caixa de ferramentas `ia636`.

instruções em um *script*, ter-se-ia agora um encadeamento de componentes gráficos (*drag and drop*), cada um com as devidas entradas e saídas. Todas as informações necessárias neste sistema estão disponíveis nos documentos XML. Um *parser* XML para Python, associado a uma interface GUI Tkinter ou wxPython, poderia ser usado com este propósito. Quando se trata de interfaces gráficas, Java é uma ferramenta bastante requisitada. E `applets` Java permitem o uso destes recursos gráficos também remotamente (pela Internet). A exploração da linguagem Jython[1] seria interessante por agregar o uso de módulos Python normalmente e as vantagens do uso de interfaces em Java.

Outro trabalho que vem sendo desenvolvido pelo Guilherme S. Mazzela [11] é o de portar, em Python, a aplicação Prontovideo [16], até então programada em Tcl. O benefício principal seria aquele que foi comentado no final da seção anterior.

[1] `http://www.jython.org`

Referências Bibliográficas

[1] A. C. R. Almeida, A. A. S. Sol e A. A. Araújo. PhotoPixJ: uma Plataforma para Processamento Digital de Imagens em Java. In *Proceedings of the SIBGRAPI*, Rio de Janeiro, Brazil, October 1998. IEEE.

[2] A. G. Silva, R. A. Lotufo e R. C. Machado. Ambiente de Suporte ao Ensino de Processamento de Imagens Usando a Linguagem Python. In *XIII SBIE*, pages 39–48, São Leopoldo, Brasil, Novembro 2002. Unisinos.

[3] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. *Numerical Python*. September 2001. http://www.pfdubois.com/numpy/html2/numpy.html.

[4] J. Barrera, G. J. F. Banon, R. A. Lotufo, and R. Hirata Jr. MMach: a Mathematical Morphology Toolbox for the Khoros System. *Journal of Electronic Imaging*, 7(1):174–210, 1998.

[5] S. Beucher and F. Meyer. *Mathematical Morphology in Image Processing*, chapter 12. The Morphological Approach to Segmentation: The Watershed Transformation, pages 433–481. Marcel Dekker, 1992.

[6] J.-P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, and S. Steer. *Introduction à SCILAB*. Springer, 2002.

[7] R. C. Machado e R. A. Lotufo. Adesso: Ambiente Computacional para Desenvolvimento Rápido de Aplicações. Technical report, DCA-FEEC-Unicamp/FCTI, Campinas, Brasil, Maio 2000.

[8] J. W. Eaton. *GNU Octave, a high-level interactive language for numerical computations*. 3^{rd} edition, February 1997. http://www.octave.org/doc/octave_toc.html.

[9] S. L. Eddins and M. T. Orchard. Using MATLAB and C in an Image Processing Lab Course. In *Proceedings of ICIP-94*, pages 515–519, Austin, USA, November 1994.

[10] J. L. Fontaine. *tclpython, a Python package for Tcl*. http://jfontain.free.fr/tclpython.htm.

[11] G. S. Mazzela e R. A. Lotufo. Ferramenta de Segmentação Interativa de Imagens e Vídeo Usando Python-Numeric-Tk. In *X Congresso Interno de Iniciação Científica da Unicamp*, Campinas, 2002.

[12] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison-Wesley Publishing Company, 1992.

[13] ia636. Course of Image Processing in Python. 2002. http://www.dca.fee.unicamp.br/~alexgs/curso.

[14] L. Jochen. tDOM - A fast XML/DOM/XPath package for Tcl written in C. 1999. http://sdf.lonestar.org/~loewerj/tdom.cgi.

[15] R. Jordan and R. A. Lotufo. Interactive Digital Imagem Processing Course on the World-Wide Web. In *Proceedings of the 1996 International Conference on Image Processing*, pages 433–436, Lausanne, Switzerland, September 1996. IEEE Signal Processing Society.

[16] R. Lotufo, R. Machado, F. Flores, A. Falcão, R. Koo, G. Mazzela, and R. Costa. Prontovideo - An Image Sequence Segmentation Tool Applied to Video Edition. In *Proceedings of XIV SIBGRAPI*, page 404, Florianópolis, Brazil, October 2001. IEEE Signal Processing Society.

[17] R. A. Lotufo and R. Jordan. Hands-On Digital Image Processing. In *IEEE Fontiers in Education – 26th Annual Conference*, pages 1199–1202, Salt Lake City, USA, November 1996. FIE-96.

[18] F. Lundh and M. Ellis. *Python Imaging Library Handbook*. March 2002. http://www.pythonware.com/products/pil/pil-handbook.pdf.

[19] R. M. S. Luppi, D. M. Kligerman, A. X. Falcão, U. M. B. Neto, and R. A. Lotufo. V3DTOOLS: A KHOROS Toolbox for 3D Imaging. In *Word Congress on Medical Physics and Biomedical Engineering*, Rio de Janeiro, Brazil, August 1994.

[20] M. Lutz and D. Ascher. *Learning Python*. O'Reilly & Associates, April 1998.

[21] R. C. Machado. Adesso: Ambiente para Desenvolvimento de Software Científico. Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação - Universidade Estadual de Campinas, Junho 2002.

[22] L. Prechelt. An Empirical Comparison of Seven Programming Languages. *Computing Pratices IEEE*, pages 23–29, October 2000.

[23] J. A. Robinson. A Software System for Laboratory Experiments in Image Processing. *IEEE Transactions on Education*, 43:455–459, November 2000.

[24] G. Rossum. *Python Tutorial*. 2002. http://www.python.org/doc/current/tut/tut.html.

[25] A. G. Silva, R. A. Lotufo, and F. C. Flores. Classification of Microstructures by Morphological Analysis and Estimation of the Hydration Degree of Cement Past in Concrete. In *Proceedings of XV SIBGRAPI*, pages 138–145, Fortaleza, Brazil, October 2002. IEEE Signal Processing Society.

[26] A. G. Silva, R. A. Lotufo, and R. C. Machado. Toolbox of Image Processing for Numerical Python. In *Proceedings of XIV SIBGRAPI*, page 402, Florianópolis, Brazil, October 2001. IEEE Signal Processing Society.

[27] SDC Information Systems. SDC Morphology Toolbox for MATLAB. 1998. http://www.mmorph.com.

[28] L. Vincent and P. Soille. Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598, June 1991.

[29] W3C Recommendation 16 November 1999. XML Path Language (XPath) 1.0. 1999. http://www.w3.org/TR/xpath.

[30] W3C Recommendation 16 October 1998. Document Object Model (DOM) Level 1 Specification Version 1.0. 1998. http://www.w3.org/TR/REC-DOM-Level-1.

[31] W3C Recommendation 6 October 2000. Extensible Markup Language (XML) 1.0. 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[32] W3C Working Draft 27 March 2000. Extensible Stylesheet Language (XSL) 1.0. 2000. http://www.w3.org/TR/xsl.

[33] N. Walsh and L. Muellner. *DocBook: The Definitive Guide*. O'Reilly & Associates, 1999.

[34] B. B. Welch. *Pratical Programming in Tcl and Tk*. Prentice Hall PTR, 3^{rd} edition, 1999.

[35] T. Williams and C. Kelley. *gnuplot, An Interactive Plotting Program*. December 1998. http://www.ucc.ie/gnuplot/gnuplot.html.

Trabalhos Publicados

[1] A. G. Silva, R. A. Lotufo e R. C. Machado. Ambiente de Suporte ao Ensino de Processamento de Imagens Usando a Linguagem Python. In *XIII SBIE*, pages 39-48, São Leopoldo, Brasil, Novembro 2002. Unisinos.

[2] A. G. Silva, R. A. Lotufo, and R. C. Machado, Toolbox of Image Processing for Numerical Python. In *Proceedings of XIV SIBGRAPI*, page 402, Florianópolis, October 2001. IEEE Signal Processing Society.

[3] A. G. Silva, R. A. Lotufo, and F. C. Flores. Classification of Microstructures by Morphological Analysis and Estimation of the Hydration Degree of Cement Past in Concrete. In *Proceedings of XV SIBGRAPI*, pages 138-145, Fortaleza, Brazil, October 2002. IEEE Signal Processing Society.

[4] F. C. Flores, R. A. Lotufo, S. Isernhagen, L. M. Rocha, A. G. Silva, and E. F. Rocha. Method for Assessment of Telangiectasia Degreeing by Mathematical Morphology. In *Proceedings of XV SIBGRAPI*, pages 153-160, Fortaleza, Brazil, October 2002. IEEE Signal Processing Society.

Apêndice A

Implementações de MSE

Segue a listagem dos programas usados na determinação do MSE (Mean Square Error ou Erro Médio Quadrático) entre duas imagens f_1 e f_2, para as linguagens C, Java, Python e MATLAB. Os tempos de execução de cada função, dada uma imagem colorida 256x256 pixels, foram calculados e os resultados mostrados na Tabela 2.1 para comparação.

A.1 Em C

```
double mse(image f1, image f2) {
    int h = f1.h;
    int w = f1.w;
    int i;
    double MSE = 0;
    for (i=0; i < 3*h*w; i++) {
        MSE = MSE + pow((f1.data[i] - f2.data[i]), 2);
    }
    MSE = MSE / (3*h*w);
    return MSE;
}
```

A.2 Em Java

```
public double mse(int f1[], int f2[], int h, int w) {
```

```
    double MSE=0;
    for (int i=0; i < h*w; i++) {
        int R1, R2, G1, G2, B1, B2;
        B1 =  f1[i]      & 255; B2 =  f2[i]      & 255;
        G1 = (f1[i]>>8)  & 255; G2 = (f2[i]>>8)  & 255;
        R1 = (f1[i]>>16) & 255; R2 = (f2[i]>>16) & 255;
        MSE = MSE + Math.pow((R1-R2),2) + Math.pow((G1-G2),2) + Math.pow((B1-B2),2);
    }
    MSE = MSE / (3*h*w);
    return MSE;
}
```

A.3 Em Python

```
def mse_1(f1, f2):
    MSE = 0.
    for i in range(f1.shape[0]):
        for j in range(f1.shape[1]):
            MSE = MSE + (f1[i,j] - f2[i,j])**2
    MSE = MSE / (3*f1.shape[0]*f1.shape[1])
    return MSE

def mse_2(f1, f2):
    from Numeric import ravel, sum, product
    MSE = sum((ravel(f1)-ravel(f2))**2.)/(3*product(f1.shape[:2]))
    return MSE
```

A.4 Em MATLAB

```
function MSE = mse_1(f1, f2)
    MSE = 0;
    for i = 1:size(f1,1)
        for j = 1:size(f1,2)
            for k = 1:3
                MSE = MSE + (f1(i,j,k) - f2(i,j,k))^2;
            end
        end
    end
    MSE = MSE / (3*size(f1,1)*size(f1,2));

function MSE = mse_2(f1, f2)
    aux = size(f1);
    MSE = sum((f1(:)-f2(:)).^2)/(3*prod(aux(1:2)));
```

Apêndice B

Exemplo de Folha de Estilo

A seguir, é dado um exemplo de uma folha de estilo implementada para geração de código-fonte Python puro.

```
stylesheet python.pycode:

        template AdToolbox.begin:
                [sty:par release check all]
                [sty:logIt "------------------"]
                [sty:logIt "Processing toolbox '[att name]' with pycode stylesheet for PYTHON"]
                [sty:save [att name].py {[pythonify [sty:apply . module]]}]
                [sty:save [att name]demo.py {[pythonify [sty:apply . demo]]}]
                [sty:save [att name]test.py {[sty:apply . suite]}]

        template AdToolbox.module:
                """
                        Module [att name] -- [sty:value @title]
                        ---------------------------------------------------------------------
                        [sty:foreach Documentation -trim "\n\t " {
                                [formatPara [sty:apply * ascii] 76 \t]
                        }]
                        ---------------------------------------------------------------------
                        [sty:foreach [_sortNodes [concat [sty:compiled exposedTbxCFuns] [sty:compiled exposedTbxPyFuns]] @_id]
                        -trim "\n\t " -connect "\n\t" {
                                [format "%-[sty:value /AdToolbox/@_maxlen]s" [sty:call fun python]]
                                -- [formatPara1 "[sty:value Short]" [expr 80 - [sty:value /AdToolbox/@_maxlen]
                                - 8] 0 [expr [sty:value /AdToolbox/@_maxlen] + 8]]
                        }]
                """
                #
                __version__ = [att major].[att minor]
                #
                [sty:iff {[llength [sty:compiled exposedTbxCFuns]] > 0} {from _[att name] import *}]
                #
                [sty:apply [sty:compiled exposedTbxPyFuns] def -trim "\n\t " -connect \n]
```

```
        #
        #
        # ====================================================================
        #  Adesso -- Generated [clock format [clock seconds]]
        # ====================================================================
        #

------------------------------------------------------------------
        function defs
------------------------------------------------------------------

template AdFunction.def:
        [sty:if {string(Source[@lang="python"])} {
        # ====================================================================
        #
        #        Global statements for [sty:call fun python]
        #
        # ====================================================================
        [sty:value {Source[@lang="python"]}]
        }]
        # ====================================================================
        #
        #        [sty:call fun python]
        #
        # ====================================================================
        def [sty:call fun python]([sty:apply Args/Arg args -connect ", "]):
                [tabify [sty:call pyfun docstring] \t]
                [tabify [sty:apply Dependencies/Module {} -connect \n] \t]
                [tabify [sty:value {Source[@lang="python"]/Code}] \t]
                [sty:if Return {return [sty:foreach Return -connect ", " {[join [att name] ", "]}]}]
        #

template Args/Arg.args:
        [sty:if @optional="yes" {[att name]=[fixDefault [att default]]} else {[att name]}]

template Module:
        [sty:if .="" {import [att name] \n} else {from [att name] import [join [sty:value .] ", "]}]

template PYcode.examples:
        #
        [sty:iff {[llength [sty:nodelist]] > 1} {
        #        example [sty:index + 1]
        #
        }]
        [sty:apply * ascii]

macro python.demodocstring:
        from [sty:value /AdToolbox/@name] import *
        print
        print '''[formatPara [sty:value Short] 68 ""]'''
        print
        print

template AdToolbox.demo:
        [sty:apply [sty:compiled exposedTbxPyDemos] demo -trim "\n\t " -connect \n]

template AdDemo.demo:
        # ========================================================================
        #
        #        [sty:call fun python]
        #
        # ========================================================================
```

```
        def [sty:call fun python]():
                [tabify [sty:call python demodocstring] \t]
                [tabify [sty:apply Source/Slide demo] \t]
                [tabify "return" \t]

template Slide.demo:
        #
        print '========================================================================='
        print '''\n[formatPara [sty:value Descr] 68 ""]\n'''
        print '========================================================================='
        #[regsub -all "showfig" [sty:value {Code[@lang="python"]}] "#showfig" aux]
        print '''\n$aux'''
        $aux
        print
        raw_input(4*' '+'Please press return to continue...')
        print
        print
        #

template AdToolbox.suite:
        import Numeric
        from [sty:value @name] import *
        print
        print '''Testsuite [att name] -- [sty:value @title]'''
        [sty:apply [sty:compiled exposedTbxPySuites] suite -trim "\n\t " -connect \n]

template AdFunction.suite:
        print
        print '========================================================================='
        print '''[formatPara "[sty:value @name] - [sty:value Short]" 68 ""]'''
        print '========================================================================='
        [sty:foreach Testsuite {[sty:value {Code[@lang="python"]}]}]

# ================================================================
#                                      Support Tcl Procedures
# ================================================================

:adesso:

        proc tabify {txt pre} {
                regsub -all {\n+} $txt "\n$pre" txt
                return $txt
        }
        proc pythonify {txt} {
                set txt [textutil::untabify2 $txt 4]
                regsub -all {\n\s*\n} $txt "\n" txt
                return $txt
        }
        proc fixDefault {txt} {
                switch -exact -- $txt {
                        NULL {
                                set txt None
                        }
                }
                return $txt
        }
```

Apêndice C

Functions

C.1 Image Creation

C.1.1 iabwlp

Purpose: *Low-Pass Butterworth frequency filter.*

Synopsis: `H = iabwlp(fsize, tc, n, option='circle')`

Input:

`fsize`	*<type 'array'> Filter size: a col vector: first element: rows, second: cols, etc. uses same convention as the return of size.*
`tc`	*<type 'float'> Cutoff period*
`n`	*<type 'float'> filter order*
`option`	*<type 'str'> 'circle' or 'square'*

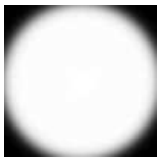
Output:

`H`	*<type 'array'> DFT mask filter, with H(0,0) as (u,v)=(0,0)*

Description: *This function generates a frequency domain Low Pass Butterworth Filter with cutoff period tc and order n . At the cutoff period the filter amplitude is about 0.7 of the amplitude at H(0,0). This function returns the mask filter with H(0,0). As the larger the filter order, sharper will be the amplitude transition at cutoff period. The minimum cutoff period is always 2 pixels, despite of the size of the frequency filter.*

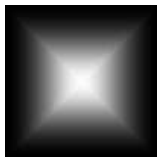
Examples:

```
>>> H2_10 = iabwlp([100,100],2,10) # cutoff period: 2 pixels, order: 10
>>> iashow(iadftview(H2_10))
>>> H4_1 = iabwlp([100,100],4,1) # cutoff period: 4, order: 1
>>> iashow(iadftview(H4_1))
>>> H8_100 = iabwlp([100,100],8,100) # cutoff period: 8, order: 100
>>> iashow(iadftview(H8_100))
>>> H4_1box = iabwlp([100,100],4,1,'square') # cutoff period: 4, order: 1
>>> iashow(iadftview(H4_1box))
```

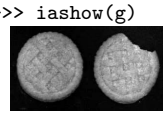

(iadftview(H2_10))

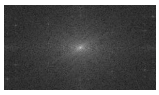

(iadftview(H4_1))


(iadftview(H8_100))

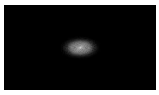

(iadftview(H4_1box))

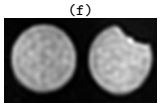
```
>>> import Numeric
>>> import FFT
>>> f = iaread('cookies.pgm')
>>> iashow(f)
>>> F = FFT.fft2d(f)
>>> iashow(iadftview(F))
>>> H = iabwlp(F.shape,16,6)
>>> iashow(iadftview(H))
>>> G = F * H
>>> iashow(iadftview(G))
>>> g = FFT.inverse_fft2d(G)
>>> g = abs(g).astype(Numeric.UnsignedInt8)
>>> iashow(g)
```


(f)


(iadftview(F))


(iadftview(H))


(iadftview(G))


(g)

Equation:

$$
\begin{aligned}
H(u,v) &= \frac{1}{1+(\sqrt{2}-1)(\sqrt{(\frac{u}{N})^2+(\frac{v}{M})^2}\ \ t_c)^{2n}} \\
(u,v) &\in \{-\lfloor\frac{N}{2}\rfloor : N-\lfloor\frac{N}{2}\rfloor-1, -\lfloor\frac{M}{2}\rfloor : M-\lfloor\frac{M}{2}\rfloor-1\} \\
t_c &\in \{2 : \max\{N,M\}\}
\end{aligned}
$$

C.1.2 iacircle

Purpose: *Create a binary circle image.*

Synopsis: `g = iacircle(s, r, c)`

Input:

`s`	*<type 'array'> [rows cols], output image dimensions.*
`r`	*<type 'float'> radius.*
`c`	*<type 'array'> [row0 col0], center of the circle.*

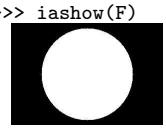
Output:

`g`	*<type 'array'>*

Description: *Creates a binary image with dimensions given by s, radius given by r and center given by c. The pixels inside the circle are one and outside zero.*

Examples:

```
>>> F = iacircle([5,7], 2, [2,3])
>>> print F
[[0 0 0 1 0 0 0]
 [0 0 1 1 1 0 0]
 [0 1 1 1 1 1 0]
 [0 0 1 1 1 0 0]
 [0 0 0 1 0 0 0]]
>>> F = iacircle([200,300], 90, [100,150])
>>> iashow(F)
```



(F)

Equation:

$$g(x,y) = (x - x_c)^2 + (y - y_c)^2 \leq r^2$$

See also: *iarectangle* (C.1.8)

C.1.3 iacomb

Purpose: *Create a grid of impulses image.*

Synopsis: `g = iacomb(s, delta, offset)`

Input:

`s`	*<type 'array'> output image dimensions (1-D, 2-D or 3-D).*
`delta`	*<type 'array'> interval between the impulses in each dimension (1-D, 2-D or 3-D).*
`offset`	*<type 'array'> offset in each dimension (1-D, 2-D or 3-D).*

Output:

`g`	*<type 'array'>*

Examples:

```
>>> u1 = iacomb(10, 3, 2)
>>> print u1
[0 0 1 0 0 1 0 0 1 0]
>>> u2 = iacomb((7,9), (3,4), (3,2))
>>> print u2
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0]]
>>> u3 = iacomb((7,9,4), (3,4,2), (2,2,1))
>>> print u3[:,:,0]
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
```

```
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
>>> print u3[:,:,1]
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0]]
>>> print u3[:,:,2]
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
>>> print u3[:,:,3]
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0]]
```

Equation:

$$u(x) = \sum_{i=0}^{N-1} \delta\left(x - (ki + o)\right)$$
$$x \in [0, D-1]$$

$$u\left(x_1, x_2, \cdots, x_n\right) = \sum_{i_1=0}^{N_1-1} \sum_{i_2=0}^{N_2-1} \cdots \sum_{i_n=0}^{N_n-1} \delta\left(x_1 - (k_1 i_1 + o_1), x_2 - (k_2 i_2 + o_2), \cdots, x_n - (k_n i_n + o_n)\right)$$
$$\left(x_1, x_2, \cdots, x_n\right) \in \left[(0, 0, \cdots, 0), (D_1 - 1, D_2 - 1, \cdots, D_3 - 1)\right]$$

$$where \quad \delta(i) = \begin{cases} 1, & i = 0 \\ 0, & otherwise \end{cases}$$

See also: *iaptrans* (C.6.3)

C.1.4 iacos

Purpose: *Create a cossenoidal image.*

Synopsis: `f = iacos(s, t, theta, phi)`

Input:

s	*<type 'array'> size: [rows cols].*
t	*<type 'array'> Period: in pixels.*
theta	*<type 'float'> spatial direction of the wave, in radians. 0 is a wave on the horizontal direction.*
phi	*<type 'float'> Phase*

Output:

f	*<type 'array'>*

Description: *Generate a cosenosoid image of size s with amplitude 1, period T, phase phi and wave direction of theta. The output image is a double array.*

Examples:

```
>>> import Numeric
>>> f = iacos([128,256], 100, Numeric.pi/4, 0)
>>> iashow(ianormalize(f, [0,255]))
```

(ianormalize(f, [0,255]))

Equation:

$$\begin{aligned} f(x,y) &= sin(2\pi(f_x x + f_y y) + \phi) \\ f_x &= \frac{cos(\theta)}{T} \\ f_y &= \frac{sin(\theta)}{T} \end{aligned}$$

C.1.5 iagaussian

Purpose: *Generate a 2D Gaussian image.*

Synopsis: `g = iagaussian(s, mu, sigma)`

Input:

s	*<type 'array'> [rows columns]*
mu	*<type 'array'> Mean vector. 2D point (x;y). Point of maximum value.*
sigma	*<type 'array'> covariance matrix (square). [Sx^2 Sxy; Syx Sy^2]*

Output:

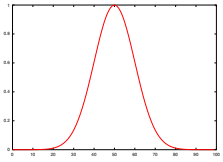
g	*<type 'array'>*

Description: *A 2D Gaussian image is an image with a Gaussian distribution. It can be used to generate test patterns or Gaussian filters both for spatial and frequency domain. The integral of the gaussian function is 1.0.*

Examples:

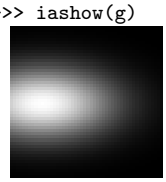
```
>>> import Numeric
>>> f = iagaussian([8,4], [3,1], [[1,0],[0,1]])
>>> print Numeric.array2string(f, precision=4, suppress_small=1)
[[ 0.0011  0.0018  0.0011  0.0002]
 [ 0.0131  0.0215  0.0131  0.0029]
 [ 0.0585  0.0965  0.0585  0.0131]
```

```
 [ 0.0965  0.1592  0.0965  0.0215]
 [ 0.0585  0.0965  0.0585  0.0131]
 [ 0.0131  0.0215  0.0131  0.0029]
 [ 0.0011  0.0018  0.0011  0.0002]
 [ 0.      0.0001  0.      0.    ]]
>>> g = ianormalize(f, [0,255]).astype(Numeric.UnsignedInt8)
>>> print g
[[  1   2   1   0]
 [ 20  34  20   4]
 [ 93 154  93  20]
 [154 255 154  34]
 [ 93 154  93  20]
 [ 20  34  20   4]
 [  1   2   1   0]
 [  0   0   0   0]]
>>> f = iagaussian(100, 50, 10*10)
>>> g = ianormalize(f, [0,1])
>>> g,d = iaplot(g)
```



(g)

```
>>> f = iagaussian([50,50], [25,10], [[10*10,0],[0,20*20]])
>>> g = ianormalize(f, [0,255]).astype(Numeric.UnsignedInt8)
>>> iashow(g)
```



(g)

Equation:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} exp\left[-\frac{1}{2}(\mathbf{x}-\mu)^t \Sigma^{-1}(\mathbf{x}-\mu)\right]$$

See also: *iacircle* (C.1.2)

C.1.6 ialog

Purpose: *Laplacian of Gaussian image.*

Synopsis: `g = ialog(s, mu, sigma)`

Input:

s	*<type 'array'> [rows cols], output image dimensions.*
mu	*<type 'array'> [row0 col0], center of the function.*
sigma	*<type 'float'> spread factor.*

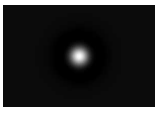
Output:

g	*<type 'array'>*

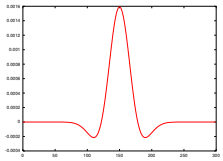
Description: *Creates a Laplacian of Gaussian image with dimensions given by s, origin given by c and spreading factor given by sigma. This function is used in the Marr-Hildreth filter.*

Examples:

```
>>> F = ialog([5,7], [3,4], 1)
>>> print Numeric.array2string(F, precision=4, suppress_small=1)
[[-0.     -0.0006 -0.0053 -0.0172 -0.0248 -0.0172 -0.0053]
 [-0.0003 -0.0053 -0.035  -0.0784 -0.0862 -0.0784 -0.035 ]
 [-0.001  -0.0172 -0.0784 -0.      0.1931 -0.     -0.0784]
 [-0.0015 -0.0248 -0.0862  0.1931  0.6366  0.1931 -0.0862]
 [-0.001  -0.0172 -0.0784 -0.      0.1931 -0.     -0.0784]]
>>> h = ialog([200,300], [100,150], 20)
>>> iashow(h)
>>> g,d = iaplot(h[100,:])
```



(h)



(h[100,:])

C.1.7 iaramp

Purpose: *Create an image with vertical bands of increasing gray values.*

Synopsis: `g = iaramp(s, n, range)`

Input:

s	*<type 'array'> [H W], height and width output image dimensions.*
n	*<type 'float'> number of vertical bands.*
range	*<type 'array'> [kmin, kmax], minimum and maximum gray scale values.*

Output:

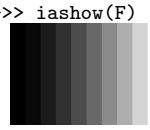
g	*<type 'array'>*

Description: *Creates a gray scale image with dimensions given by s, with n increasing gray scale bands from left to right with values varying from the specified range.*

Examples:

```
>>> F = iaramp([5,7], 3, [4,10])
>>> print F
[[ 4  4  4  7  7 10 10]
 [ 4  4  4  7  7 10 10]
 [ 4  4  4  7  7 10 10]
 [ 4  4  4  7  7 10 10]
 [ 4  4  4  7  7 10 10]]
```

```
>>> F = iaramp([200,300], 10, [0,255])
>>> iashow(F)
```



(F)

Equation:

$$g(x,y) = \lfloor \lfloor \frac{n}{W} x \rfloor \frac{k_{max} - k_{min}}{n-1} \rfloor + k_{min}$$

See also: *iacircle* (C.1.2)

C.1.8 iarectangle

Purpose: *Create a binary rectangle image.*

Synopsis: `g = iarectangle(s, r, c)`

Input:

`s`	*<type 'array'> [rows cols], output image dimensions.*
`r`	*<type 'float'> [rrows ccols], rectangle image dimensions.*
`c`	*<type 'array'> [row0 col0], center of the rectangle.*

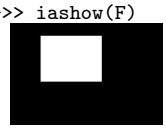
Output:

`g`	*<type 'array'>*

Description: *Creates a binary image with dimensions given by s, rectangle dimensions given by r and center given by c. The pixels inside the rectangle are one and outside zero.*

Examples:

```
>>> F = iarectangle([7,9], [3,2], [3,4])
>>> print F
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 1 1 0 0 0 0]
 [0 0 0 1 1 0 0 0 0]
 [0 0 0 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
>>> F = iarectangle([200,300], [90,120], [70,120])
>>> iashow(F)
```



(F)

Equation:

$$g(x,y) = \begin{cases} 1, & x_{min} \leq x < x_{max} \quad and \quad y_{min} \leq y < y_{max} \\ 0, & otherwise \end{cases}$$

See also: *iacircle* (C.1.2)

C.2 Image Information and Manipulation

C.2.1 iacrop

Purpose: *Crop an image to find the minimum rectangle.*

Synopsis: `g = iacrop(f, side='all', color='black')`

Input:

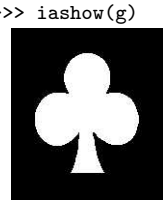
f	*<type 'array'> input image.*
side	*<type 'str'> side of the edge which will be removed. Possible values: 'all', 'left', 'right', 'top', 'bottom'.*
color	*<type 'str'> color of the edge. Possible values: 'black', 'white'.*

Output:

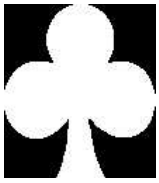
g	*<type 'array'>*

Examples:

```
>>> f = iaread('club.pgm')
>>> iashow(f)
>>> g = iacrop(f)
>>> iashow(g)
```



(f)



(g)

See also: *iapad* (C.2.6)

C.2.2 iaind2sub

Purpose: *Convert linear index to double subscripts.*

Synopsis: `x, y = iaind2sub(dim, i)`

Input:

dim	*<type 'array'> Dimension.*
i	*<type 'array'> Index.*

Output:

x	*<type 'array'>*
y	*<type 'array'>*

Examples:

```
>>> f = Numeric.array([[0,6,0,2],[4,0,1,8],[0,0,3,0]])
>>> print f
[[0 6 0 2]
 [4 0 1 8]
 [0 0 3 0]]
>>> i = Numeric.nonzero(Numeric.ravel(f))
>>> (x,y) = iaind2sub(f.shape, i)
>>> print x
```

```
[0 0 1 1 1 2]
>>> print y
[1 3 0 2 3 2]
>>> print f[x[0],y[0]]
6
>>> print f[x[4],y[4]]
8
```

C.2.3 iameshgrid

Purpose: *Create two 2-D matrices of indexes.*

Synopsis: `x, y = iameshgrid(vx, vy)`

Input:

`vx` *<type 'array'> Vector of indices of x coordinate.*
`vy` *<type 'array'> Vector of indices of y coordinate.*

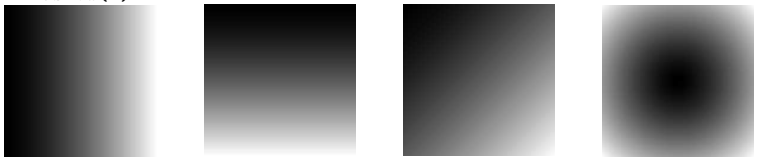
Output:

`x` *<type 'array'> 2-D matrix of indexes of x coordinate.*
`y` *<type 'array'> 2-D matrix of indexes of y coordinate.*

Description: *This function generates 2-D matrices of indices of the domain specified by arange1 and arange2. This is very useful to generate 2-D functions. Note that unlike other functions, the order of the parameters uses the cartesian coordenate convention. arange1 is for x (horizontal), and arange2 is for y (vertical).*

Examples:

```
>>> (x, y) = iameshgrid(Numeric.arange(1,3,0.5), Numeric.arange(2,4,0.6))
>>> print x
[[ 1.   1.5  2.   2.5]
 [ 1.   1.5  2.   2.5]
 [ 1.   1.5  2.   2.5]
 [ 1.   1.5  2.   2.5]]
>>> print y
[[ 2.   2.   2.   2. ]
 [ 2.6  2.6  2.6  2.6]
 [ 3.2  3.2  3.2  3.2]
 [ 3.8  3.8  3.8  3.8]]
>>> print x + y
[[ 3.   3.5  4.   4.5]
 [ 3.6  4.1  4.6  5.1]
 [ 4.2  4.7  5.2  5.7]
 [ 4.8  5.3  5.8  6.3]]
>>> (x, y) = iameshgrid(range(256), range(256))
>>> iashow(x)
>>> iashow(y)
>>> iashow(x + y)
>>> z = Numeric.sqrt((x-127)**2 + (y-127)**2)
>>> iashow(z)
```



(x) (y) (x + y) (z)

Equation:

$$\begin{aligned} v_x(j), j = 0, 1, \ldots n \\ v_y(i), i = 0, 1, \ldots m \\ x(i,j) = v_x(i) \\ y(i,j) = v_y(j) \end{aligned}$$

C.2.4 ianeg

Purpose: *Negate an image.*

Synopsis: `g = ianeg(f)`

Input:

`f` *<type 'array'> Set initial.*

Output:

`g` *<type 'array'>*

Description: *Returns an image that is the negation (i.e., inverse or involution) of the input image.*

Examples:

```
>>> c1 = Numeric.array([0, 53, 150, 255], Numeric.UnsignedInt8)
>>> print c1
[  0  53 150 255]
>>> n1 = ianeg(c1)
>>> print n1
[255 202 105   0]
>>> c2 = Numeric.array([-129, -128, 0, 127, 128], Numeric.Int8)
>>> print c2
[ 127 -128    0  127 -128]
>>> n2 = ianeg(c2)
>>> print n2
[-127.  128.   -0. -127.  128.]
```

Equation:

$$\nu(f)(x) = k - f(x)$$

C.2.5 ianormalize

Purpose: *Normalize the pixels values between the specified range.*

Synopsis: `g = ianormalize(f, range)`

Input:

`f` *<type 'array'> input image.*

`range` *<type 'array'> vector: minimum and maximum values in the output image, respectively.*

Output:

`g` *<type 'array'> normalized image.*

Description: *Normalize the input image f. The minimum value of f is assigned to the minimum desired value and the maximum value of f, to the maximum desired value. The minimum and maximum desired values are given by the parameter range.*

Examples:

```
>>> import Numeric
>>> f = Numeric.array([100., 500., 1000.])
>>> g1 = ianormalize(f, [0,255])
>>> print g1
[   0.          113.33333333  255.        ]
>>> g2 = ianormalize(f, [-1,1])
>>> print g2
[-1.         -0.11111111  1.        ]
>>> g3 = ianormalize(f, [0,1])
>>> print g3
[ 0.          0.44444444  1.        ]
>>> #
>>> f = Numeric.array([-100., 0., 100.])
>>> g4 = ianormalize(f, [0,255])
>>> print g4
[   0.   127.5  255. ]
>>> g5 = ianormalize(f, [-1,1])
>>> print g5
[-1.  0.  1.]
```

Equation:

$$\begin{aligned} g &= f|_{gmin}^{gmax} \\ g(p) &= \frac{g_{max} - g_{min}}{f_{max} - f_{min}}(f(p) - f_{min}) + g_{min} \end{aligned}$$

C.2.6 iapad

Purpose: *Extend the image inserting a frame around it.*

Synopsis: `g = iapad(f, thick=[1,1], value=0)`

Input:

`f`	*<type 'array'> input image.*
`thick`	*<type 'array'> [rows cols] to be padded.*
`value`	*<type 'float'> value used in the frame around the image.*

Output:

`g`	*<type 'array'>*

Examples:

```
>>> f = Numeric.array([[0,1,2],[3,4,5]], Numeric.UnsignedInt8)
>>> print f
[[0 1 2]
 [3 4 5]]
>>> g1 = iapad(f)
>>> print g1
[[0 0 0 0 0]
 [0 0 1 2 0]
 [0 3 4 5 0]
 [0 0 0 0 0]]
```

```
>>> g2 = iapad(f, (1,3), 5)
>>> print g2
[[5 5 5 5 5 5 5 5 5]
 [5 5 5 0 1 2 5 5 5]
 [5 5 5 3 4 5 5 5 5]
 [5 5 5 5 5 5 5 5 5]]
```

See also: *iacrop* (C.2.1)

C.2.7 iaroi

Purpose: *Cut a rectangle out of an image.*

Synopsis: `g = iaroi(f, p1, p2)`

Input:

f	*<type 'array'> input image.*
p1	*<type 'array'> indices of the coordinates at top-left.*
p2	*<type 'array'> indices of the coordinates at bottom-right.*

Output:

g	*<type 'array'>*

Examples:

```
>>> f = iaread('lenina.pgm')
>>> iashow(f)
>>> froi = iaroi(f, [90,70], [200,180])
>>> iashow(froi)
```

(f)

(froi)

C.2.8 iasub2ind

Purpose: *Convert linear double subscripts to linear index.*

Synopsis: `i = iasub2ind(dim, x, y)`

Input:

dim	*<type 'array'> Dimension.*
x	*<type 'array'> x index.*
y	*<type 'array'> y index.*

Output:

i	*<type 'array'>*

Examples:

```
>>> f = Numeric.array([[0,6,0,2],[4,0,1,8],[0,0,3,0]])
>>> print f
[[0 6 0 2]
 [4 0 1 8]
```

```
 [0 0 3 0]]
>>> x=[0,0,1,2,2,2]
>>> y=[0,2,1,0,1,3]
>>> print x
[0, 0, 1, 2, 2, 2]
>>> print y
[0, 2, 1, 0, 1, 3]
>>> i = iasub2ind(f.shape, x, y)
>>> print i
[ 0  2  5  8  9 11]
>>> Numeric.put(f, i, 10)
>>> print f
[[10  6 10  2]
 [ 4 10  1  8]
 [10 10  3 10]]
```

C.2.9 iatile

Purpose: *Replicate the image until reach a new size.*

Synopsis: `g = iatile(f, new_size)`

Input:

`f`	*<type 'array'> input image.*
`new_size`	*<type 'array'> [rows cols], output image dimensions.*

Output:

`g`	*<type 'array'>*

Examples:

```
>>> f=[[1,2],[3,4]]
>>> print f
[[1, 2], [3, 4]]
>>> g = iatile(f, (3,6))
>>> print g
[[1 2 1 2 1 2]
 [3 4 3 4 3 4]
 [1 2 1 2 1 2]]
```

C.3 Image file I/O

C.3.1 iaread

Purpose: *Read an image file (PBM, PGM and PPM).*

Synopsis: `img = iaread(filename)`

Input:

`filename`	*<type 'str'>*

Output:

`img`	*<type 'array'>*

Examples:

```
>>> f = iaread('boat.ppm')
>>> iashow(f)
```

(f)

See also: *iawrite* (C.3.2)

C.3.2 iawrite

Purpose: *Write an image file (PBM, PGM and PPM).*

Synopsis: `iawrite(arrayname, filename, mode='bin')`

Input:

arrayname	*<type 'array'>*
filename	*<type 'str'>*
mode	*<type 'str'>*

Examples:

```
>>> f = Numeric.resize(range(256), (256, 256))
>>> f_color = Numeric.zeros((256, 256, 3))
>>> f_color[:,:,0] = f; f_color[:,:,1] = 127; f_color[:,:,2] = 255-f
>>> import tempfile
>>> file_name = tempfile.mktemp() # Name for a temporary file
>>> print file_name
/tmp/@1647.2
>>> print 'Saving f in '+file_name+'.pgm ...'
Saving f in /tmp/@1647.2.pgm ...
>>> iawrite(f, file_name+'.pgm')
>>> print 'Saving f_color in '+file_name+'.ppm ...'
Saving f_color in /tmp/@1647.2.ppm ...
>>> iawrite(f_color, file_name+'.ppm')
```

C.4 Contrast Manipulation

C.4.1 iaapplylut

Purpose: *Intensity image transform.*

Synopsis: `g = iaapplylut(fi, it)`

Input:

fi	*<type 'array'> input image, gray scale or index image.*
it	*<type 'array'> Intensity transform. Table of one or three columns.*

Output:

g	*<type 'array'>*

Description: *Apply an intensity image transform to the input image. The input image can be seen as an gray scale image or an index image. The intensity transform is represented by a table where the input (gray scale) color address the table line and its column contents indicates the output (gray scale) image color. The table can have one or three columns. If it has three columns, the output image is a three color band image. This intensity image transformation is very powerful and can be use in many applications involving gray scale and color images. If the input image has an index (gray scale color) that is greater than the size of the intensity table, an error is reported.*

Examples:

```
>>> f = [[0,1,2], [3,4,5]]
>>> it = Numeric.array(range(6)) # identity transform
>>> print it
[0 1 2 3 4 5]
>>> g = iaapplylut(f, it)
>>> print g
[[0 1 2]
 [3 4 5]]
>>> itn = 5 - it  # negation
>>> g = iaapplylut(f, itn)
>>> print g
[[5 4 3]
 [2 1 0]]
>>> f = iaread('cameraman.pgm')
>>> it = 255 - Numeric.arange(256)
>>> g = iaapplylut(f, it)
>>> iashow(f)
>>> iashow(g)
```

(f)

(g)

```
>>> f = [[0,1,1], [0,0,1]]
>>> ct = [[255,0,0], [0,255,0]]
>>> g = iaapplylut(f,ct)
>>> print g
[[[255   0   0]
  [  0 255   0]
  [  0 255   0]]
 [[255   0   0]
  [255   0   0]
  [  0 255   0]]]
>>> f = iaread('cameraman.pgm')
>>> aux = Numeric.resize(range(256), (256,1))
>>> ct = Numeric.concatenate((aux, Numeric.zeros((256,2))), 1)
>>> g = iaapplylut(f, ct)
>>> iashow(f)
>>> iashow(g)
```

(f)

(g)

Equation:

$$g(x,y) = IT(f(x,y))$$

$$g_r(x,y) = IT(f(x,y),0)$$
$$g_g(x,y) = IT(f(x,y),1)$$
$$g_b(x,y) = IT(f(x,y),2)$$

C.4.2 iacolormap

Purpose: *Create a colormap table.*

Synopsis: `ct = iacolormap(type='gray')`

Input:

`type` *<type 'str'> Type of the colormap. Options: 'gray', 'hsv', 'hot', 'cool', 'bone','copper', 'pink'.*

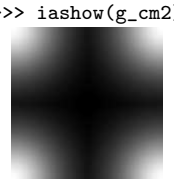
Output:

`ct` *<type 'array'> Colormap table.*

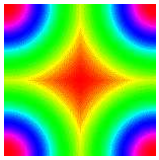
Description: *Create a colormap table.*

Examples:

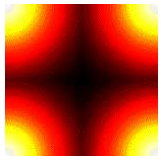
```
>>> f = ianormalize(iabwlp([150,150], 4, 1), [0,255]).astype('b')
>>> cm1 = iacolormap('hsv')
>>> g_cm1 = iaapplylut(f, cm1)
>>> cm2 = iacolormap('hot')
>>> g_cm2 = iaapplylut(f, cm2)
>>> iashow(f)
>>> iashow(g_cm1)
>>> iashow(g_cm2)
```



(f)



(g_cm1)



(g_cm2)

C.5 Color Processing

C.5.1 iahsv2rgb

Purpose: *Convert HSV to RGB color model.*

Synopsis: `g = iahsv2rgb(f)`

Input:

`f` *<type 'array'> HSV color model.*

Output:

g *<type 'array'> True color RGB image.*

Description: *Converts hue-saturation-value to red-green-blue colors.*

Examples:

```
>>> import Numeric
>>> r = [[4,5,6],[4,2,4]]
>>> g = [[0,1,2],[0,4,0]]
>>> b = [[1,0,2],[1,2,2]]
>>> f = Numeric.zeros((2,3,3))
>>> f[:,:,0], f[:,:,1], f[:,:,2] = r, g, b
>>> print f[:,:,0]
[[4 5 6]
 [4 2 4]]
>>> print f[:,:,1]
[[0 1 2]
 [0 4 0]]
>>> print f[:,:,2]
[[1 0 2]
 [1 2 2]]
>>> g = iargb2hsv(f)
>>> f_ = iahsv2rgb(g)
>>> print f_[:,:,0]
[[ 4.  5.  6.]
 [ 4.  2.  4.]]
>>> print f_[:,:,1]
[[ 0.  1.  2.]
 [ 0.  4.  0.]]
>>> print f_[:,:,2]
[[ 1.  0.  2.]
 [ 1.  2.  2.]]
```

C.5.2 iargb2hsv

Purpose: *Convert RGB to HSV color model.*

Synopsis: `g = iargb2hsv(f)`

Input:

f *<type 'array'> True color RGB image.*

Output:

g *<type 'array'> HSV color model.*

Description: *Converts red-green-blue colors to hue-saturation-value.*

Examples:

```
>>> import Numeric
>>> r = [[4,5,6],[4,2,4]]
>>> g = [[0,1,2],[0,4,0]]
>>> b = [[1,0,2],[1,2,2]]
>>> f = Numeric.zeros((2,3,3))
>>> f[:,:,0], f[:,:,1], f[:,:,2] = r, g, b
>>> print f[:,:,0]
[[4 5 6]
```

```
 [4 2 4]]
>>> print f[:,:,1]
[[0 1 2]
 [0 4 0]]
>>> print f[:,:,2]
[[1 0 2]
 [1 2 2]]
>>> g = iargb2hsv(f)
>>> print g[:,:,0]
[[ 0.95833333  0.03333333  0.        ]
 [ 0.95833333  0.33333333  0.91666667]]
>>> print g[:,:,1]
[[ 1.          1.          0.66666667]
 [ 1.          0.5         1.        ]]
>>> print g[:,:,2]
[[ 0.01568627  0.01960784  0.02352941]
 [ 0.01568627  0.01568627  0.01568627]]
```

C.5.3 iargb2ycbcr

Purpose: *Convert RGB to YCbCr color model.*

Synopsis: `g = iargb2ycbcr(f)`

Input:

`f` *<type 'array'> True color RGB image.*

Output:

`g` *<type 'array'> YCbCr image.*

Description: *Convert RGB values to YCbCr color space.*

Examples:

```
>>> import Numeric
>>> r = [[4,5,6],[4,2,4]]
>>> g = [[0,1,2],[0,4,0]]
>>> b = [[1,0,2],[1,2,2]]
>>> f = Numeric.zeros((2,3,3))
>>> f[:,:,0], f[:,:,1], f[:,:,2] = r, g, b
>>> print f[:,:,0]
[[4 5 6]
 [4 2 4]]
>>> print f[:,:,1]
[[0 1 2]
 [0 4 0]]
>>> print f[:,:,2]
[[1 0 2]
 [1 2 2]]
>>> g = iargb2ycbcr(f)
>>> print g[:,:,0]
[[ 17.126  17.789  18.746]
 [ 17.126  18.726  17.224]]
>>> print g[:,:,1]
[[ 127.847  126.969  127.408]
 [ 127.847  127.418  128.286]]
>>> print g[:,:,2]
[[ 129.685  129.827  129.756]
 [ 129.685  127.264  129.614]]
```

C.5.4 iatcrgb2ind

Purpose: *True color RGB to index image and colormap.*

Synopsis: `fi,cm = iatcrgb2ind(f)`

Input:

`f` *<type 'array'> True color RGB image.*

Output:

`fi,cm` *<type 'array'> Index image and its colormap.*

Description: *Converts a true color RGB image to the format of index image and a colormap.*

Examples:

```
>>> import Numeric
>>> r = [[4,5,6],[4,2,4]]
>>> g = [[0,1,2],[0,4,0]]
>>> b = [[1,0,2],[1,2,2]]
>>> f = Numeric.zeros((2,3,3))
>>> f[:,:,0], f[:,:,1], f[:,:,2] = r, g, b
>>> print f
[[[4 0 1]
  [5 1 0]
  [6 2 2]]
 [[4 0 1]
  [2 4 2]
  [4 0 2]]]
>>> (fi, tm) = iatcrgb2ind(f)
>>> print fi
[[1 0 3]
 [1 4 2]]
>>> print tm
[[5 1 0]
 [4 0 1]
 [4 0 2]
 [6 2 2]
 [2 4 2]]
```

C.5.5 iaycbcr2rgb

Purpose: *Convert RGB to YCbCr color model.*

Synopsis: `g = iaycbcr2rgb(f)`

Input:

`f` *<type 'array'> YCbCr image.*

Output:

`g` *<type 'array'> True color RGB image.*

Description: *Convert RGB values to YCbCr color space.*

Examples:

```
>>> import Numeric
>>> r = [[4,5,6],[4,2,4]]
>>> g = [[0,1,2],[0,4,0]]
>>> b = [[1,0,2],[1,2,2]]
>>> f = Numeric.zeros((2,3,3))
>>> f[:,:,0], f[:,:,1], f[:,:,2] = r, g, b
>>> print f[:,:,0]
[[4 5 6]
 [4 2 4]]
>>> print f[:,:,1]
[[0 1 2]
 [0 4 0]]
>>> print f[:,:,2]
[[1 0 2]
 [1 2 2]]
>>> g = iargb2ycbcr(f)
>>> f_ = iaycbcr2rgb(g)
>>> print f_[:,:,0]
[[ 3.999924  4.998288  5.99892 ]
 [ 3.999924  1.998408  4.00068 ]]
>>> print f_[:,:,1]
[[  7.35000000e-04   1.00119700e+00   2.00078000e+00]
 [  7.35000000e-04   3.99957600e+00   4.42000000e-04]]
>>> print f_[:,:,2]
[[ 1.002063  0.002869  2.00228 ]
 [ 1.002063  1.99917   2.001598]]
```

C.6 Geometric Manipulations

C.6.1 iaffine

Purpose: *Affine transform.*

Synopsis: `g = iaffine(f, T)`

Input:

`f` <*type 'array'*>
`T` <*type 'array'*> *Affine matrix for the geometric transformation.*

Output:

`g` <*type 'array'*>

Description: *Apply the affine transform to the coordinate pixels of image f. The resultant image g has the same domain of the input image. Any pixel outside this domain is not shown and any pixel that does not exist in the original image has the nearest pixel value. This method is based on the inverse mapping. An affine transform is a geometrical transformation that preserves the parallelism of lines but not their lengths and angles. The affine transform can be a composition of translation, rotation, scaling and shearing. To simplify this composition, these transformations are represented in homogeneous coordinates using 3x3 matrix T.*

Examples:

```
>>> f = Numeric.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
>>> print f
[[ 1  2  3  4  5]
```

```
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
>>> T = Numeric.array([[1,0,0],[0,1,0],[0,0,1]], 'd')
>>> print T
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
>>> g = iaffine(f,T)
>>> print g
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
>>> T[0,0] = 0.5
>>> print T
[[ 0.5  0.   0. ]
 [ 0.   1.   0. ]
 [ 0.   0.   1. ]]
>>> g = iaffine(f,T)
>>> print g
[[ 2  4  5  5  5]
 [ 7  9 10 10 10]
 [12 14 15 15 15]]
```

Equation:

$$
\begin{aligned}
T &= \begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} &= T \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
g(x,y) &= f(T^{-1}(x,y)); (x,y) \in D(f)
\end{aligned}
$$

See also: *iageorigid* (C.6.2)

C.6.2 iageorigid

Purpose: *2D Rigid body geometric transformation and scaling.*

Synopsis: `g = iageorigid(f, scale, theta, t)`

Input:

`f`	*<type 'array'>*
`scale`	*<type 'array'> [srow scol], scale in each dimension*
`theta`	*<type 'float'> Rotation*
`t`	*<type 'array'> [trow tcol], translation in each dimension*

Output:

`g`	*<type 'array'>*

Examples:

```
>>> import Numeric
>>> f = iaread('lenina.pgm')
>>> g = iageorigid(f, [0.5,0.5], Numeric.pi/4, [0,64])
```

```
>>> iashow(g)
```

(g)

Equation:

$$
\begin{aligned}
T_\theta &= \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
T_s &= \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
T_t &= \begin{bmatrix} 0 & 0 & t_x \\ 0 & 0 & t_y \\ 0 & 0 & 1 \end{bmatrix} \\
T &= T_t T_\theta T_s \\
g(x,y) &= f(T^{-1}(x,y)); (x,y) \in D(f)
\end{aligned}
$$

See also: *iaffine* (C.6.1)

C.6.3 iaptrans

Purpose: *Periodic translation.*

Synopsis: `g = iaptrans(f, t)`

Input:

`f` *<type 'array'>*
`t` *<type 'array'> [rows cols] to translate.*

Output:

`g` *<type 'array'>*

Description: *Translate the image periodically by t=[r0 c0]. This translation can be seen as a window view displacement on an infinite tile wall where each tile is a copy of the original image. The periodical translation is related to the periodic convolution and discrete Fourier transform. Be careful when implementing this function using the mod, some mod implementations in C does not follow the correct definition when the number is negative.*

Examples:

```
>>> import Numeric
>>> f = Numeric.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
>>> print f
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
>>> print iaptrans(f, [-1,2]).astype(Numeric.UnsignedInt8)
[[ 9 10  6  7  8]
```

```
 [14 15 11 12 13]
 [ 4  5  1  2  3]]
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> iashow(iaptrans(f, Numeric.array(f.shape)/2))
```

(f)

(iaptrans(f, Numeric.array(f.shape)/2))

Equation:

$$\begin{aligned} t &= (t_x t_y) \\ g = f_t &= f_{tx,ty} \\ g(x,y) &= f((x-t_x)\ mod\ W, (y-t_y)\ mod\ H), 0 \le x < W, 0 \le y < H \\ \text{where} & \\ a\ mod\ N &= (a+kN)\ mod\ N, k \in Z \end{aligned}$$

See also: *iapconv* (C.8.4)

C.6.4 iaresize

Purpose: *Resizes an image.*

Synopsis: `g = iaresize(f, new_shape)`

Input:

f	*<type 'array'>*
new_shape	*<type 'array'> [h w], new image dimensions*

Output:

g	*<type 'array'>*

Examples:

```
>>> import Numeric
>>> f = Numeric.array([[10,20,30],[40,50,60]])
>>> print f
[[10 20 30]
 [40 50 60]]
>>> g = iaresize(f, [5,7])
>>> print g
[[10 10 10 20 20 30 30]
 [10 10 10 20 20 30 30]
 [10 10 10 20 20 30 30]
 [40 40 40 50 50 60 60]
 [40 40 40 50 50 60 60]]
```

C.7 Image Transformation

C.7.1 iadct

Purpose: *Discrete Cossine Transform.*

Synopsis: `F = iadct(f)`

Input:

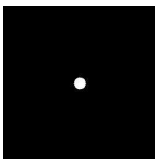
`f` *<type 'array'>*

Output:

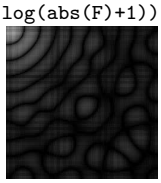
`F` *<type 'array'>*

Examples:

```
>>> import Numeric
>>> f = 255 * iacircle([256,256], 10, [129,129])
>>> iashow(f)
>>> F = iadct(f)
>>> iashow(Numeric.log(abs(F)+1))
```



(f)



(Numeric.log(abs(F)+1))

C.7.2 iadctmatrix

Purpose: *Kernel matrix for the DCT Transform.*

Synopsis: `A = iadctmatrix(N)`

Input:

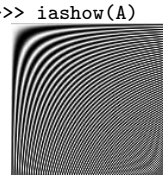
`N` *<type 'float'>*

Output:

`A` *<type 'array'>*

Examples:

```
>>> A = iadctmatrix(128)
>>> iashow(A)
```



(A)

```
>>> import Numeric
>>> A = iadctmatrix(4)
>>> print Numeric.array2string(A, precision=4, suppress_small=1)
[[ 0.5     0.5     0.5     0.5   ]
 [ 0.6533  0.2706 -0.2706 -0.6533]
```

```
 [ 0.5    -0.5    -0.5     0.5   ]
 [ 0.2706 -0.6533  0.6533 -0.2706]]
>>> B = Numeric.matrixmultiply(A,Numeric.transpose(A))
>>> print Numeric.array2string(B, precision=4, suppress_small=1)
[[ 1.  0. -0. -0.]
 [ 0.  1.  0.  0.]
 [-0.  0.  1.  0.]
 [-0.  0.  0.  1.]]
```

C.7.3 iadft

Purpose: *Discrete Fourier Transform.*

Synopsis: `F = iadft(f)`

Input:

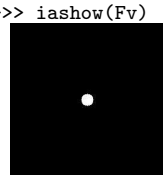
`f` *<type 'array'>*

Output:

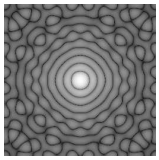
`F` *<type 'array'>*

Examples:

```
>>> f = 255 * iacircle([256,256], 10, [129,129])
>>> iashow(f)
>>> F = iadft(f)
>>> Fv = iadftview(F)
>>> iashow(Fv)
```



(f)



(Fv)

Equation:

$$\begin{aligned} F(u) &= \sum_{x=0}^{N-1} f(x)\exp(-j2\pi\frac{ux}{N}) \\ & 0 \le x < N, 0 \le u < N \\ F &= \sqrt{N}A_N f \end{aligned}$$

$$\begin{aligned} F(u,v) &= \sum_{x=0}^{N-1}\sum_{y=0}^{M-1} f(x,y)\exp(-j2\pi(\frac{ux}{N}+\frac{vy}{M})) \\ & (0,0) \le (x,y) < (N,M), (0,0) \le (u,v) < (N,M) \\ F &= \sqrt{NM}A_N f A_M \end{aligned}$$

See also: *iaidft* (C.7.11), *iadftmatrix* (C.7.4), *iadftview* (C.12.1), *iaisdftsym* (C.7.15)

C.7.4 iadftmatrix

Purpose: *Kernel matrix for the DFT Transform.*

Synopsis: `A = iadftmatrix(N)`

Input:

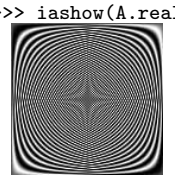
`N` *<type 'float'>*

Output:

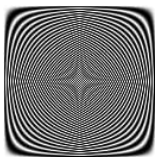
`A` *<type 'array'>*

Examples:

```
>>> import Numeric
>>> A = iadftmatrix(128)
>>> iashow(A.imag)
>>> iashow(A.real)
```



(A.imag)



(A.real)

```
>>> import Numeric
>>> import LinearAlgebra
>>> A = iadftmatrix(4)
>>> print Numeric.array2string(A, precision=4, suppress_small=1)
[[ 0.5+0.j   0.5+0.j   0.5+0.j   0.5+0.j ]
 [ 0.5+0.j   0. -0.5j -0.5-0.j  -0. +0.5j]
 [ 0.5+0.j  -0.5-0.j   0.5+0.j  -0.5-0.j ]
 [ 0.5+0.j  -0. +0.5j -0.5-0.j   0. -0.5j]]
>>> print Numeric.array2string(A-Numeric.transpose(A), precision=4, suppress_small=1)↩

[[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]]
>>> print abs(LinearAlgebra.inverse(A)-Numeric.conjugate(A)) < 10E-15
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

Equation:

$$
\begin{aligned}
W_N &= \exp\frac{-j2\pi}{N} \\
A_N &= \frac{1}{\sqrt{N}}(W_N)^{\mathbf{u}\mathbf{x}^T} \\
\mathbf{u} &= \mathbf{x} = [0, 1, 2, \ldots, N-1]^T
\end{aligned}
$$

$$
\begin{aligned}
A_N &= A_N^T \text{ symmetric} \\
(A_N)^{-1} &= (A_N)^* \text{ column orthogonality, unitary matrix}
\end{aligned}
$$

See also: *iadft* (C.7.3), *iaidft* (C.7.11)

C.7.5 iafftshift

Purpose: *Shifts zero-frequency component to center of spectrum.*

Synopsis: `HS = iafftshift(H)`

Input:

`H` *<type 'array'> FFT image.*

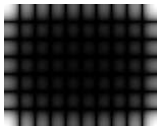
Output:

`HS` *<type 'array'> Shift of the FFT image.*

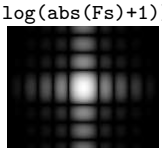
Description: *The origen (0,0) of the DFT is normally at top-left corner of the image. For visualization purposes, it is common to periodicaly translate the origen to the image center. This is particularly interesting because of the complex conjugate simmetry of the DFT of a real function. Note that as the image can have even or odd sizes, to translate back the DFT from the center to the corner, there is another correspondent function.*

Examples:

```
>>> import Numeric
>>> import FFT
>>> f = iarectangle([120,150],[7,10],[60,75])
>>> F = FFT.fft2d(f)
>>> Fs = iafftshift(F)
>>> iashow(Numeric.log(abs(F)+1))
>>> iashow(Numeric.log(abs(Fs)+1))
```



(Numeric.log(abs(F)+1))



(Numeric.log(abs(Fs)+1))

Equation:

$$\begin{aligned} HS &= H_{xo,yo} \\ xo &= \lfloor W/2 \rfloor \\ yo &= \lfloor H/2 \rfloor \end{aligned}$$

See also: *iaiafftshift* (C.7.12), *iaptrans* (C.6.3), *iadftview* (C.12.1)

C.7.6 iahaarmatrix

Purpose: *Kernel matrix for the Haar Transform.*

Synopsis: `A = iahaarmatrix(N)`

Input:

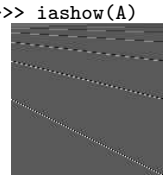
`N` *<type 'float'>*

Output:

`A` *<type 'array'>*

Examples:

```
>>> A = iahaarmatrix(128)
>>> iashow(A)
```



(A)

```
>>> import Numeric
>>> A = iahaarmatrix(4)
>>> print A
[[ 0.5         0.5         0.5         0.5       ]
 [ 0.5         0.5        -0.5        -0.5       ]
 [ 0.70710678 -0.70710678  0.          0.        ]
 [ 0.          0.          0.70710678 -0.70710678]]
>>> print Numeric.matrixmultiply(A, Numeric.transpose(A))
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

C.7.7 iahadamard

Purpose: *Hadamard Transform.*

Synopsis: `F = iahadamard(f)`

Input:

`f` *<type 'array'>*

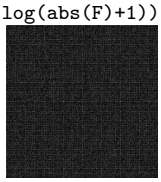
Output:

`F` *<type 'array'>*

Examples:

```
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> F = iahadamard(f)
>>> iashow(Numeric.log(abs(F)+1))
```

(f)



(Numeric.log(abs(F)+1))

C.7.8 iahadamardmatrix

Purpose: *Kernel matrix for the Hadamard Transform.*

Synopsis: `A = iahadamardmatrix(N)`

Input:

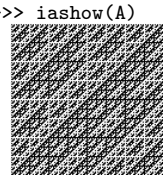
N *<type 'float'>*

Output:

A *<type 'array'>*

Examples:

```
>>> A = iahadamardmatrix(128)
>>> iashow(A)
```



(A)

```
>>> import Numeric
>>> A = iahadamardmatrix(4)
>>> print A
[[ 0.5  0.5  0.5  0.5]
 [ 0.5 -0.5  0.5 -0.5]
 [ 0.5  0.5 -0.5 -0.5]
 [ 0.5 -0.5 -0.5  0.5]]
>>> print Numeric.matrixmultiply(A, Numeric.transpose(A))
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

C.7.9 iahwt

Purpose: *Haar Wavelet Transform.*

Synopsis: `F = iahwt(f)`

Input:

f *<type 'array'>*

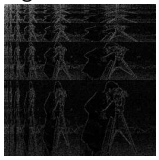
Output:

F *<type 'array'>*

Examples:

```
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> F = iahwt(f)
>>> iashow(Numeric.log(abs(F)+1))
```

(f)



(Numeric.log(abs(F)+1))

C.7.10 iaidct

Purpose: *Inverse Discrete Cossine Transform.*

Synopsis: `F = iaidct(f)`

Input:

`f` *<type 'array'>*

Output:

`F` *<type 'array'>*

Examples:

```
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> F = iadct(f)
>>> g = iaidct(F)
>>> print Numeric.sum(Numeric.sum(abs(f-g)))
1.3993287657e-07
```

(f)

C.7.11 iaidft

Purpose: *Inverse Discrete Fourier Transform.*

Synopsis: `f = iaidft(F)`

Input:

`F` *<type 'array'>*

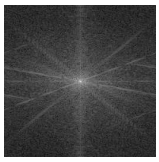
Output:

`f` *<type 'array'>*

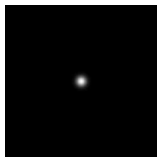
Examples:

```
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> F = iadft(f)
>>> H = iagaussian(F.shape, Numeric.array(F.shape)/2., [[50,0],[0,50]])
>>> H = ianormalize(H,[0,1])
>>> FH = F * iaifftshift(H)
>>> print iaisdftsym(FH)
1
>>> g=iaidft(FH)
>>> iashow(f)
>>> iashow(iadftview(F))
>>> iashow(ianormalize(H,[0,255]))
>>> iashow(iadftview(FH))
>>> iashow(abs(g))
```

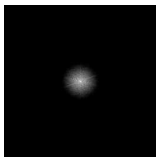
(f)



(iadftview(F))



(ianormalize(H,[0,255]))



(iadftview(FH))

(abs(g))

Equation:

$$\begin{aligned} f(x) &= \frac{1}{N}\sum_{u=0}^{N-1} F(u)\exp(j2\pi\frac{ux}{N}) \\ & \quad 0 \le x < N, 0 \le u < N \\ f &= \frac{1}{\sqrt{N}}(A_N)^* F \end{aligned}$$

$$\begin{aligned} f(x,y) &= \frac{1}{NM}\sum_{u=0}^{N-1}\sum_{v=0}^{M-1} F(u,v)\exp(j2\pi(\frac{ux}{N}+\frac{vy}{M})) \\ & \quad (0,0) \le (x,y) < (N,M), (0,0) \le (u,v) < (N,M) \\ f &= \frac{1}{\sqrt{NM}}(A_N)^* F (A_M)^* \end{aligned}$$

See also: *iadft* (C.7.3), *iaisdftsym* (C.7.15)

C.7.12 iaifftshift

Purpose: *Undoes the effects of iafftshift.*

Synopsis: `HS = iaifftshift(H)`

Input:

`H` *<type 'array'> DFT image with (0,0) in the center.*

Output:

`HS` *<type 'array'> DFT image with (0,0) in the top-left corner.*

Equation:

$$\begin{aligned} HS &= H_{xo,yo} \\ xo &= \lceil -W/2 \rceil \\ yo &= \lceil -H/2 \rceil \end{aligned}$$

See also: *iafftshift* (C.7.5), *iaptrans* (C.6.3)

C.7.13 iaihadamard

Purpose: *Inverse Hadamard Transform.*

Synopsis: `F = iaihadamard(f)`

Input:

`f` *<type 'array'>*

Output:

`F` *<type 'array'>*

Examples:

```
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> F = iahadamard(f)
>>> g = iaihadamard(F)
>>> print Numeric.sum(Numeric.sum(abs(f.astype(Numeric.Float)-g.astype(Numeric.Float)↩
)))
0.0
```

(f)

C.7.14 iaihwt

Purpose: *Inverse Haar Wavelet Transform.*

Synopsis: `F = iaihwt(f)`

Input:

`f` *<type 'array'>*

Output:

`F` *<type 'array'>*

Examples:

```
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> F = iahwt(f)
>>> g = iaihwt(F)
>>> print Numeric.sum(Numeric.sum(abs(f.astype(Numeric.Float)-g.astype(Numeric.Float)↩
)))
4.29271901825e-10
```

(f)

C.7.15 iaisdftsym

Purpose: *Check for conjugate symmetry*

Synopsis: `b = iaisdftsym(F)`

Input:

`F` *<type 'array'> Complex image.*

Output:

`b` *<type 'int'>*

Description: *Verify if a complex array show the conjugate simmetry. Due to numerical precision, this comparison is not exact but with within a small tolerance (10E-4). This comparison is useful to verify if the result of a filtering in the frequency domain is correct. Before taking the inverse DCT, the Fourier transform must be conjugate symmetric so that its inverse is a real function (an image).*

Examples:

```
>>> import FFT, MLab
>>> print iaisdftsym(FFT.fft2d(MLab.rand(100,100)))
1
>>> print iaisdftsym(FFT.fft2d(MLab.rand(101,100)))
1
>>> print iaisdftsym(FFT.fft2d(MLab.rand(101,101)))
1
>>> print iaisdftsym(iabwlp([10,10], 8, 5))
1
>>> print iaisdftsym(iabwlp([11,11], 8, 5))
0
```

Equation:

$$\begin{aligned} F(u,v) &= F^{\star}(-u \bmod N, -v \bmod M) \\ & \quad (0,0) \leq (u,v) < (N,M) \end{aligned}$$

See also: *iadft* (C.7.3), *iaidft* (C.7.11)

C.8 Image Filtering

C.8.1 iacontour

Purpose: *Contours of binary images.*

Synopsis: `g = iacontour(f)`

Input:

`f` *<type 'array'> input image*

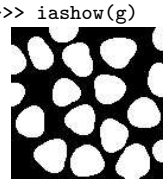
Output:

`g` *<type 'array'>*

Description: *Contours of binary images.*

Examples:

```
>>> f = iaread('blobs.pbm')
>>> g = iacontour(f)
>>> iashow(f)
>>> iashow(g)
```



(f)

(g)

C.8.2 iaconv

Purpose: *2D convolution.*

Synopsis: `g = iaconv(f, h)`

Input:

`f` *<type 'array'> input image.*

`h` *<type 'array'> PSF (point spread function), or kernel. The origin is at the array center.*

Output:

`g` *<type 'array'>*

Description: *Perform a 2D discrete convolution. The kernel origin is at the center of image h.*

Examples:

```
>>> import Numeric
>>> f = Numeric.zeros((5,5))
>>> f[2,2] = 1
>>> print f
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 1 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
>>> h = Numeric.array([[1,2,3],[4,5,6]])
>>> print h
[[1 2 3]
 [4 5 6]]
>>> a = iaconv(f,h)
>>> print a
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 1 2 3 0 0]
 [0 0 4 5 6 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
>>> f = Numeric.array([[1,0,0,0],[0,0,0,0]])
>>> print f
[[1 0 0 0]
 [0 0 0 0]]
>>> h = Numeric.array([1,2,3])
```

```
>>> print h
[1 2 3]
>>> a = iaconv(f,h)
>>> print a
[[1 2 3 0 0 0]
 [0 0 0 0 0 0]]
>>> f = Numeric.array([[1,0,0,0,0,0],[0,0,0,0,0,0]])
>>> print f
[[1 0 0 0 0 0]
 [0 0 0 0 0 0]]
>>> h = Numeric.array([1,2,3,4])
>>> print h
[1 2 3 4]
>>> a = iaconv(f,h)
>>> print a
[[1 2 3 4 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
>>> f = iaread('cameraman.pgm')
>>> h = [[1,2,1],[0,0,0],[-1,-2,-1]]
>>> g = iaconv(f,h)
>>> gn = ianormalize(g, [0,255])
>>> iashow(gn)
```

(gn)

Equation:

$$
\begin{aligned}
(f * h)(x,y) &= \frac{1}{NM} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} f_e(i,j) h_e(x-i, y-j) \\
f_e(x,y) &= \begin{cases} f(x,y), & 0 \le x \le A-1 \quad and \quad 0 \le y \le B-1 \\ 0, & A \le x \le N-1 \quad or \quad B \le y \le M-1 \end{cases} \\
h_e(x,y) &= \begin{cases} f(x,y), & 0 \le x \le C-1 \quad and \quad 0 \le y \le D-1 \\ 0, & C \le x \le N-1 \quad or \quad D \le y \le M-1 \end{cases} \\
N &\le A + C - 1 \\
M &\le B + D - 1
\end{aligned}
$$

See also: *iaconv* (C.8.2)

C.8.3 ialogfilter

Purpose: *Laplacian of Gaussian filter.*

Synopsis: `g = ialogfilter(f, sigma)`

Input:

`f`	*<type 'array'> input image*
`sigma`	*<type 'float'> scaling factor*

Output:

g *<type 'array'>*

Description: *Filters the image f by the Laplacian of Gaussian (LoG) filter with parameter sigma. This filter is also known as the Marr-Hildreth filter. Obs: to better efficiency, this implementation computes the filter in the frequency domain.*

Examples:

```
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> g07 = ialogfilter(f, 0.7)
>>> iashow(g07)
>>> iashow(g07 > 0)
```

(f)

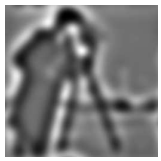
(g07)

(g07 > 0)

```
>>> import Numeric
>>> g5 = ialogfilter(f, 5)
>>> iashow(g5)
>>> iashow(g5 > 0)
>>> g10 = ialogfilter(f, 10)
>>> iashow(g10)
>>> iashow(g10 > 0)
```

(g5)

(g5 > 0)



(g10)

(g10 > 0)

C.8.4 iapconv

Purpose: *2D Periodic convolution.*

Synopsis: `g = iapconv(f, h)`

Input:

f *<type 'array'> Input image.*

h *<type 'array'> PSF (point spread function), or kernel. The origin is at the array center.*

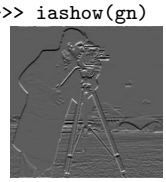
Output:

g *<type 'array'>*

Description: *Perform a 2D discrete periodic convolution. The kernel origin is at the center of image h. Both image and kernel are periodic with same period. Usually the kernel h is smaller than the image f, so h is padded with zero until the size of f.*

Examples:

```
>>> import Numeric
>>> f = Numeric.zeros((5,5))
>>> f[2,2] = 1
>>> print f
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 1 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
>>> h = Numeric.array([[1,2,3],[4,5,6]])
>>> print h
[[1 2 3]
 [4 5 6]]
>>> a = iapconv(f,h)
>>> print a
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  1.  2.  3.  0.]
 [ 0.  4.  5.  6.  0.]
 [ 0.  0.  0.  0.  0.]]
>>> f = Numeric.array([[1,0,0,0],[0,0,0,0]])
>>> print f
[[1 0 0 0]
 [0 0 0 0]]
>>> h = Numeric.array([1,2,3])
>>> print h
[1 2 3]
>>> a = iapconv(f,h)
>>> print a
[[ 2.  3.  0.  1.]
 [ 0.  0.  0.  0.]]
>>> f = Numeric.array([[1,0,0,0,0,0],[0,0,0,0,0,0]])
>>> print f
[[1 0 0 0 0 0]
 [0 0 0 0 0 0]]
>>> h = Numeric.array([1,2,3,4])
>>> print h
[1 2 3 4]
>>> a = iapconv(f,h)
>>> print a
[[ 2.  3.  4.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.]]
>>> f = iaread('cameraman.pgm')
>>> h = [[1,2,1],[0,0,0],[-1,-2,-1]]
>>> g = iapconv(f,h)
>>> gn = ianormalize(g, [0,255])
>>> iashow(gn)
```



(gn)

Equation:

$$\begin{aligned}(f *_N h)(x) &= \sum_{i=0}^{N-1} f(i)h(mod(x-i,N)) \\ f(x) &= f(x+kN), h(x) = h(x+kN)\end{aligned}$$

$$mod(x,N) = x - N\lfloor\frac{x}{N}\rfloor$$

$$(f *_{(N,M)} h)(x,y) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} f(i,j)h(mod(x-i,N), mod(y-j,M))$$

See also: *iaconv* (C.8.2), *iaptrans* (C.6.3)

C.8.5 iasobel

Purpose: *Sobel edge detection.*

Synopsis: `mag, theta = iasobel(f)`

Input:
`f` *<type 'array'> input image*

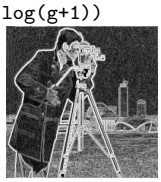
Output:
`mag, theta` *<type 'array'>*

Description: *Computes the edge detection by Sobel. Compute magnitude and angle.*

Examples:

```
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> (g,a) = iasobel(f)
>>> iashow(g)
>>> iashow(Numeric.log(g+1))
```

(g)



(Numeric.log(g+1))

```
>>> iashow(g > 150)
>>> i = Numeric.nonzero(Numeric.ravel(g > 150))
>>> z = Numeric.zeros(a.shape)
>>> Numeric.put(Numeric.ravel(z), i, Numeric.take(Numeric.ravel(a), i))
>>> iashow(z)
```


(g > 150)


(z)

Equation:

$$
\begin{aligned}
f_{sobel} &= \sqrt{f_x^2 + f_y^2} \\
f_{theta} &= \arctan\left(\frac{f_y}{f_x}\right) \\
f_x &= (f *_{(N,M)} S_x) \\
f_y &= (f *_{(N,M)} S_y) \\
S_x &= \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \\
S_y &= \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}
\end{aligned}
$$

C.8.6 iavarfilter

Purpose: *Variance filter.*

Synopsis: `g = iavarfilter(f, h)`

Input:

`f` *<type 'array'> input image.*
`h` *<type 'array'> scaling factor.*

Output:

`g` *<type 'array'>*

Description: *Computes the variance on the neighborhood of the pixel. The neighborhood is given by the set marked by the kernel elements.*

Examples:

```
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> g = iavarfilter(f, [[0,1,0],[1,1,1],[0,1,0]])
>>> iashow(g)
```


(f)


(g)

C.9 Automatic Thresholding Techniques

C.9.1 iaotsu

Purpose: *Thresholding by Otsu.*

Synopsis: `t, eta = iaotsu(f)`

Input:

`f` *<type 'array'> input image.*

Output:

`t, eta` *<type 'float'> Maximum of this result is the thresholding.*

Description: *Compute the automatic thresholding level of a gray scale image based on the Otsu method.*

Examples:

```
>>> import Numeric
>>> f = iaread('cookies.pgm')
>>> iashow(f)
>>> (t, eta) = iaotsu(f)
>>> print 'threshold at %f, goodness=%f' %(t, eta)
threshold at 90.000000, goodness=0.938940
>>> iashow(f > t)
```

(f)

(f > t)

Equation:

$$
\begin{aligned}
H(u,v) &= \frac{1}{1+(\sqrt{2}-1)(\sqrt{(\frac{u}{N})^2+(\frac{v}{M})^2}\ \ t_c)^{2n}} \\
(u,v) &\in \{-\lfloor\frac{N}{2}\rfloor : N-\lfloor\frac{N}{2}\rfloor-1, -\lfloor\frac{M}{2}\rfloor : M-\lfloor\frac{M}{2}\rfloor-1\} \\
t_c &\in \{2 : \max\{N,M\}\}
\end{aligned}
$$

C.10 Measurements

C.10.1 iacolorhist

Purpose: *Color-image histogram.*

Synopsis: `hc = iacolorhist(f, mask=None)`

Input:

`f` *<type 'array'>*
`mask` *<type 'array'>*

Output:

`hc` *<type 'array'>*

Description: *Compute the histogram of a color image and return a graphical image suitable for visualization with the 3 marginal histograms: red-green at top-left, blue-green at top-right and red-blue at bottom-left. If the optional mask image is available, the histogram is computed only for those pixels under the mask.*

Examples:

```
>>> f = iaread('boat.ppm')
>>> iashow(f)
>>> hc = iacolorhist(f)
>>> iashow(hc)
>>> iashow(Numeric.log(hc+1))
```

(f) (hc) (Numeric.log(hc+1))

C.10.2 iahistogram

Purpose: *Image histogram.*

Synopsis: `h = iahistogram(f)`

Input:

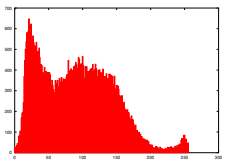
`f` *<type 'array'>*

Output:

`h` *<type 'array'>*

Examples:

```
>>> f = iaread('woodlog.pgm')
>>> iashow(f)
>>> h = iahistogram(f)
>>> g,d = iaplot(h)
>>> g('set data style boxes')
>>> g.plot(d)
```

(f) (h)

Equation:

$$h(v_i) = card\{p | f(p) = v_i\}$$

C.10.3 ialabel

Purpose: *Label a binary image.*

Synopsis: `g = ialabel(f)`

Input:

`f` *<type 'array'> input image*

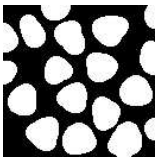
Output:

`g` *<type 'array'>*

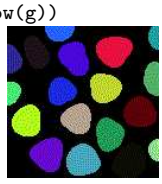
Description: *Creates an image by labeling the connect components of the input binary image. The background pixels (with value 0) are not labeled. The maximum label value in the output image gives the number of its connected components.*

Examples:

```
>>> f = Numeric.array([[0,1,0,1,1], [1,0,0,1,0]])
>>> print f
[[0 1 0 1 1]
 [1 0 0 1 0]]
>>> g = ialabel(f)
>>> print g
[[0 1 0 2 2]
 [3 0 0 2 0]]
>>> f = iaread('blobs.pbm')
>>> g = ialabel(f);
>>> nblobs = max(Numeric.ravel(g))
>>> print nblobs
18
>>> iashow(f)
>>> iashow(ialblshow(g))
```



(f)



(ialblshow(g))

C.10.4 iarec

Purpose: *Reconstruction of a connect component.*

Synopsis: `g = iarec(f, seed)`

Input:

`f` *<type 'array'> input image*
`seed` *<type 'array'> seed coordinate*

Output:

`g` *<type 'array'>*

Description: *Extracts a connect component of an image by region growing from a seed.*

Examples:

```
>>> f = Numeric.array([[0,1,0,1,1], [1,0,0,1,0]])
>>> print f
[[0 1 0 1 1]
 [1 0 0 1 0]]
```

```
>>> g = iarec(f, [0,3])
>>> print g
[[0 0 0 1 1]
 [0 0 0 1 0]]
```

C.10.5 iastat

Purpose: *Calculates MSE, PSNR and Pearson correlation between two images.*

Synopsis: `MSE, PSNR, PC = iastat(f1, f2)`

Input:

`f1` *<type 'array'> Input image 1.*
`f2` *<type 'array'> Input image 2 (Ex.: input image 1 degraded).*

Output:

`MSE, PSNR, PC` *<type 'float'> Mean Square Error; Peak Signal Noise Ratio; Correlation of the product of the moments of Pearson (r). If r is +1 then there is a perfect linear relation between f1 and f2. If r is -1 then there is a perfect linear negative relation between f1 and f2.*

Description: *Calculates the mean square error (MSE), the peak signal noise ratio (PSNR), and the correlation of the product of the moments of Pearson (PC), between two images.*

Examples:

```
>>> f1 = Numeric.array([[2,5,3],[4,1,2]])
>>> print f1
[[2 5 3]
 [4 1 2]]
>>> f2 = Numeric.array([[4,9,5],[8,3,3]])
>>> print f2
[[4 9 5]
 [8 3 3]]
>>> (mse, psnr, pc) = iastat(f1, f2)
>>> print mse
7.5
>>> print psnr
0.697381335929
>>> print pc
0.964763821238
```

Equation:

$$MSE = \frac{1}{N_h N_w} \sum \left[(f_1 - f_2)^2 \right]$$

$$PSNR = 20 log_{10} \frac{Y_{peak}}{\sqrt{MSE}}$$

$$PC = r = \frac{\sum f_1 f_2 - \frac{\sum f_1 \sum f_2}{N}}{\sqrt{\left[\sum f_1^2 - \frac{(\sum f_1)^2}{N} \right] \cdot \left[\sum f_2^2 - \frac{(\sum f_2)^2}{N} \right]}}$$

C.11 Halftoning Approximation

C.11.1 iadither

Purpose: *Ordered Dither.*

Synopsis: `g = iadither(f, n)`

Input:

`f`	*<type 'array'> input image*
`n`	*<type 'array'> dimension of the base matrix*

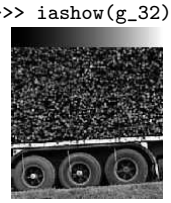
Output:

`g`	*<type 'array'>*

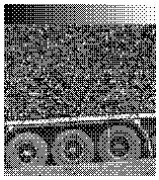
Description: *Ordered dither (Bayer 1973).*

Examples:

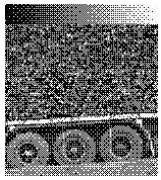
```
>>> f1 = iaramp([20,150], 256, [0,255])
>>> f2 = iaresize(iaread('woodlog.pgm'), [150,150])
>>> f = Numeric.concatenate((f1, f2))
>>> g_4 = iadither(f, 4)
>>> g_32 = iadither(f, 32)
>>> iashow(f)
>>> iashow(g_4)
>>> iashow(g_32)
```



(f)



(g_4)



(g_32)

C.11.2 iafloyd

Purpose: *Floyd-Steinberg error diffusion.*

Synopsis: `g = iafloyd(f)`

Input:

`f`	*<type 'array'> input image*

Output:

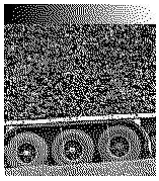
`g`	*<type 'array'>*

Description: *Floyd-Steinberg error diffusion (1976).*

Examples:

```
>>> f1 = iaramp([20,150], 256, [0,255])
>>> f2 = iaresize(iaread('woodlog.pgm'), [150,150])
>>> f = Numeric.concatenate((f1, f2))
>>> g = iafloyd(f)
>>> iashow(f)
>>> iashow(g)
```

(f)



(g)

C.12 Visualization

C.12.1 iadftview

Purpose: *Generate optical Fourier Spectrum for display from DFT data.*

Synopsis: `G = iadftview(F)`

Input:

`F` *<type 'array'> DFT complex data. F(1,1) is the center of the spectrum (u,v)=(0,0)*

Output:

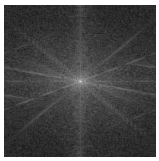
`G` *<type 'array'> uint8 image suitable for displaying*

Description: *Generate the logarithm of the magnitude of F, shifted so that the (0,0) stays at the center of the image. This is suitable for displaying only.*

Examples:

```
>>> import FFT
>>> f = iaread('cameraman.pgm')
>>> iashow(f)
>>> F = FFT.fft2d(f)
>>> Fv = iadftview(F)
>>> iashow(Fv)
```

(f)



(Fv)

Equation:

$$\begin{aligned} Gaux &= \log(|F_{xc,yc}| + 1) \\ xc &= \lfloor W/2 \rfloor \\ yc &= \lfloor H/2 \rfloor \\ G &= Gaux|_0^{255} \end{aligned}$$

See also: *iadft* (C.7.3), *iaidft* (C.7.11), *iafftshift* (C.7.5)

C.12.2 iagshow

Purpose: *Matrix of the image display.*

Synopsis: `g = iagshow(f)`

Input:

`f` *<type 'array'> Image.*

Output:

`g` *<type 'array'>*

Description: *Return the matrix of the image display.*

See also: *iashow* (C.12.6)

C.12.3 iaisolines

Purpose: *Isolines of a grayscale image.*

Synopsis: `g = iaisolines(f, nc=10, np=1)`

Input:

`f` *<type 'array'> Input image.*
`nc` *<type 'float'> Number of colors.*
`np` *<type 'float'> Number of pixels by isoline.*

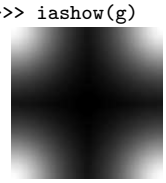
Output:

`g` *<type 'array'> Input image with color isolines.*

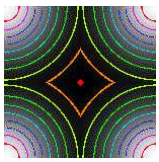
Description: *Shows lines where the pixels have same intensity with a unique color.*

Examples:

```
>>> f = ianormalize(iabwlp([150,150], 4, 1), [0,255]).astype('b')
>>> g = iaisolines(f, 10, 3)
>>> iashow(f)
>>> iashow(g)
```



(f)



(g)

C.12.4 ialblshow

Purpose: *Display a labeled image assigning a random color for each label.*

Synopsis: `g = ialblshow(f)`

Input:

`f` *<type 'array'> input image*

Output:

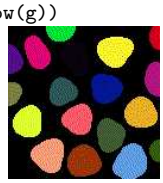
`g` *<type 'array'>*

Description: *Displays the labeled input image (uint8 or uint16) with a pseudo color where each label appears with a random color.*

Examples:

```
>>> f = iaread('blobs.pbm')
>>> g = ialabel(f);
>>> iashow(g)
>>> iashow(ialblshow(g))
```

(g)



(ialblshow(g))

C.12.5 iaplot

Purpose: *Plot a function.*

Synopsis: `g, d = iaplot(x=0, y=None, filename=None)`

Input:

x	*<type 'array'> x.*
y	*<type 'array'> f(x).*
filename	*<type 'str'> Name of the postscript file.*

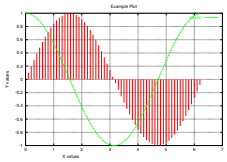
Output:

g	*<type 'str'> Gnuplot pointer.*
d	*<type 'str'> Gnuplot data.*

Description: *Plot a 2D function y=f(x).*

Examples:

```
>>> import Numeric
>>> x = Numeric.arange(0, 2*Numeric.pi, 0.1)
>>> y = Numeric.sin(x)
>>> g, d = iaplot(x,y)
>>> g('set data style impulses')
>>> g('set grid')
>>> g('set xlabel "X values" -20,0')
>>> g('set ylabel "Y values"')
>>> g('set title "Example Plot"')
>>> g.plot(d, 'cos(x)')
```



(d, 'cos(x)')

C.12.6 iashow

Purpose: *Image display.*

Synopsis: `g = iashow(f)`

Input:

`f` *<type 'array'> Image.*

Output:

`g` *<type 'array'>*

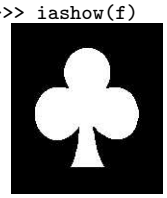
Description: *Display an image.*

Examples:

```
>>> f = iaread('boat.ppm')
>>> iashow(f)
```

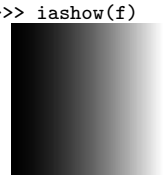
(f)

```
>>> f = iaread('club.pgm')
>>> iashow(f)
```



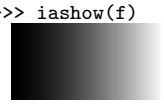
(f)

```
>>> import Numeric
>>> f = Numeric.resize(range(256), (256,256))
>>> iashow(f)
```



(f)

```
>>> import Numeric
>>> f = Numeric.resize(range(-100,100), (100,200))
>>> iashow(f)
```



(f)

```
>>> f = iaread('lenina.pgm')
>>> aux,k = iasobel(f)
>>> g = aux > 100
>>> iashow(f)
>>> iashow(g)
```

```
>>> iashow((f,g))
```

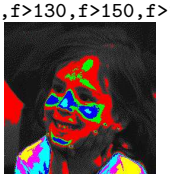
(f)

(g)

((f,g))

```
>>> f = iaread('lenina.pgm')
>>> iashow(f)
>>> iashow((f,f>100,f>130,f>150,f>180,f>220,f==255))
```

(f)



((f,f>100,f>130,f>150,f>180,f>220,f==255))

See also: *iagshow* (C.12.2)

C.12.7 iasplot

Purpose: *Plot a surface.*

Synopsis: `g, d = iasplot(x=0, y=None, z=None, filename=None)`

Input:

`x`	*<type 'array'> x range.*
`y`	*<type 'array'> y range.*
`z`	*<type 'array'> object function (z=f(x,y)).*
`filename`	*<type 'str'> Name of the postscript file.*

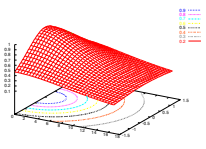
Output:

`g`	*<type 'str'> Gnuplot pointer.*
`d`	*<type 'str'> Gnuplot data.*

Description: *Plot a 3D function z=f(x,y).*

Examples:

```
>>> x = Numeric.arange(35)/2.0
>>> y = Numeric.arange(30)/10.0 - 1.5
>>> def z(x,y): return 1.0 / (1 + 0.01 * x**2 + 0.5 * y**2)
>>> g,d = iasplot(x, y, z)
```

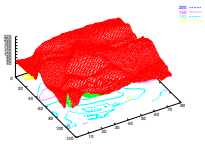


(z)

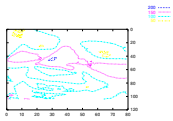
```
>>> f = iaread('lenina.pgm')
>>> k = f[90:195, 70:150]
>>> iashow(k)
>>> g,d = iasplot(k)
```

```
>>> g('set view 30,60')
>>> g.splot(d) # surface
>>> g('set view 0,90')
>>> g('set nosurface') # level curves
>>> g.splot(d)
```

(k)



(Surface of k)



(Level curves of k)

C.13 Functions

C.13.1 iaerror

Purpose: *Print message of error.*

Synopsis: `iaerror(msg)`

Input:

`msg` *<type 'str'> Error message.*

C.13.2 iahelp

Purpose: *Print the help of a function.*

Synopsis: `iahelp(obj)`

Input:

`obj` *<type 'NoneType'> Object name.*

Examples:

```
>>> iahelp(iacos)

        - Purpose
            Create a cossenoidal image.
        - Synopsis
            f = iacos(s, t, theta, phi=0)
        - Input
            s:     size: [rows cols].
            t:     period: in pixels.
            theta: spatial direction of the wave, in radians. 0 is a wave on
                   the horizontal direction.
            phi:   phase
        - Output
            f:
        - Description
            Generate a cosenosoid image of size s with amplitude 1, period
            T, phase phi and wave direction of theta. The output image is a
            double array.
```

```
        - Examples
            #
            #   example 1
            #
            import Numeric
            f = iacos((5,10), 6, Numeric.pi/3)
            print Numeric.array2string(f, precision=2, suppress_small=1)
            #
            #   example 2
            #
            f = iacos([256,256], 100, Numeric.pi/4)
            iashow(f)
            (g,d) = iaplot(f[0])
            showfig(f[0])
```

C.13.3 iatype

Purpose: *Print the source code of a function.*

Synopsis: `iatype(obj)`

Input:

`obj` *<type 'NoneType'> Object name.*

Examples:

```
>>> iatype(iacos)
def iacos(s, t, theta, phi=0):
    "
        - Purpose
            Create a cossenoidal image.
        - Synopsis
            f = iacos(s, t, theta, phi=0)
        - Input
            s:     size: [rows cols].
            t:     period: in pixels.
            theta: spatial direction of the wave, in radians. 0 is a wave on
                   the horizontal direction.
            phi:   phase
        - Output
            f:
        - Description
            Generate a cosenosoid image of size s with amplitude 1, period
            T, phase phi and wave direction of theta. The output image is a
            double array.
        - Examples
            #
            #   example 1
            #
            import Numeric
            f = iacos((5,10), 6, Numeric.pi/3)
            print Numeric.array2string(f, precision=2, suppress_small=1)
            #
            #   example 2
            #
            f = iacos([256,256], 100, Numeric.pi/4)
```

```
        iashow(f)
        (g,d) = iaplot(f[0])
        showfig(f[0])
"
from Numeric import cos, sin, pi
cols, rows = s[1], s[0]
(x, y) = iameshgrid(range(cols),range(rows))
freq = 1./t
fcols = freq * cos(theta)
frows = freq * sin(theta)
f = cos(2*pi*(fcols*x + frows*y) + phi)
return f
```

C.13.4 iaunique

Purpose: *Set unique.*

Synopsis: `t, i, j = iaunique(f)`

Input:

`f` *<type 'array'> Set initial.*

Output:

`t` *<type 'array'>*
`i` *<type 'array'>*
`j` *<type 'array'>*

Description: *Returns the set unique.*

Examples:

```
>>> c = Numeric.array([2, 10, 5, 5, 10, 7, 5])
>>> print c
[ 2 10  5  5 10  7  5]
>>> (u,i,j) = iaunique(c)
>>> print u
[ 2  5  7 10]
>>> print i
[0 6 5 4]
>>> print j
[0 3 1 1 3 2 1]
```

Apêndice D

Lessons

D.1 iaconvteo

Illustrate the convolution theorem

Reading and ROI selection. *The image is read and displayed*

```
>>> fin = iaread('lenina.pgm')
>>> iashow(fin)
>>> froi = iaroi(fin, (90,70), (200,180))
>>> iashow(froi)
```

(fin)

(froi)

Convolution with the Laplacian kernel. *The image is convolved (periodicaly) with the 3x3 Laplacian kernel*

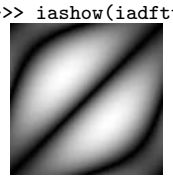
```
>>> import Numeric
>>> fd = froi.astype(Numeric.Float)
>>> h = Numeric.array([[-2,-1,0],[-1,0,1],[0,1,2]])
>>> g = iapconv(fd,h)
>>> iashow(g)
```

(g)

Equivalent filter in frequency domain. *The 3x3 kernel is zero padded to the size of the input image and periodicaly translated so that the center of the kernel stays at the top-left image corner. Its*

spectrum is visualized.

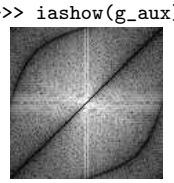
```
>>> hx = Numeric.zeros(Numeric.array(froi.shape))
>>> hx[:h.shape[0],:h.shape[1]] = h
>>> hx = iaptrans(hx,-Numeric.floor((Numeric.array(h.shape)-1)/2).astype(Numeric.Int)↩
)
>>> H = iadft(hx);
>>> iashow(iadftview(H))
```



(iadftview(H))

Filtering in the frequency domain. *The image is filtered by multiplying its DFT by the frequency mask computed in the previous step.*

```
>>> F = iadft(fd)
>>> G = F * H
>>> print "Is symmetrical:", iaisdftsym(G)
Is symmetrical: 1
>>> iashow(iadftview(G))
>>> g_aux = iaidft(G).real
>>> iashow(g_aux)
```



(iadftview(G))

(g_aux)

Comparing the results. *Both images, filtered by the convolution and filtered in the frequency domain are compared to see that they are the same. The small differencies are due to numerical precision errors.*

```
>>> e = abs(g - g_aux)
>>> print "Max error:", max(Numeric.ravel(e))
Max error: 4.25410817684e-11
```

D.2 iacorrdemo

Illustrate the Template Matching technique

Image read and pattern selection. *We have a gray scale image and a pattern extracting from the image.*

```
>>> import Numeric
>>> f = iaread('cameraman.pgm')
>>> f = Numeric.asarray(f).astype(Numeric.Float)
>>> iashow(f)
>>> w = f[25:25+17,106:106+17]
>>> iashow(w)
```


(f)

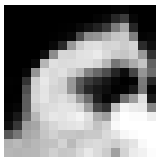
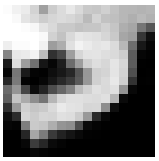

(w)

Image correlation. *Pure image correlation is not good for template matching because it depends on the mean gray value in the image, so light regions gives higher score than dark regions. A normalization factor can be used which improves the template matching.*

```
>>> w1 = w[::-1,::-1]
>>> iashow(w1)
>>> g = iapconv(f, w1)
>>> iashow(g)
>>> i = Numeric.ones(Numeric.shape(w1))
>>> fm2 = iapconv(f*f, i)
>>> g2 = g/Numeric.sqrt(fm2)
>>> iashow(g2)
>>> v, pos = max(Numeric.ravel(g2)), Numeric.argmax(Numeric.ravel(g2))
>>> (row, col) = iaind2sub(g2.shape, pos)
>>> print 'found best match at (%3.0f,%3.0f)' %(col,row)
found best match at (114, 33)
```


(w1)

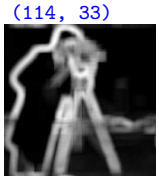

(g)


(g2)

Correlation index. *A better pattern matching is achievied subtrating the mean value on the image application.*

```
>>> import MLab
>>> n = Numeric.product(w.shape)
>>> wm = MLab.mean(Numeric.ravel(w))
>>> fm = 1.*iapconv(f,i)/n
>>> num = g - (n*fm*wm)
>>> iashow(num)
>>> den = Numeric.sqrt(fm2 - (n*fm*fm))
>>> iashow(den)
>>> cn = 1.*num/den
>>> iashow(cn)
>>> v, pos = max(Numeric.ravel(cn)), Numeric.argmax(Numeric.ravel(cn))
>>> (row, col) = iaind2sub(g2.shape, pos)
>>> print 'found best match at (%3.0f,%3.0f)' %(col,row)
found best match at (114, 33)
```


(num)


(den)


(cn)

D.3 iadftdecompose

Illustrate the decomposition of the image in primitive 2-D waves.

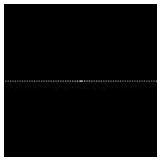
Step function image. *The image is created using.*

```
>>> import Numeric
>>> f = Numeric.ones((128, 128)) * 50
>>> x, y = [], map(lambda k:k%128, range(-32,32))
>>> for i in range(128): x = x + (len(y) * [i])
>>> y = 128 * y
>>> Numeric.put(f, iasub2ind([128,128], x, y), 200)
>>> iashow(f)
```

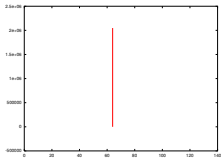
(f)

Discrete Fourier Transform. *The DFT is computed and displayed*

```
>>> F = iadft(f)
>>> E = iadftview(F)
>>> iashow(E)
>>> g,d = iaplot(iafftshift(F[:,0]).real)
>>> g('set data style impulses')
>>> g.plot(d)
```



(E)



(central_line)

D.4 iadftexamples

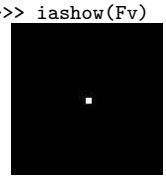
Demonstrate the DFT spectrum of simple synthetic images.

Constant image. *The DFT of a constant image is a single point at F(0,0), which gives the sum of all pixels in the image.*

```
>>> import Numeric, FFT
>>> f = 50 * Numeric.ones((10, 20))
>>> F = FFT.fft2d(f)
>>> aux = F.real > 1E-5
>>> r, c = iaind2sub([10, 20], Numeric.nonzero(Numeric.ravel(aux)))
>>> print r
[0]
>>> print c
[0]
>>> print F[r[0],c[0]]/(10.*20.)
(50+0j)
```

Square. *The DFT of a square image is a digital sync.*

```
>>> f = Numeric.zeros((128, 128))
>>> f[63:63+5,63:63+5] = 1
>>> iashow(f)
>>> F = FFT.fft2d(f)
>>> Fv = iadftview(F)
>>> iashow(Fv)
```

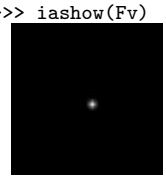


(f)

(Fv)

Pyramid. *The DFT of a pyramid is the square of the digital sync.*

```
>>> f = Numeric.zeros((128, 128))
>>> k = Numeric.array([[1,2,3,4,5,6,5,4,3,2,1]])
>>> k2 = Numeric.matrixmultiply(Numeric.transpose(k), k)
>>> f[63:63+k2.shape[0], 63:63+k2.shape[1]] = k2
>>> iashow(f)
>>> F = FFT.fft2d(f)
>>> Fv = iadftview(F)
>>> iashow(Fv)
```

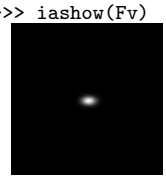


(f)

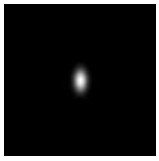
(Fv)

Gaussian. *The DFT of a Gaussian image is a Gaussian image.*

```
>>> f = iagaussian([128,128],[65,65],[[3*3,0],[0,5*5]])
>>> fn = ianormalize(f,[0,255])
>>> iashow(fn)
>>> F = FFT.fft2d(f)
>>> Fv = iadftview(F)
>>> iashow(Fv)
```



(fn)

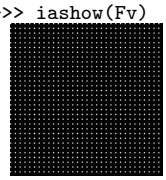


(Fv)

Impulse. *The DFT of an impulse image is an impulse image.*

```
>>> f = iacomb((128,128), (4,4), (0,0))
>>> fn = ianormalize(f, (0,255))
>>> iashow(fn)
>>> F = FFT.fft2d(f)
```

```
>>> Fv = iadftview(F)
>>> iashow(Fv)
```



(fn)

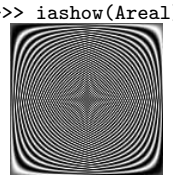
(Fv)

D.5 iadftmatrixexamples

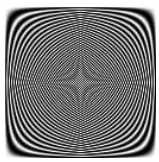
Demonstrate the kernel matrix for the DFT Transform.

Kernel images generated. *Imaginary and real parts of the DFT kernel.*

```
>>> A = iadftmatrix(128)
>>> Aimag, Areal = A.imag, A.real
>>> iashow(Aimag)
>>> iashow(Areal)
```



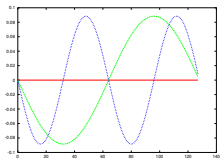
(Aimag)



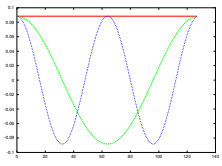
(Areal)

Three first lines. *Three first lines from imaginary and real parts of the kernel matrix. Observe the increasing frequencies of the senoidals (imaginary part) and cossenoidals (real part).*

```
>>> g,i1 = iaplot(Aimag[0,:])
>>> g,i2 = iaplot(Aimag[1,:])
>>> g,i3 = iaplot(Aimag[2,:])
>>> g.plot(i1,i2,i3)
>>> g,r1 = iaplot(Areal[0,:])
>>> g,r2 = iaplot(Areal[1,:])
>>> g,r3 = iaplot(Areal[2,:])
>>> g.plot(r1,r2,r3)
```



(i1,i2,i3)



(r1,r2,r3)

D.6 iadftscaleproperty

Illustrate the scale property of the Discrete Fourier Transform.

Image read, ROI selection and display. *The image is read and a small portion (64x64) is selected.*

```
>>> f = iaread('cameraman.pgm')
>>> froi = f[19:19+64,99:99+64] # ROI selection
>>> iashow(f)
>>> iashow(froi)
```


(f)


(froi)

DFT of the ROI image. *The DFT of the ROI image is taken and its spectrum is displayed*

```
>>> import Numeric, FFT
>>> fd = froi.astype(Numeric.Float)
>>> F = FFT.fft2d(fd) # F is the DFT of f
>>> iashow(froi)
>>> iashow(iadftview(F))
```


(froi)

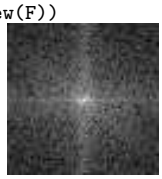

(iadftview(F))

Image expansion (without interpolation) and DFT. *The image is expanded by 4, but filling the new pixels with 0*

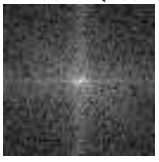
```
>>> fx4 = Numeric.zeros(4*Numeric.array(froi.shape)) # size is 4 times larger
>>> fx4[::4,::4] = froi                              # filling the expanded image
>>> iashow(froi)
>>> iashow(fx4)
```

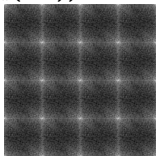

(froi)


(fx4)

DFT of the expansion without interpolation. *the resulting DFT is a periodical replication of the original DFT.*

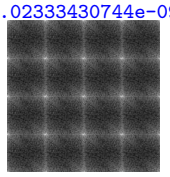
```
>>> fdx4 = fx4.astype(Numeric.Float)
>>> Fx4 = FFT.fft2d(fdx4) # Fx4 is the DFT of fx4 (expanded f)
>>> iashow(iadftview(F))
>>> iashow(iadftview(Fx4))
```


(iadftview(F))


(iadftview(Fx4))

Comparing in the frequency domain.. *Alternatively, the original DFT (F) is replicated by 4 in each direction and compared with the DFT of the expanded image. For quantitative comparison, both the sum of the absolute errors of all pixels is computed and displayed.*

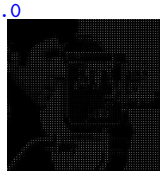
```
>>> aux = Numeric.concatenate((F,F,F,F))
>>> FFx4 = Numeric.concatenate((aux,aux,aux,aux), 1) # replicate the DFT of f
>>> iashow(iadftview(FFx4))
>>> diff = abs(FFx4 - Fx4)                          # compare the replicated DFT wit↩
h DFT of expanded f
>>> print Numeric.sum(Numeric.ravel(diff))          # print the error signal power
9.02333430744e-09
```



(iadftview(FFx4))

Comparing in the spatial domain..

```
>>> ffdx4 = FFT.inverse_fft2d(FFx4)
>>> fimag = ffdx4.imag
>>> print Numeric.sum(Numeric.ravel(fimag))
5.04870979341e-29
>>> ffdx4 = Numeric.floor(0.5 + ffdx4.real) # round
>>> iashow(ffdx4.astype(Numeric.Int))
>>> error = abs(fdx4 - ffdx4)
>>> print Numeric.sum(Numeric.ravel(error))
0.0
```



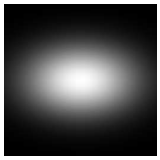
(ffdx4.astype(Numeric.Int))

D.7 iagenimages

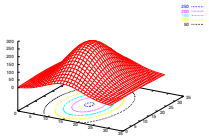
Illustrate the generation of different images

Gaussian images. *A gaussian image is controled by its mean and variance.*

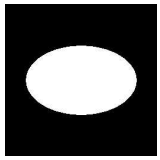
```
>>> f1 = iagaussian([256,256], [128,128], [[50*50,0],[0,80*80]])
>>> fn = ianormalize(f1, [0,255])
>>> iashow(fn)
>>> g,d = iasplot(iaresize(fn, [32,32]))
>>> iashow(fn > 128)
```



(fn)



(fn)

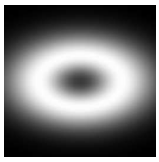


(fn > 128)

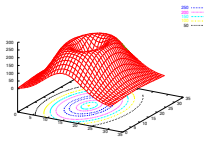
Difference of two Gaussian images. *The difference of two gaussian images gives a Laplacian image.*

```
>>> f2 = iagaussian([256,256], [128,128], [[25*25,0],[0,40*40]])
>>> fn2 = ianormalize(f2, [0,255])
>>> f = f1 - f2/5.
>>> fn = ianormalize(f, [0,255])
>>> iashow(fn2)
>>> iashow(fn)
>>> g,d = iasplot(iaresize(fn, [32,32]))
>>> iashow(fn > 230)
```

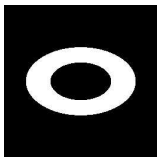
(fn2)



(fn)



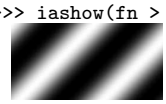
(fn)



(fn > 230)

Sinusoidal images. *A bidimensional sinusoidal image depends on its period, phase, and direction of the wave.*

```
>>> import Numeric
>>> f1 = iacos([128,256], 100, Numeric.pi/4, 0)
>>> fn = ianormalize(f1, [0,255])
>>> iashow(fn)
>>> iashow(fn > 128)
```



(fn)

(fn > 128)

Multiplication of two sinusoidal images. *The multiplication of two sinusoidal images gives a kind of a grid of hills and depressions.*

```
>>> f2 = iacos([128,256], 40, Numeric.pi/8, 0)
>>> fn2 = ianormalize(f2, [0,255])
>>> f = f1 * f2
>>> fn = ianormalize(f, [0,255])
>>> iashow(fn2)
>>> iashow(fn)
>>> iashow(fn > 128)
```

(fn2)

(fn)

(fn > 128)

Another example of multiplication of two sinusoidal images. *In this case, the waves are orthogonal to each other.*

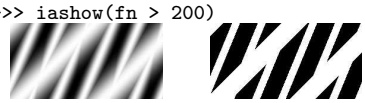
```
>>> f3 = iacos([128,256], 40, (Numeric.pi/8)+(Numeric.pi/2), 0)
>>> fn2 = ianormalize(f3, [0,255])
>>> f = f2 * f3
>>> fn = ianormalize(f, [0,255])
>>> iashow(fn2)
```

```
>>> iashow(fn)
>>> iashow(fn > 190)
```

(fn2) (fn) (fn > 190)

Maximum of two sinusoidal images. *The maximum of two sinusoidal images gives a kind of a surface of the union of both waves. They look like pipes*

```
>>> f = Numeric.maximum(f1, f2)
>>> fn = ianormalize(f, [0,255])
>>> iashow(fn)
>>> iashow(fn > 200)
```



(fn) (fn > 200)

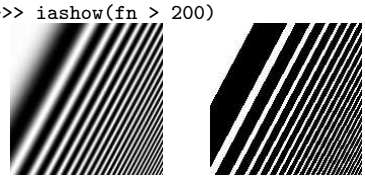
Sinusoidal where x and y are multiplied. *If the arguments of the cos are multiplied instead of added as before, we have a rather interesting pattern. It is important to remember that the proper sinusoidal image that are related to Fourier transforms are the bidimensional sinusoide shown earlier*

```
>>> x,y = iameshgrid(range(256), range(256))
>>> f = Numeric.cos((2*Numeric.pi*x/256) * (2*Numeric.pi*y/256))
>>> fn = ianormalize(f, [0,255])
>>> iashow(fn)
>>> iashow(fn > 200)
```

(fn) (fn > 200)

Sinusoidal with varying period. *If the arguments of the cos are taking at the power of a number, we have a varying period effect*

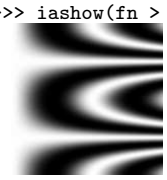
```
>>> x,y = iameshgrid(range(150), range(150))
>>> f = Numeric.cos(2*Numeric.pi* (x/80. + y/150.)**3)
>>> fn = ianormalize(f, [0,255])
>>> iashow(fn)
>>> iashow(fn > 200)
```



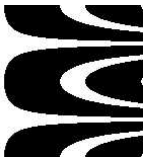
(fn) (fn > 200)

Sinusoidal with sinusoidal period. *In this case the arguments of the cos are the multiplication of x and cos(y)*

```
>>> x,y = iameshgrid(range(150), range(150))
>>> f = Numeric.cos(2*Numeric.pi*(x/80.) * Numeric.cos(2*Numeric.pi*(y/150.)))
>>> fn = ianormalize(f, [0,255])
>>> iashow(fn)
>>> iashow(fn > 200)
```



(fn)

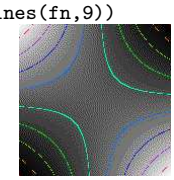


(fn > 200)

Saddle point function. *The multiplication of x and y gives a surface with saddle point*

```
>>> x,y = iameshgrid(range(-75,75), range(-75,75))
>>> f = x * y
>>> fn = ianormalize(f, [0,255]).astype('b')
>>> iashow(fn)
>>> iashow(iaisolines(fn,9))
```

(fn)



(iaisolines(fn,9))

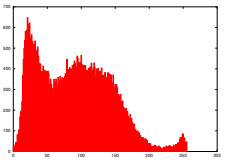
D.8 iahisteq

Illustrate how to make a histogram equalization

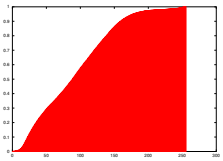
Computing the normalized cumulative histogram (NCH). *...*

```
>>> import Numeric, MLab
>>> f = iaread('woodlog.pgm')
>>> iashow(f)
>>> h = iahistogram(f)
>>> g,d1 = iaplot(h)
>>> g('set data style boxes')
>>> g.plot(d1)
>>> nch = MLab.cumsum(h) / (1.*Numeric.product(f.shape))
>>> g,d2 = iaplot(nch)
>>> g('set data style boxes')
>>> g.plot(d2)
```

(f)



(h)

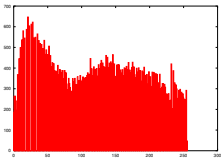


(nch)

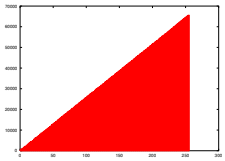
Use the NCH as intensity transform. *...*

```
>>> gn = (255 * iaapplylut(f, nch)).astype('b')
>>> iashow(gn)
>>> gh = iahistogram(gn)
>>> g,d1 = iaplot(gh)
>>> g('set data style boxes')
>>> g.plot(d1)
>>> gch = MLab.cumsum(gh) # cumulative histogram
>>> g,d2 = iaplot(gch)
>>> g('set data style boxes')
>>> g.plot(d2)
```

(gn)



(gh)



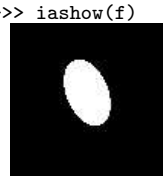
(gch)

D.9 iahotelling

Illustrate the Hotelling Transform

Image creation. *A binary image with an ellipsis*

```
>>> f = iagaussian([100,100], [45,50], [[20,5],[5,10]]) > 0.0000001
>>> iashow(f)
```



(f)

Feature vector. *The coordinates of each 1 pixel are the features: cols and rows coordinates. The mean value is the centroid.*

```
>>> import Numeric, MLab
>>> (rows, cols) = iaind2sub(f.shape, Numeric.nonzero(Numeric.ravel(f)))
>>> x = Numeric.concatenate((cols[:,Numeric.NewAxis], rows[:,Numeric.NewAxis]), 1)
>>> mx = MLab.mean(x)
>>> print mx
[50 45]
```

Computing eigenvalues and eigenvectors. *The eigenvalues and eigenvectors are computed from the covariance. The eigenvalues are sorted in decrescent order*

```
>>> Cx = MLab.cov(x)
>>> print Cx
[[  59.31288344   29.65644172]
 [  29.65644172  117.32924335]]
>>> [aval, avec] = MLab.eig(Cx)
>>> aux = Numeric.argsort(aval)[::-1]
>>> aval = MLab.diag(Numeric.take(aval, aux))
>>> print aval
```

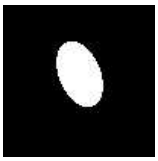
```
[[ 129.8057478     0.        ]
 [   0.          46.83637899]]
>>> avec = Numeric.take(avec, aux)
>>> print avec
[[ 0.38778193  0.92175115]
 [-0.92175115  0.38778193]]
```

Measure the angle of inclination. *The direction of the eigenvector of the largest eigenvalue gives the inclination of the elongated figure*

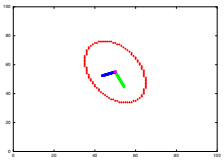
```
>>> a1 = Numeric.sqrt(aval[0,0])
>>> a2 = Numeric.sqrt(aval[1,1])
>>> vec1 = avec[:,0]
>>> vec2 = avec[:,1]
>>> theta = Numeric.arctan(1.*vec1[1]/vec1[0])*180/Numeric.pi
>>> print 'angle is %3f degrees' % (theta)
angle is -67.183445 degrees
```

Plot the eigenvectors and centroid. *The eigenvectors are placed at the centroid and scaled by the square root of its correspondent eigenvalue*

```
>>> iashow(f)
>>> x_,y_ = iaind2sub(f.shape, Numeric.nonzero(Numeric.ravel(iacontour(f))))
>>> g,d0 = iaplot(y_, 100-x_)
>>> # esboça os autovetores na imagem
>>> x1, y1 = Numeric.arange(mx[0], 100-mx[0]+a1*vec1[0], 0.1), Numeric.arange(mx[1], ↩
mx[1]-a1*vec1[1],0.235)
>>> g,d1 = iaplot(x1, 100-y1) # largest in green
>>> x2, y2 = Numeric.arange(mx[0], mx[0]-a2*vec2[0],-0.1), Numeric.arange(mx[1], mx[1↩
]+a2*vec2[1],0.042)
>>> g,d2 = iaplot(x2, 100-y2) # smaller in blue
>>> g,d3 = iaplot(mx[0], 100-mx[1]) # centroid in magenta
>>> g('set data style points')
>>> g('set xrange [0:100]')
>>> g('set yrange [0:100]')
>>> g.plot(d0, d1, d2, d3)
```



(f)



(mx)

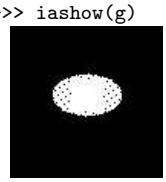
Compute the Hotelling transform. *The Hotelling transform, also called Karhunen-Loeve (K-L) transform, or the method of principal components, is computed below*

```
>>> y = Numeric.transpose(Numeric.matrixmultiply(avec, Numeric.transpose(x-mx)))
>>> my = MLab.mean(y)
>>> print my
[  6.13287652e-16  -1.10001668e-16]
>>> Cy = MLab.cov(y)
>>> print Cy
[[  1.29805748e+02   6.15367789e-15]
 [  6.15367789e-15   4.68363790e+01]]
>>> print Numeric.floor(0.5 + Numeric.sqrt(Cy))
```

```
[[ 11.   0.]
 [  0.   7.]]
```

Display the transformed data. *The centroid of the transformed data is zero (0,0). To visualize it as an image, the features are translated by the centroid of the original data, so that only the rotation effect of the Hotelling transform is visualized.*

```
>>> ytrans = Numeric.floor(0.5 + y + Numeric.resize(mx, (x.shape[0], 2)))
>>> g = Numeric.zeros(f.shape)
>>> i = iasub2ind(f.shape, ytrans[:,1], ytrans[:,0])
>>> Numeric.put(g, i, 1)
>>> iashow(g)
```



(g)

Image read and display. *The RGB color image is read and displayed*

```
>>> f = iaread('boat.ppm')
>>> iashow(f)
```

(f)

Extracting and display RGB components. *The color components are stored in the third dimension of the image array*

```
>>> r = f[:,:,0]
>>> g = f[:,:,1]
>>> b = f[:,:,2]
>>> iashow(r)
>>> iashow(g)
>>> iashow(b)
```

(r)

(g)

(b)

Feature vector: R, G and B values. *The features are the red, green and blue components. The mean vector is the average color in the image. The eigenvalues and eigenvectors are computed. The dimension of the covariance matrix is 3x3 as there are 3 features in use*

```
>>> x = 1.*Numeric.concatenate((Numeric.ravel(r)[:,Numeric.NewAxis], Numeric.ravel(g)↩
[:,Numeric.NewAxis], Numeric.ravel(b)[:,Numeric.NewAxis]), 1)
>>> mx = MLab.mean(x)
```

```
>>> print mx
[ 101.26770732   94.9191999    87.41316573]
>>> Cx = MLab.cov(x)
>>> print Cx
[[ 3497.95108557  3273.20413327  3086.73663437]
 [ 3273.20413327  3239.67139072  3103.11181452]
 [ 3086.73663437  3103.11181452  3032.39588874]]
>>> [aval, avec] = MLab.eig(Cx)
>>> aux = Numeric.argsort(aval)[::-1]
>>> aval = MLab.diag(Numeric.take(aval, aux))
>>> print aval
[[ 9572.66965395     0.             0.        ]
 [    0.           180.2290775      0.        ]
 [    0.             0.            17.11963358]]
>>> avec = Numeric.take(avec, aux)
>>> print avec
[[-0.59515464 -0.58010086 -0.55612404]
 [ 0.76785348 -0.20637147 -0.60647494]
 [ 0.2370485  -0.78796815  0.5682554 ]]
```

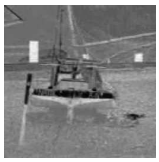
Hotelling transform. *The K-L transform is computed. The mean vector is zero and the covariance matrix is decorrelated. We can see the values of the standard deviation of the first, second and third components*

```
>>> y = Numeric.transpose(Numeric.matrixmultiply(avec, Numeric.transpose(x-mx)))
>>> my = MLab.mean(y)
>>> print my
[  1.89761523e-13  -1.30753601e-15  -9.03424412e-15]
>>> Cy = MLab.cov(y)
>>> print Cy
[[  9.57266965e+03   6.47705145e-12   7.30079934e-15]
 [  6.47705145e-12   1.80229077e+02   2.98030617e-13]
 [  7.30079934e-15   2.98030617e-13   1.71196336e+01]]
>>> print Numeric.floor(0.5 + Numeric.sqrt(Cy))
[[ 98.   0.   0.]
 [  0.  13.   0.]
 [  0.   0.   4.]]
```

Displaying each component of the transformed image. *The transformed features are put back in three different images with the g1 with the first component (larger variance) and g3 with the smaller variance*

```
>>> g1 = y[:,0]
>>> g2 = y[:,1]
>>> g3 = y[:,2]
>>> g1 = Numeric.reshape(g1, r.shape)
>>> g2 = Numeric.reshape(g2, r.shape)
>>> g3 = Numeric.reshape(g3, r.shape)
>>> iashow(g1)
>>> iashow(g2)
>>> iashow(g3)
```


(g1)


(g2)

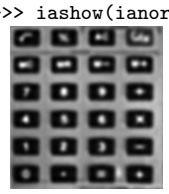

(g3)

D.10 iainversefiltering

Illustrate the inverse filtering for restoration.

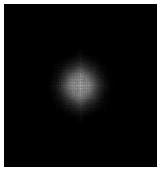
Original and distorted images. *The original image is corrupted using a low pass Butterworth filter with cutoff period of 8 pixels and order 4. This has the a similar effect of an out of focus image.*

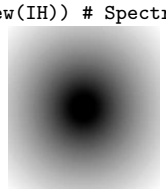
```
>>> import Numeric, FFT
>>> f = iaread('keyb.pgm').astype(Numeric.Float)
>>> F = FFT.fft2d(f)                # Discrete Fourier Transform of f
>>> H = iabwlp(f.shape, 16, 4)      # Butterworth filter, cutoff period 16, order 4
>>> G = F*H                         # Filtering in frequency domain
>>> g = FFT.inverse_fft2d(G).real   # inverse DFT
>>> iashow(ianormalize(g, [0,255])) # display distorted image
```


(ianormalize(g, [0,255]))

Spectrum of the blurred image and its inverse filter.

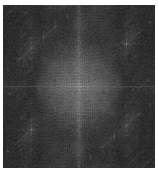
```
>>> iashow(iadftview(G))  # Spectrum of the corrupted image
>>> IH = 1.0/H            # The inverse filter
>>> iashow(iadftview(IH)) # Spectrum of the inverse filter
```


(iadftview(G))


(iadftview(IH))

Applying the inverse filter.

```
>>> FR = G*IH             # Inverse filtering
>>> iashow(iadftview(FR)) # Spectrum of the restored image
>>> fr = FFT.inverse_fft2d(FR).real
>>> iashow(fr)            # display the restored image
```

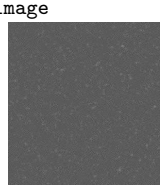

(iadftview(FR))

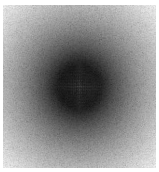

(fr)

Adding a little of noise. *The previous example is rather didactical. Just as an experience, instead of using the corrupted image with pixels values represented in double, we will use a integer truncation to the pixel values.*

```
>>> gfix = Numeric.floor(g)                                          # truncat↩
e pixel values
>>> Gfix = FFT.fft2d(gfix)                                           # DFT of ↩
truncated filted image
>>> FRfix = Gfix * IH                                                # applyin↩
g the inverse filtering
>>> frfix = FFT.inverse_fft2d(FRfix).real
>>> iashow(gfix)
>>> iashow(ianormalize(frfix, [0,255]))                              # display↩
 the restored image
>>> iashow(iadftview(FRfix))                                         # spectru↩
m of the restored image
```

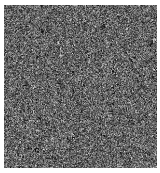

(gfix)

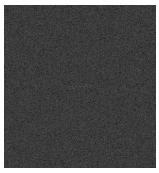

(ianormalize(frfix, [0,255]))


(iadftview(FRfix))

Why is the result so noisy?. *When the distorted image is rounded, it is equivalent of subtracting a noise from the distorted image. This noise has uniform distribution with mean 0.5 pixel intensity. The inverted filter is very high pass frequency, and at high frequency, the noise outcomes the power of the signal. The result is a magnification of high frequency noise.*

```
>>> import MLab
>>> fn = g - gfix                                                    # noise that w↩
as added
>>> print [MLab.mean(Numeric.ravel(fn)), MLab.min(Numeric.ravel(fn)), MLab.max(Numeri↩
c.ravel(fn))] # mean, minumum and maximum values
[0.4994099940305845, 9.2963516067356977e-06, 0.99996062920033069]
>>> iashow(ianormalize(fn, [0,255]))                                 # display nois↩
e
>>> FN = FFT.fft2d(fn)
>>> iashow(iadftview(FN))                                            # spectrum of ↩
the noise
>>> FNI = FN*IH                                                      # inverse filt↩
ering in the noise
>>> fni = FFT.inverse_fft2d(FNI).real
>>> print [MLab.min(Numeric.ravel(fni)), MLab.max(Numeric.ravel(fni))] # min and max ↩
of restored noise
[-15348654.784229765, 15787958.430655854]
>>> iashow(ianormalize(fni, [0,255]))                                # display rest↩
oration of noise
```


(ianormalize(fn, [0,255]))


(iadftview(FN))

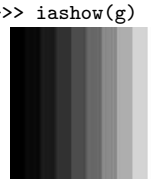

(ianormalize(fni, [0,255]))

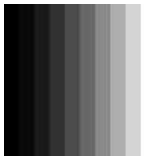
D.11 iait

Illustrate the contrast transform function

Identity intensity function. *The simplest intensity function is the identify s=v. This transform is a line of 45 degrees. It makes the output image the same as the input image.*

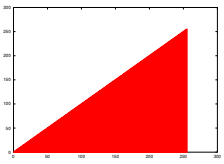
```
>>> import Numeric
>>> f = iaramp([100,100], 10, [0,255])
>>> it = Numeric.arange(256)
>>> g = iaapplylut(f, it)
>>> iashow(f)
>>> iashow(g)
```


(f)


(g)

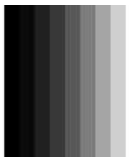
Visualizing the intensity transform function. *It is common to visualize the intensity transform function in a plot, T(v) x v.*

```
>>> g,d = iaplot(it)
>>> g('set data style boxes')
>>> g.plot(d)
```


(it)

Changing a particular color of an image. *To change a given gray scale value v1 of the input image to another gray scale value s1, we can change the identity function such that T(v1)=s1. Suppose we want to change any pixel with value 0 to 255.*

```
>>> it1 = 1*it
>>> it1[0] = 255
>>> print it1[0:5] # show the start of the intensity table
[255   1   2   3   4]
>>> g = iaapplylut(f, it1)
>>> iashow(g)
```

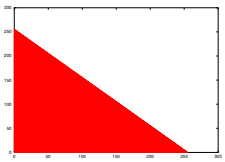


(g)

Negative of an image. *To invert the gray scale of an image, we can apply an intensity transform of the form T(v) = 255 - v. This transform will make dark pixels light and light pixels dark.*

```
>>> v = Numeric.arange(255)
>>> Tn = 255 - v
>>> f = iaread('cameraman.pgm')
>>> g = iaapplylut(f, Tn)
>>> iashow(g)
>>> g,d = iaplot(Tn)
>>> g('set data style boxes')
>>> g.plot(d)
```

(g)

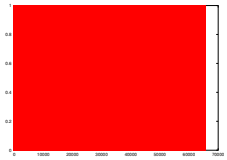


(Tn)

Thresholding. *A common operation in image processing is called thresholding. It assigns value 1 to all pixels equal or above a threshold value and assigns zero to the others. The threshold operator converts a gray scale image into a binary image. It can be easily implemented using the intensity transform. In the example below the threshold value is 128.*

```
>>> f = iaread('cameraman.pgm')
>>> thr = Numeric.concatenate((Numeric.zeros(128), Numeric.ones(128)))
>>> g = iaapplylut(f, thr)
>>> iashow(g)
>>> g,d = iaplot(g)
>>> g('set data style boxes')
>>> g.plot(d)
```

(g)



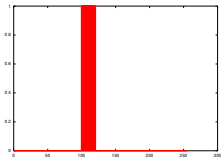
(thr)

Threshold band. *A variation of the thresholding is when the output is one for a range of the input gray scale values. In the example below, only pixels between values 100 and 120 are turned to one, the others to zero.*

```
>>> t1 = Numeric.zeros(256)
>>> t1[100:121] = 1
>>> g = iaapplylut(f, t1)
>>> iashow(g)
>>> g,d = iaplot(t1)
```

```
>>> g('set data style boxes')
>>> g.plot(d)
```

(g)

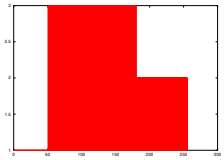


(t1)

Generalized Threshold. *A generalization of the previous case is to assign different values (classes) to different input ranges. This is a typical classification decision. For pixels from 0 and t1, assign 1; from t1 to t2, assign 2, etc. In the example below, the pixels are classified in three categories: class 1: dark pixels (between 0 and 50) corresponding to the cameraman clothing, class 3: medium gray pixels (from 51 to 180), and class 2: white pixels (from 181 to 255), corresponding to the sky.*

```
>>> t2 = Numeric.zeros(256)
>>> t2[0:51] = 1
>>> t2[51:181] = 3
>>> t2[181:256] = 2
>>> g = iaapplylut(f, t2)
>>> iashow(g)
>>> g,d = iaplot(t2)
>>> g('set data style boxes')
>>> g.plot(d)
```

(g)



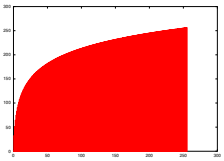
(t2)

Crescent functions. *When the intensity transform is a crescent function, if v1 ¿= v1 then T(v1) ¿= T(v2). In this class of intensity transforms, the order of the gray scale does not change, i.e., if a gray scale value is darker than another in the input image, in the transformed image, it will still be darker. The intensity order does not change. This particular intensity transforms are of special interest to image enhancing as our visual system does not feel confortable to non-crescent intensity transforms. Note the negation and the generalized threshold examples. The identity transform is a crescent function with slope 1. If the slope is higher than 1, for a small variation of v, there will be a larger variation in T(v), increasing the contrast around the gray scale v. If the slope is less than 1, the effect is opposite, the constrast around v will be decreased. A logarithm function has a higher slope at its beginning and lower slope at its end. It is normally used to increase the contrast of dark areas of the image.*

```
>>> v = Numeric.arange(256)
>>> Tlog = (256./Numeric.log(256.)) * Numeric.log(v + 1)
>>> f = ianormalize(iaread('lenina.pgm'), [0, 255])
>>> g = iaapplylut(f.astype('b'), Tlog)
>>> iashow(f)
>>> iashow(g)
>>> g,d = iaplot(Tlog)
>>> g('set data style boxes')
>>> g.plot(d)
```


(f)


(g)


(Tlog)

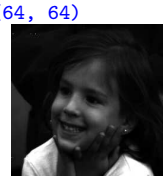
Normalization function. *Sometimes the input image has ranges in floating point or ranges not suitable for displaying or processing. The normalization function is used to fit the range of the gray scales, preserving the linear gray scale relationship of the input image. The intensity function that provides the normalization is a straight line segment which can be defined by its two extremities. Suppose the input gray scales ranges from m1 to m2 and we want to normalize the image to the range of M1 to M2. The two extremities points are (m1,M1) and (m2,M2). The equation for the intensity normalization function is point-slope form of the line equation: T(v) - M1 =(M2-M1)/(m2-m1)*(v - m1). The function ianormalize does that.*

D.12 iamagnify

Illustrate the interpolation of magnified images

Reading and ROI selection. *The image is read and a 64x64 ROI is selected and displayed*

```
>>> fin = iaread('lenina.pgm')
>>> iashow(fin)
>>> froi = fin[137:137+64,157:157+64]
>>> iashow(froi)
>>> print froi.shape
(64, 64)
```

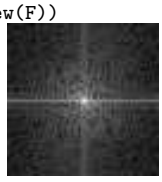

(fin)


(froi)

DFT. *The DFT of the small image is taken and its spectrum displayed*

```
>>> import Numeric, FFT
>>> fd = froi.astype(Numeric.Float)
>>> F = FFT.fft2d(fd)
>>> iashow(froi)
>>> iashow(iadftview(F))
```

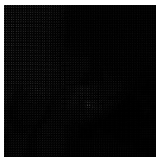

(froi)


(iadftview(F))

Expansion by 4 without interpolation. *The image is expanded by 4, but filling the new pixels with 0*

```
>>> fx4 = Numeric.zeros(4*Numeric.array(froi.shape))
>>> fx4[::4,::4] = froi
>>> iashow(froi)
>>> iashow(fx4)
```

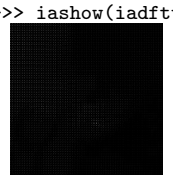
(froi)



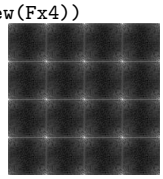
(fx4)

DFT of the expansion without interpolation. *Using the expansion propertie of the DFT (only valid for the discrete case), the resulting DFT is a periodical replication of the original DFT.*

```
>>> fdx4 = fx4.astype(Numeric.Float)
>>> Fx4 = FFT.fft2d(fdx4)
>>> iashow(fx4)
>>> iashow(iadftview(Fx4))
```



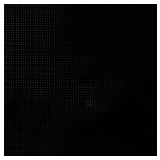
(fx4)



(iadftview(Fx4))

Filtering by mean filtering - nearest neighbor. *Filtering the expanded image using an average filter of size 4x4 is equivalent of applying a nearest neighbor interpolator. The zero pixels are replaced by the nearest non-zero pixel. This is equivalent to interpolation by pixel replication.*

```
>>> k = Numeric.ones((4,4))
>>> fx4nn = iapconv(fdx4, k)
>>> iashow(fx4)
>>> iashow(fx4nn.astype(Numeric.Int))
```

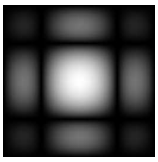


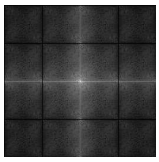
(fx4)

(fx4nn.astype(Numeric.Int))

Interpretation of the mean filtering in the frequency domain. *Filtering by the average filter in space domain is equivalent to filter in the frequency domain by the sync filter.*

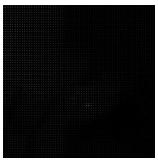
```
>>> kzero = Numeric.zeros(fx4.shape)
>>> kzero[0:4,0:4] = k
>>> K = FFT.fft2d(kzero)
>>> iashow(iadftview(K))
>>> Fx4nn = K * Fx4
>>> iashow(iadftview(Fx4nn))
```


(iadftview(K))


(iadftview(Fx4nn))

Filtering by pyramidal kernel, linear interpolation. *Filtering by a pyramidal kernel in space domain is equivalent to make a bi-linear interpolation. The zero pixels are replaced by a weighted sum of the neighbor pixels, the weight is inversely proportional to the non-zero pixel distance.*

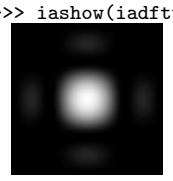
```
>>> klinear = Numeric.array([1,2,3,4,3,2,1])/4.
>>> k2dlinear = Numeric.matrixmultiply(Numeric.reshape(klinear, (7,1)), Numeric.resha↩
pe(klinear, (1,7)))
>>> fx4li = iapconv(fdx4, k2dlinear)
>>> iashow(fx4)
>>> iashow(fx4li.astype(Numeric.Int))
```

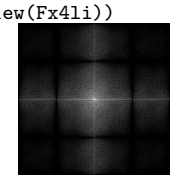

(fx4)


(fx4li.astype(Numeric.Int))

Interpretation of the pyramid filtering in the frequency domain. *Filtering by the pyramid filter in space domain is equivalent to filter in the frequency domain by the square of the sync filter.*

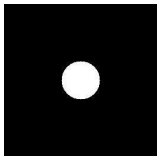
```
>>> klizero = Numeric.zeros(fx4.shape).astype(Numeric.Float)
>>> klizero[0:7,0:7] = k2dlinear
>>> Klinear = FFT.fft2d(klizero)
>>> iashow(iadftview(Klinear))
>>> Fx4li = Klinear * Fx4
>>> iashow(iadftview(Fx4li))
```

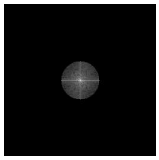

(iadftview(Klinear))


(iadftview(Fx4li))

Using an ideal filter. *Filtering by cutoff period of 8*

```
>>> H8 = iabwlp(fx4.shape, 8, 10000)
>>> iashow(iadftview(H8))
>>> G8 = Fx4 * H8
>>> iashow(iadftview(G8))
>>> g_ideal = FFT.inverse_fft2d(G8)
>>> print max(Numeric.ravel(g_ideal.imag))
3.02612295926e-14
>>> g_ideal = ianormalize(g_ideal.real, [0,255])
>>> iashow(g_ideal)
```

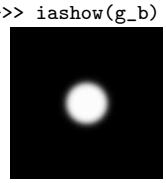

(iadftview(H8))

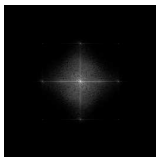

(iadftview(G8))


(g_ideal)

Using a Butterworth filter of order 5. *Filtering by cutoff period of 8*

```
>>> HB8 = iabwlp(fx4.shape, 8, 5)
>>> iashow(iadftview(HB8))
>>> GB = Fx4 * HB8
>>> iashow(iadftview(GB))
>>> g_b = FFT.inverse_fft2d(GB)
>>> print max(Numeric.ravel(g_b).imag)
3.21183896607e-14
>>> g_b = ianormalize(g_b.real, [0,255])
>>> iashow(g_b)
```


(iadftview(HB8))


(iadftview(GB))


(g_b)

Display all four for comparison. *Top-left: nearest neighbor, Top-right: linear, Bottom-left: ideal, Bottom-right: Butterworth*

```
>>> aux1 = Numeric.concatenate((fx4nn[0:256,0:256], fx4li[0:256,0:256]), 1)
>>> aux2 = Numeric.concatenate((g_ideal, g_b), 1)
>>> iashow(Numeric.concatenate((aux1, aux2)))
```


(Numeric.concatenate((aux1, aux2)))

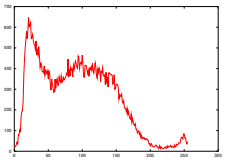
D.13 iaotsudemo

Illustrate the Otsu Thresholding Selection Method

Image read and histograming. *Gray scale image and its histogram*

```
>>> import Numeric
>>> f = iaread('woodlog.pgm');
>>> iashow(f)
>>> H = iahistogram(f)
>>> x = Numeric.arange(len(H))
>>> k = x[0:-1]
>>> g,d = iaplot(x, H)
```

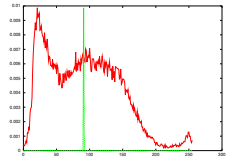
(f)



(H)

Normalized histogram. *If the histogram is divided by the number of pixels in the image, it can be seen as a probability distribution. The sum of each values gives one. The mean gray level can be computed from the normalized histogram.*

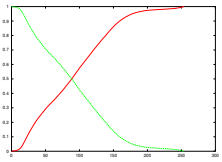
```
>>> import MLab
>>> h = 1.*H/Numeric.product(f.shape)
>>> print Numeric.sum(h)
1.0
>>> mt = Numeric.sum(x * h)
>>> st2 = Numeric.sum((x-mt)**2 * h)
>>> if abs(mt - MLab.mean(Numeric.ravel(f))) > 0.01: iaerror('error in computing mean↩
')
error in computing mean
>>> if abs(st2 - MLab.std(Numeric.ravel(f))**2) > 0.0001: iaerror('error in computing↩
 var')
error in computing var
>>> print 'mean is %2.2f, var2 is %2.2f' % (mt, st2)
mean is 91.03, var2 is 2873.86
>>> maux = 0 * h
>>> maux[int(mt)] = max(h)
>>> g,d1 = iaplot(x,h)
>>> g,d2 = iaplot(x,maux)
>>> g.plot(d1, d2)
```



(h)

Two classes. *Suppose the pixels are categorized in two classes: smaller or equal than the gray scale t and larger than t. The probability of the first class occurrence is the cummulative normalized histogram. The other class is the complementary class.*

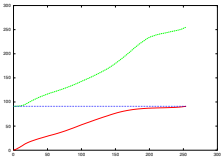
```
>>> w0 = MLab.cumsum(h[0:-1])
>>> aux = h[1::]
>>> w1aux = MLab.cumsum(aux[::-1])[::-1]
>>> w1 = 1 - w0
>>> if max(abs(w1-w1aux)) > 0.0001: iaerror('error in computing w1')
>>> g,d1 = iaplot(k,w0)
>>> g,d2 = iaplot(k,w1)
>>> g.plot(d1, d2)
```



(w0,w1)

Mean gray level of each class. *The mean gray level as a function of the thresholding t is computed and displayed below.*

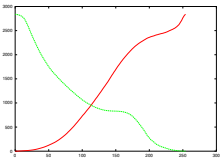
```
>>> m0 = MLab.cumsum(k * h[0:-1]) / (1.*w0)
>>> m1 = (mt - m0*w0)/w1
>>> aux = (k+1) * h[1::]
>>> m1x = MLab.cumsum(aux[::-1])[::-1] / (1.*w1)
>>> mm = w0 * m0 + w1 * m1
>>> if max(abs(m1-m1x)) > 0.0001: iaerror('error in computing m1')
>>> g,d1 = iaplot(k,m0)
>>> g,d2 = iaplot(k,m1)
>>> g,d3 = iaplot(k,mm)
>>> g.plot(d1,d2,d3)
```



(m0,m1)

Variance of each class. *The gray level variance as a function of the thresholding t is computed and displayed below.*

```
>>> s02 = MLab.cumsum((k-m0)**2 * h[0:-1]) / (1.*w0)
>>> aux = ((k+1)-m1)**2 * h[1::]
>>> s12 = MLab.cumsum(aux[::-1])[::-1] / (1.*w1)
>>> g,d1 = iaplot(k, s02)
>>> g,d2 = iaplot(k, s12)
>>> g.plot(d1, d2)
```

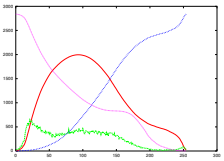


(s02)

Class separability. *The variance between class is a good measure of class separability. As higher this variance, the better the class clustering.*

```
>>> sB2 = w0 * ((m0 - mt)**2) + w1 * ((m1 - mt)**2)
>>> sBaux2 = w0 * w1 * ((m0 - m1)**2)
>>> if max(sB2-sBaux2) > 0.0001: iaerror('error in computing sB')
>>> v = max(sB2)
>>> t = (sB2.tolist()).index(v)
>>> eta = 1.*sBaux2[t]/st2
```

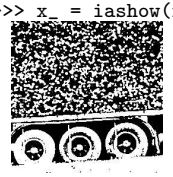
```
>>> print 'Optimum threshold at %f quality factor %f' % (t, eta)
Optimum threshold at 93.000000 quality factor 0.694320
>>> g,d1 = iaplot(k, sB2)
>>> g,d2 = iaplot(k, H[0:-1])
>>> g,d3 = iaplot(k, s02)
>>> g,d4 = iaplot(k, s12)
>>> g.plot(d1, d2, d3, d4)
```



(sB)

Thresholding. *The thresholded image is displayed to illustrate the result of the binarization using the Otsu method.*

```
>>> x_ = iashow(f > t)
```



(f > t)