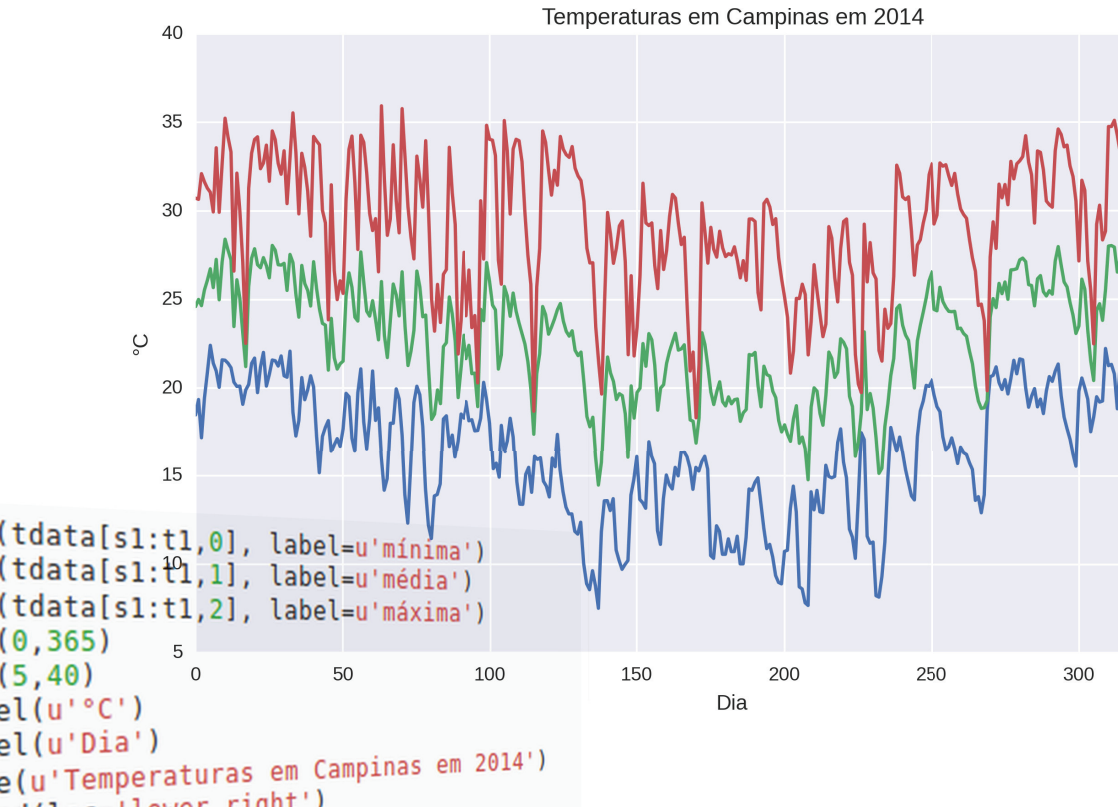
Documentos

ISSN 1677-9274
Dezembro, 2014
131

Introdução à computação científica com SciPy



Embrapa

ISSN 1677-9274
Dezembro, 2014

Empresa Brasileira de Pesquisa Agropecuária
Embrapa Informática Agropecuária
Ministério da Agricultura, Pecuária e Abastecimento

Documentos 131

Introdução à computação científica com SciPy

Thiago Teixeira Santos

Embrapa Informática Agropecuária
Campinas, SP
2014

Embrapa Informática Agropecuária
Av. André Tosello, 209 - Barão Geraldo
Caixa Postal 6041 - 13083-886 - Campinas, SP
Fone: (19) 3211-5700 - Fax: (19) 3211-5754
www.embrapa.br/informatica-agropecuaria
SAC: www.embrapa.br/fale-conosco/sac/

Comitê de Publicações

Presidente: *Silvia Maria Fonseca Silveira Massruhá*

Secretária: *Carla Cristiane Osawa*

Membros: *Adhemar Zerlotini Neto, Stanley Robson de Medeiros Oliveira, Thiago Teixeira Santos, Maria Goretti Gurgel Praxedes, Adriana Farah Gonzalez, Neide Makiko Furukawa, Carla Cristiane Osawa*

Membros suplentes: *Felipe Rodrigues da Silva, José Ruy Porto de Carvalho, Eduardo Delgado Assad, Fábio César da Silva*

Supervisor editorial: *Stanley Robson de Medeiros Oliveira, Neide Makiko Furukawa*

Revisor de texto: *Adriana Farah Gonzalez*

Normalização bibliográfica: *Maria Goretti Gurgel Praxedes*

Editoração eletrônica: *Neide Makiko Furukawa*

Imagens capa: *Thiago Teixeira Santos*

1ª edição
on-line 2014

Dados Internacionais de Catalogação na Publicação (CIP)
Embrapa Informática Agropecuária

Santos, Thiago Teixeira.
Introdução à computação científica com SciPy / Thiago Teixeira Santos. - Campinas : EmbrapaInformática Agropecuária, 2014.
76 p. : il. ; 14,8 cm x 21 cm. - (Documentos / Embrapa Informática Agropecuária, ISSN 1677-9274.

1. Computação científica. 2. Linguagem Python. I. Embrapa Informática Agropecuária. Título. II. Série.

CDD 006.6 (21.ed.)

Autor

Thiago Teixeira Santos
Cientista da computação, doutor em Ciências da Computação, pesquisador da Embrapa Informática Agropecuária, Campinas, SP

Apresentação

Quais são as instituições e grupos que obtêm o máximo da chamada Ciência dos Dados? Segundo Pedro Domingos, professor da Universidade de Washington, são aquelas que possuem uma infraestrutura que permite a exploração, de maneira fácil e eficiente, de diversos problemas de aprendizado de máquina. Tal estrutura deve promover a experimentação com vários algoritmos, métodos e fontes de dados, em uma colaboração sinérgica entre especialistas do domínio de aplicação, estatísticos, engenheiros e cientistas da computação.

Evidentemente, a Empresa Brasileira de Pesquisa Agropecuária (Embrapa) pode e deve se tornar uma organização capaz de tirar grande proveito desta nova era da ciência, beneficiando-se do conhecimento, da experiência e dos grandes volumes de dados obtidos em mais de 40 anos de atividade em pesquisa e desenvolvimento. Iniciativas como o arranjo DataExp, destinado ao armazenamento e processamento de dados experimentais da Embrapa, são esforços nessa direção.

O presente texto é outra iniciativa relacionada à Ciência dos Dados. Trata-se de um tutorial que apresenta ao leitor uma das plataformas mais populares atualmente, para o processamento de dados científicos: SciPy. Baseada na linguagem Python, essa plataforma apresenta diversas ferramentas como análise estatística, processamento de sinais, reconhecimento de padrões, álgebra linear e matemática simbólica, entre muitas outras. A tradicional revista Nature tem demonstrado entusiasmo por tal ambiente em artigos recentes, ver volume 514 de 2014 e volume 518 de 2015. Os IPython notebooks, discutidos neste tutorial e destacados pela Nature, podem se tornar uma língua franca entre grupos multidisciplinares de cientistas, que precisam compartilhar dados, algoritmos e resultados.

Esperamos que este tutorial seja útil a pesquisadores e técnicos em diversas instituições que tenham o desafio de realizar pesquisa e desenvolvimento em temas complexos que exigem computação científica desenvolvida e partilhada por equipes multidisciplinares. Este é um dos desafios da Embrapa e o desafio de toda organização vivendo na era da Ciência dos Dados.

Kleber Xavier Sampaio de Souza

Chefe-geral

Embrapa Informática Agropecuária

Sumário

Introdução à computação científica com SciPy

Thiago Teixeira Santos

1 Introdução

Pesquisadores empregam, cotidianamente, técnicas computacionais para simulação, visualização e análise de dados e teste de hipóteses. Assumindo uma visão Popperiana[1] da ciência, a computação auxilia o cientista no desenvolvimento de modelos com boa capacidade de predição, isto é, capazes não só de explicar os dados observados no passado, mas também prever fenômenos futuros (DHAR, 2013).

O presente material se originou em um tutorial sobre **computação científica com Python** regularmente oferecido pelo autor na Embrapa Informática Agropecuária, na forma de um curso introdutório com um dia de duração. Texto e tutorial são inspirados nas *Python scientific lecture notes* (HAENEL et al., 2013[2]) e fortemente recomendadas ao leitor interessado no tema. O presente texto (bem como o curso tutorial) não tem a pretensão de ser

[1] Karl Popper defendia que o critério principal para se avaliar um modelo era sua capacidade de prever fenômenos futuros, ou seja, seu **poder de predição**. Popper (2014) e Dhar (2013) para um visão moderna envolvendo computação e ciência dos dados.

[2] Alguns exemplos utilizados no presente texto são modificações de exemplos contidos nas *Python scientific lecture notes*, mas obedecem estritamente os termos da licença Creative Commons Atribuição 3.0 (CC BY 3.0) que garante o direito de remix, transformação e criação a partir do material para qualquer fim.

uma referência definitiva, nem prover uma cobertura exaustiva de toda a funcionalidade existente. Seu objetivo é introduzir os recursos encontrados no ambiente Python que podem atender às necessidades de diversos pesquisadores em suas atividades, envolvendo computação.

1.1 Organização, componentes e materiais

Este texto é organizado da seguinte forma: o restante da introdução (Seção 1) apresenta brevemente o conceito de ambiente de computação científica e algumas soluções existentes, finalizando com informações sobre a instalação do ambiente científico Python. A Seção 2 apresenta rudimentos mínimos da linguagem necessários ao pesquisador. A Seção 3 apresenta o interpretador IPython como ambiente interativo para computação e as vantagens de sua funcionalidade notebook para pesquisa. A Seção 4 introduz a representação utilizada para matrizes de dados (*arrays* multidimensionais). A Seção 5 apresenta recursos de *plotting* para visualização de dados. Finalmente, a Seção 6 introduz alguns dos componentes da biblioteca científica *SciPy Library* e a Seção 7 encerra o texto, apresentando mais referências e novas ferramentas promissoras no ambiente Python.

O conteúdo deste texto, incluindo o código fonte dos exemplos, pode ser obtido na forma de *IPython* notebooks na web[3].

1.2 Computação científica

Com o enorme aumento no volume de dados disponíveis aos cientistas devido à informatização e a novos sensores, o desenvolvimento de modelos preditivos vem sendo chamado de **ciência dos dados** (HEY et al., 2009; DHAR, 2013). O cientista necessita então de um ambiente no qual um vasto conjunto de ferramentas computacionais e estatísticas estejam disponíveis, auxiliando-o no desenvolvimento de resultados replicáveis, ou seja, um ambiente de **computação científica**.

Tal ambiente deveria prover fácil acesso a implementações de diversas ferramentas matemáticas em álgebra linear, estatística, processamento de sinais, visualização científica (*plotting* de gráficos 2D e 3D), cálculo diferencial e integral, otimização e probabilidade entre muitas outras. Contudo, o uso de tal ambiente deveria implicar em um tempo de codificação curto, permitindo que a exploração de ideias e a avaliação de modelos sejam realizadas rapidamente, além de produzir código e documentação legíveis para o cientista e seus pares, favorecendo a **reprodutibilidade** dos resultados(VANDEWALLE et al., 2009).

[3] Scipy-intro no GitHub. Disponível em: <https://github.com/thsant/scipy-intro>.

1.3 Soluções existentes

1.3.1 Linguagens compiladas como C/C++ e Fortran

A linguagem Fortran foi criada na década de 1950 como uma alternativa à linguagem *assembly* para computação científica e análise numérica (seu nome é derivado da expressão *Formula Translating System*). Ao longo de quase 60 anos, inúmeras bibliotecas de computação científica foram produzidas para a linguagem, como por exemplo as bibliotecas de álgebra linear, Basic Linear Algebra Subprograms (BLAS) e Linear Algebra Package (LAPACK). Com a criação da linguagem C em 1972 e da linguagem C++ em 1985, parte da comunidade científica passou a utilizá-las, e bibliotecas de computação científica foram desenvolvidas especialmente para elas, como por exemplo a GNU Scientific Library (GSL).

Embora capazes de produzir código extremamente eficiente quanto ao uso da capacidade dos processadores, essas linguagens apresentam tempos de desenvolvimento longos, marcados por ciclos de edição, compilação e execução de código e atividades de gerenciamento de memória e depuração. Mesmo operações simples, como a leitura de dados a partir de um arquivo, podem requerer diversas linhas de código. Outra desvantagem vem do fato de não fornecerem um modo **interativo** de programação, no qual pequenos trechos de código podem ser executados e seus resultados examinados prontamente pelo pesquisador.

1.3.2 MATLAB

MATLAB, nome derivado da expressão *matrix laboratory*, é um ambiente de computação científica que se originou dos esforços de Cleve Moler, professor da Universidade do Novo México, que desejava introduzir as ferra-

mentas de álgebra linear disponíveis nas bibliotecas LAPACK e EISPACK aos seus alunos, mas sem que estes necessitassem aprender Fortran. A MATLAB foi lançada como produto comercial em 1984. Desde então, diversas bibliotecas de computação científica, chamadas de *toolboxes*, passaram a ser fornecidas para MATLAB, em áreas tão diversas como processamento de imagens, bioinformática, econometria e sistemas de controle.

As duas principais desvantagens de MATLAB são:

1) Seu elevado custo, o que torna a redistribuição de código entre pesquisadores mais difícil, comprometendo a reprodutibilidade dos experimentos.

2) Sua linguagem, bastante limitada frente a linguagens de programação como Fortran, C ou Python.

1.3.3 Scilab, Octave e R

Scilab e Octave são ambientes de computação científica de código aberto que surgiram como alternativas ao MATLAB. R é um ambiente para computação estatística que surgiu como uma alternativa ao ambiente S, tornando-se uma das plataformas mais populares atualmente em estatística e análise de dados.

1.3.4 Python

Nos últimos anos, um ambiente para computação científica emergiu em torno da linguagem de programação Python. Seu nascimento pode ser traçado a partir do desenvolvimento do módulo Numarray, criado pela comunidade de astrofísica para a representação de matrizes n-dimensionais de modo similar a ambientes como MATLAB e Octave. Seu sucessor, o módulo **NumPy**, tornou-se o padrão para a representação de dados na forma de *arrays* n-dimensionais em Python. Originou-se daí uma pilha de software (*software stack*) dedicada à computação científica em Python, a **SciPy Stack**, bem como uma comunidade dedicada de desenvolvedores, reunidos regularmente em conferências *SciPy* em todo o mundo.

Uma das principais vantagens do ambiente SciPy é o uso da linguagem Python (OLIPHANT, 2007; PEREZ et al., 2011). Clara e elegante, a linguagem não tem uma curva íngreme de aprendizado como C ou Fortran.

Recursos avançados como gerenciamento automático de memória (*garbage collection*), tipagem dinâmica e introspecção facilitam e aceleram o desenvolvimento de código. Expressiva e madura, Python é uma linguagem usada no desenvolvimento de produtos e serviços no mundo todo e conta com uma extensa lista de módulos para programação de propósito geral, além dos componentes de computação científica. A Tabela 1 compara Python a outras seis linguagens em relação ao desempenho, à interatividade, à gratuidade e à disponibilidade de código. Recentemente, o uso de notebooks IPython (parte do ambiente Python de computação científica) foi destacado pela revista Nature como ferramenta para registro e reprodução de análises de dados (SHEN, 2014).

Tabela 1. Comparativo entre várias opções para computação científica.

	C/C++	Fortran	MATLAB	Octave	R	Python
Desempenho	+++	+++	+	+	+	++
Ambiente interativo	Não	Não	Sim	Sim	Sim	Sim
Gratuidade	Sim	Sim	Não	Sim	Sim	Sim
Código aberto	Sim	Sim	Não	Sim	Sim	Sim

Na primeira linha da Tabela 1, o sinal “+” é utilizado para sinalizar melhor desempenho (“+++” melhor, “+” pior). A classificação de desempenho é baseada no comparativo realizado pela equipe da linguagem Julia[4].

1.4 Componentes de um ambiente Python para computação científica

A comunidade SciPy assume que há um **núcleo duro** (*core*) de módulos que constitui um ambiente SciPy de computação científica. Fazem parte de tal núcleo a própria **linguagem Python**, o módulo **NumPy** para *arrays* n-dimensionais, o interpretador de comandos **IPython** e a **biblioteca SciPy**, um conjunto de módulos que provê uma vasta gama de rotinas científicas em estatística, otimização, interpolação e álgebra linear, entre outras. Também fazem parte do núcleo os módulos **Matplotlib** e **SymPy**. Matplotlib fornece rotinas para a produção de gráficos 2D e 3D para visua-

[4] Disponível em: <http://julialang.org/benchmarks/>.

lização científica. SymPy é um sistema para **álgebra computacional** que fornece ferramentas para cálculo diferencial e integral, manipulação de polinômios e equações diferenciais, entre outras.

Diversos outros módulos científicos estão disponíveis fora do *core*. Um dos mais populares é o módulo de aprendizado de máquina **Scikit-learn**. Tendo seu desenvolvimento liderado por pesquisadores ligados ao Institut National de Recherche en Informatique et en Automatique (INRIA), França, Scikit-learn provê ferramentas em regressão, classificação, *clustering*, redução de dimensionalidade e validação cruzada (*cross-validation*).

O presente trabalho irá introduzir, ao leitor, a linguagem Python, o ambiente IPython e os módulos NumPy, SciPy e Matplotlib. Por serem módulos fundamentais da computação científica em Python, muito pode ser realizado apenas com tais componentes e a compreensão de sua funcionalidade básica auxilia no trabalho com outros módulos, como a Scikit-learn.

1.5 Instalação do ambiente de computação científica Python

1.5.1 Sistemas Windows, Mac e Linux

Usuários de sistemas operacionais Microsoft Windows, Apple Mac e Linux podem considerar pacotes de instalação como o **Anaconda**[5] da Continuum Analytics ou o **Canopy**[6] da Enthought, Inc. Outra alternativa é a instalação individual de cada um dos pacotes da *SciPy Stack*, de acordo com as instruções atualizadas[7] encontras em SciPy.org.

1.5.2 Pacotes Linux

Usuários Linux podem preferir a utilização do sistema gerenciador de pacotes de sua distribuição. Por exemplo, em sistemas Debian/Ubuntu a instalação pode ser realizada com o uso do sistema `apt`:

[5] Anaconda. Disponível em: <http://continuum.io/downloads>.
[6] Canopy. Disponível em: <https://store.enthought.com/downloads>.
[7] SciPy. Disponível em: <http://www.scipy.org/install.html>.


```
$ sudo apt-get install ipython ipython-qtconsole ipython-notebook
$ sudo apt-get install python-scipy python-matplotlib python-matplotlib-data
```

1.5.3 Instalação com a ferramenta `pip`

Usuários que já tenham um ambiente Python funcional em seus sistemas podem optar pela ferramenta `pip` para instalar e gerenciar os módulos de computação científica:

```
$ pip install ipython scipy matplotlib
```

2 A Linguagem Python

Esta seção apresenta uma **introdução sucinta** à linguagem Python. Pesquisadores com experiência prévia em programação procedural como Fortran, C ou MATLAB não deverão ter maiores dificuldades com o conteúdo a seguir, uma vez que ele define em Python conceitos familiares como atribuição de variáveis, controle de fluxo e iterações. Pesquisadores que não possuem conhecimentos prévios sobre programação de computadores podem buscar orientação no projeto Software Carpentry[8], destinado a ensinar diversas técnicas de computação a pesquisadores com formações acadêmicas variadas.

2.1 Tipos numéricos

Há quatro tipos numéricos em Python: inteiros simples (*plain integers* - ou apenas inteiros), inteiros longos (*long integers*), números em ponto flutuante (*floating points*) e complexos.

2.1.1 Inteiros

A diferença entre o inteiro simples e o inteiro longo é que o último possui *precisão infinita*, permitindo a representação de qualquer número em $\mathbb{Z}$ (na

[8] Software Carpentry. Disponível: < http://software-carpentry.org>.

prática, há limites impostos pela memória do computador). Na maior parte do tempo, o programador utiliza inteiros simples, capazes de representar números inteiros no intervalo `[-n - 1,n]` . O valor de n varia de acordo com o sistema (32 ou 64 bits), podendo ser consultado na variável sys.maxint:

```
In [1]:sys.maxint
Out[1]:9223372036854775807
```

Inteiros são representados e manipulados trivialmente:

```
In [2]:1 + 1
Out[2]:2
```

Como usual na maioria das linguagens procedurais, uma **atribuição de variável** é realizada utilizando-se o operador `=`. Podemos conferir o **tipo** da variável utilizando a função `type`:

```
In [3]:a = 4
       type(a)
Out[3]:int
```

O resto da divisão inteira pode ser obtido utilizando o operador %:

```
In [4]:11 % 4
Out[4]:3
```

Inteiros longos (precisão infinita) são representados acionando-se a letra `l` (minúscula ou maiúscula), ao final do inteiro:

```
In [5]:a = 32L
       a
Out[5]:32L
In [6]:b = a * sys.maxint
       b
Out[6]:295147905179352825824L
In [7]:c = b * sys.maxint
       c
Out[7]:2722258935367507707116701049095440039968L
```



```
In [8]:type(c)
Out[8]:long
```

2.1.2 Reais

Diferenciam-se os números reais dos inteiros com o uso do caractere “`.`”, opcionalmente anexando-se casas decimais se houver:

```
In [9]:a = 2
       type(a)
Out[9]:int
In [10]: a = 2.
         type(a)
Out[10]: float
In [11]: a = 2.1
         type(a)
Out[11]: float
```

Informações sobre a representação dos números em ponto flutuante no sistema, como valores máximos e mínimos, podem sem obtidos com a variável `sys.float_info`:

```
In [12]: sys.float_info
Out[12]:
sys.float info(max=1.7976931348623157e+308, max exp=1024,
max 10 exp=308, min=2.2250738585072014e-308,
min exp=-1021, min 10 exp=-307, dig=15, mant dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

2.1.3 Complexos

Números complexos têm sua parte imaginária definida pelo caractere “`j`”. Os atributos `real` e `imag` permitem acesso direto às partes real e imaginária do número.

```
In [13]: a = 1.2 + 0.7j
         a
Out[13]: (1.2+0.7j)
In [14]: type(a)
```

```
Out[14]: complex
In [15]: a.real
Out[15]: 1.2
In [16]: a.imag
Out[16]: 0.7
```

2.2 Operações matemáticas

Além das operações aritméticas de adição, subtração, multiplicação e divisão, Python fornece uma maneira simples de definir **potenciação**, utilizando a notação "`**`":

```
In [17]: 2**3
Out[17]: 8
```

Quanto à divisão, vale lembrar a diferença entre dividir números inteiros e reais:

```
In [18]: 3/2
Out[18]: 1
In [19]: 3./2
Out[19]: 1.5
```

2.3 *Containers*

Todas as variáveis em Python, até mesmo as variáveis numéricas básicas, são objetos. Uma categoria especial de objetos em Python são os *containers*, capazes de armazenar outros objetos, como listas, tuplas e dicionários.

2.3.1 Listas

Listas são as sequências mais comuns e mais utilizadas em Python. Listas podem ser criadas diretamente utilizando-se colchetes "`[]`", como no exemplo abaixo contendo oito objetos *string*:


```
In [20]: L = ['vermelho', 'azul', 'verde', 'amarelo', 'ciano',
'magenta', 'branco', 'preto']
         type(L)
Out[20]: list
In [21]: L
Out[21]: ['vermelho', 'azul', 'verde', 'amarelo', 'ciano',
'magenta', 'branco', 'preto']
```

Elementos da lista podem ser acessados por seu **índice**, lembrando que, de modo similar a Fortran e `C` mas diferentemente de MATLAB, o primeiro elemento é indexado por `0`:

```
In [22]: L[0]
Out[22]: 'vermelho'
In [23]: L[2]
Out[23]: 'verde'
```

Um recurso útil na indexação em Python é que índices negativos podem ser utilizados para acessar a lista em **ordem reversa**. Por exemplo, `-1` pode ser utilizado para acessar o último elemento da lista:

```
In [24]: L[-1]
Out[24]: 'preto'
In [25]: L[-2]
Out[25]: 'branco'
```

Outro recurso útil é chamado de *slicing*. Utilizando a notação `início:fim:passo`, podemos obter listas que são subsequências da lista inicial:

```
In [26]: L[2:6]
Out[26]: ['verde', 'amarelo', 'ciano', 'magenta']
In [27]: L[2:]
Out[27]: ['verde', 'amarelo', 'ciano', 'magenta', 'branco', 'preto']
In [28]: L[1:6:2]
Out[28]: ['azul', 'amarelo', 'magenta']
```

Listas podem conter tipos diferentes de objetos:

```
In [29]: L = [3, -200.7, 3+5j, 'hello']
         L
Out[29]: [3, -200.7, (3+5j), 'hello']
```

O método `pop` pode ser utilizado para obter e simultaneamente retirar um objeto da lista. Seu comportamento padrão (sem argumentos) é remover o último elemento da lista:

```
In [30]: x = L.pop()
         x
Out[30]: 'hello'
In [31]: L
Out[31]: [3, -200.7, (3+5j)]
```

A posição do elemento-alvo pode ser informada:

```
In [32]: x = L.pop(0)
         x
Out[32]: 3
In [33]: L
Out[33]: [-200.7, (3+5j)]
```

Os metodos `append` e `extend` podem ser utilizados para adicionar elementos a lista. O primeiro adiciona um unico objeto como elemento da lista. Já `extend` adiciona a lista cada um dos elementos de uma segunda lista passada como argumento:

```
In [34]: L
Out[34]: [-200.7, (3+5j)]
In [35]: L.append(42)
         L
Out[35]: [-200.7, (3+5j), 42]
In [36]: L.extend([3, 5, 'ueba!'])
         L
Out[36]: [-200.7, (3+5j), 42, 3, 5, 'ueba!']
```

Se uma lista for utilizada como argumento da função `append`, ela será inserida como um único elemento:


```
In [38]: L.append([1,2,3])
         L
Out[38]: [-200.7, (3+5j), 42, 3, 5, 'ueba!', [1, 2, 3]]
```

2.3.2 Tuplas

Tuplas são **listas imutáveis**. Elementos podem ser lidos a partir das tuplas e até mesmo *slicing* pode ser utilizado para gerar novas tuplas. Porém, não é possível alterar uma tupla de forma alguma após sua criação.

```
In [37]: ponto = (23, 542)
In [38]: ponto
Out[38]: (23, 542)
In [39]: ponto[0]
Out[39]: 23
In [40]: x, y = ponto
         print x
         print y
23
542
```

2.3.3 Dicionários

Dicionários são tabelas para armazenamento de pares *chave*, *valor*. Eles são implementados através de eficientes **tabelas de espalhamento** (*hash tables*).

```
In [41]: tel = {'emmanuelle': 5752, 'sebastian': 5578}
In [42]: tel
Out[42]: {'emmanuelle': 5752, 'sebastian': 5578}
In [43]: tel.keys()
Out[43]: ['sebastian', 'emmanuelle']
In [44]: tel.values()
Out[44]: [5578, 5752]
In [45]: tel['sebastian']
Out[45]: 5578
```

Um nova entrada pode ser adicionada ao dicionário simplesmente atribuindo-se um valor a uma nova chave:

```
In [46]: tel['thiago'] = 5823
         tel
Out[46]: {'emmanuelle': 5752, 'sebastian': 5578, 'thiago': 5823}
```

Iteração em dicionários:

```
In [47]: for k in tel.keys():
             print 'Fale com %s no ramal %d' % (k, tel[k])
Fale com sebastian no ramal 5578
Fale com thiago no ramal 5823
Fale com emmanuelle no ramal 5752
```

2.4 Controle de fluxo

2.4.1 if/elif/else

```
In [48]: if 2+2 == 4:
             print 'OK, o Universo está bem'
OK, o Universo está bem
In [49]: a = 10
         if a == 1:
           print 'É pouco'
         elif a == 2:
           print 'É bom'
         else:
           print 'É demais!'
É demais!
```

2.4.2 for/range

```
In [50]: for i in range(5):
             print i
0
1
2
3
4
```



```
In [51]: for word in ('legal', 'poderoso', 'legível'):
print 'Python é %s' % word
Python é legal
Python é poderoso
Python é legível
In [52]: for i, word in enumerate(('legal', 'poderoso', 'legível')):
             print '%d°: Python é %s' % (i, word)
0°: Python é legal
1°: Python é poderoso
2°: Python é legível
```

2.4.3 while/break

```
In [53]: z = 1 + 1j
         while abs(z) < 100:
           if z.imag == 0:
             break
           z = z**2 + 1
         z
Out[53]: (-134+352j)
```

2.5 Compreensão de listas

Compreensão de listas é um dos recursos mais interessantes da linguagem Python. Esse recurso possibilita a criação dinâmica de novas listas a partir de objetos iteráveis, sendo mais eficiente que o uso de um laço for e produzindo código mais legível. Considere o exemplo abaixo:

```
In [54]: sqr = [i**2 for i in range(10)]
         sqr
Out[54]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A semântica do código acima é simples: `sqr` é uma lista contendo i^2 para cada i na sequência `range(10)`, que no caso é o intervalo `[0,9]`. Considere outro exemplo:

```
In [55]: doc = ['Hello', 'Embrapa', 'Informática']
         wlen = [len(word) for word in doc]
         wlen
```

```
Out[55]: [5, 7, 12]
```

Aqui, `wlen` é a lista contendo o comprimento de cada palavra na sequência `doc`. Usada corretamente, a compreensão de listas pode produzir códigos elegantes de fácil compreensão por parte de outros pesquisadores. Note como o primeiro exemplo corresponde diretamente a expressão matemática $i^2, \forall i \in [0,9]$.

3 IPython

IPython é um ambiente de **computação interativa** que enriquece os recursos originais do console Python padrão. Na computação interativa, fragmentos de código podem ser inseridos pelo pesquisador, executados imediatamente e seus resultados se tornam prontamente disponíveis. O sistema mantém seu estado em memória, isto é, seu conjunto de variáveis em sua configuração atual é acessível ao usuário. Perez e Granger (2007) argumentam sobre as vantagens de um ambiente interativo ao pesquisador:

> "Este estilo flexível casa bem com o espírito da computação em um contexto científico, no qual a determinação de quais computações devem ser realizadas em seguida geralmente requer um esforço significativo [por parte do pesquisador]. Um ambiente interativo permite aos cientistas examinar os dados, testar novas ideias, combinar algoritmos e avaliar os resultados diretamente."

IPython provê ainda recursos para paralelizar a execução do código, com pouco esforço adicional, permitindo ao pesquisador tirar proveito de processadores multi-core e *clusters* de computação[9].

3.1 Console básico

O console básico pode ser iniciado executando-se o comando `ipython`:

[9] O leitor interessado pode visitar *Using IPython for parallel computing* http://ipython.org/ipython-doc/stable/parallel/index.html para maiores detalhes sobre computação paralela com IPython.


```
$ ipython
Python 2.7.5+ (default, Feb 27 2014, 19:37:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra
details.

In [1]:
```

3.2 Qtconsole

Qtconsole é um terminal IPython que utiliza uma interface gráfica baseada na biblioteca `Qt`. Ele pode ser inicializado através do comando:

```
$ ipython qtconsole
```

3.3 A opção `pylab`

A opção `pylab` faz com que o interpretador IPython importe para o ambiente os pacotes essenciais para *arrays* da `NumPy` e as rotinas de *plotting* da Matplotlib. Alternativamente, pode-se executar o comando "mágico" (*magic command*) `%pylab` dentro de um console IPython em execução.

```
$ ipython qtconsole --pylab
```

3.4 Um "caderno de laboratório" executável: IPython Notebook

Um Notebook IPython é uma versão baseada em arquitetura web do console IPython que permite a mesma computação interativa dos consoles, mas oferece funcionalidade adicional. Utilizando um navegador web, o pesquisador pode organizar trechos de anotações e código de maneira flexível. Texto e código são organizados em **células** que podem ser inseridas, apagadas, reorganizadas e executadas conforme necessário. Um

Notebook IPython pode apresentar gráficos, fórmulas matemáticas e saída de código, tudo organizado em um único documento. Notebooks vêm sendo utilizados em anotações de pesquisa, redação de artigos científicos, análises de dados e na produção de livros.

IPython pode ser iniciado no modo notebook através do comando:

```
$ ipython notebook --pylab=inline
```

Tal comando iniciará um servidor web e, em seguida, levará o navegador web padrão do sistema até um painel de controle que permite criação e gerenciamento de notebooks.

3.4.1 Células

Notebooks são compostos por células de dois tipos: células de texto e células de código. Células de código contém trechos de código Python a ser executado. Já as células de texto podem conter anotações formatadas utilizando as convenções *Markdown*[10].

3.4.2 Expressões matemáticas

Células de texto suportam também **expressões matemáticas**, utilizando a sintaxe do LaTeX. Por exemplo, a expressão:

```
y = 3 x^2 + \epsilon
```

produz a fórmula $y = 3x^2 + \epsilon$. A seguir estão alguns exemplos de fórmulas matemáticas produzidas através da notação LaTeX[11].

$\lim_{x \to \infty} \exp(-x) = 0$	`$\lim_{x \to \infty} \exp(-x) = 0$`
$\sqrt[n]{1+x+x^2+x^3+\ldots}$	`$\sqrt[n]{1+x+x^2+x^3+\ldots}$`
$\int_0^\infty \mathrm{e}^{-x}\mathrm{d}x$	`$\int_0^\infty \mathrm{e}^{-x} \mathrm{d}x$`
$\sum_{i=1}^{10} t_i$	`$\sum_{i=1}^{10} t_i$`

[10] Markdown. Disponível em: <http://daringfireball.net/projects/markdown/syntax>.

[11] Para mais detalhes sobre como escrever fórmulas matemáticas na sintaxe do LaTeX, o leitor pode considerar <http://en.wikibooks.org/wiki/LaTeX/Mathematics>.

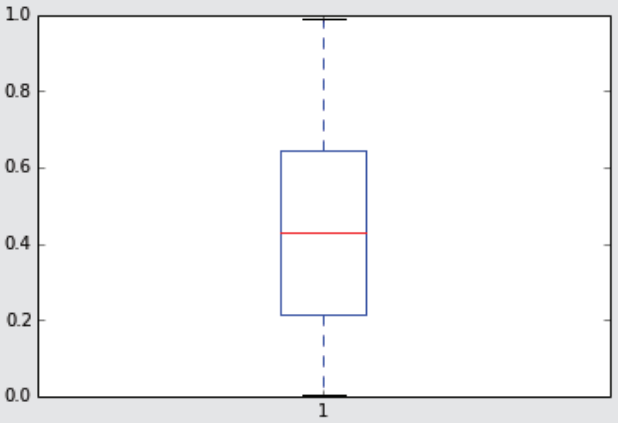
3.4.3 Como notebooks são armazenados

Notebooks são armazenados como arquivos JSON que guardam o conteúdo das células, incluindo os resultados das computações realizadas. Tais arquivos possuem a extensão `.ipynb` e podem ser convertidos em programas Python regulares e em documentos HTML ou PDF. O que faz dos notebooks uma excelente ferramenta para pesquisa reprodutível é que eles são **documentos executáveis** que armazenam não só descrições textuais e matemáticas de um procedimento, mas também o código necessário para replicar as computações.

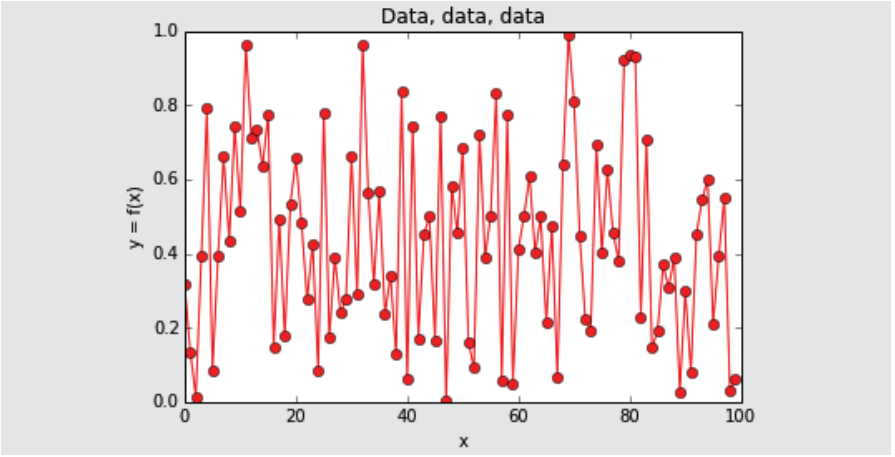
3.5 Visualização científica

IPython integra-se facilmente à Matplotlib, permitindo a exibição de gráficos diretamente no navegador web ou no console `Qt` (a criação de gráficos será introduzida posteriormente, na Seção 5):

```
In [2]:data = random.rand(100)
       bp = boxplot(data)
```




```
In [3]:plot(data, 'r-')
       plot(data, 'ro')
       title('Data, data, data')
       xlabel('x')
       ylabel('y = f(x)')
Out[3]:<matplotlib.text.Text at 0x7f91d0716150>
```



3.6 IPython como um console de sistema

Podemos executar comandos do sistema diretamente do console IPython. Por exemplo, para listar arquivos no diretório corrente, podemos utilizar o comando de sistema ls (em sistema baseados em Unix como Linux e Mac OS):

```
In [4]: ls
01._Introducao.ipynb                04._NumPy.ipynb            data/
01._Introducao.pdf                  04._NumPy.pdf              figs/
01._Introducao.slides.html          04._NumPy.slides.html      foo.ipynb~
01._Introducao.tex                  04._NumPy.tex              LICENSE
02._A_Linguagem_Python.ipynb        05._Matplotlib_files/      README.md
02._A_Linguagem_Python.pdf          05._Matplotlib.ipynb       reveal.js/
02._A_Linguagem_Python.slides.html  05._Matplotlib.pdf         scipy_embrapa.aux
02._A_Linguagem_Python.tex          05._Matplotlib.slides.html scipy_embrapa.log
03._IPython_files/                  05._Matplotlib.tex         scipy_embrapa.out
03._IPython.ipynb                   06._Scipy_files/           scipy_embrapa.pdf
03._IPython.pdf                     06._Scipy.ipynb            scipy_embrapa.tex
03._IPython.slides.html             06._Scipy.pdf              scipy_embrapa.
tex~
03._IPython.tex                     06._Scipy.slides.html      trig.pdf
04._NumPy_files/                    06._Scipy.tex              trig.tif
```

Utilizando `!` antes do comando, podemos armazenar o resultado em uma variável Python:


```
In [5]: l = !ls figs
In [6]: l
Out[6]: ['fig.png', 'fitting.png', 'john-hunter.jpg', 'python.png']
In [7]: figs = [f for f in l if f.endswith('.png')]
In [8]: figs
Out[8]: ['fig.png', 'fitting.png', 'python.png']
```

Comandos do sistema podem ser executados utilizando parâmetros armazenados em variáveis Python. O exemplo abaixo copia arquivos entre diretórios utilizando o comando de sistema `cp`. Os arquivos de interesse são lidos da lista `figs` e passados como argumento para o comando:

```
In [9]: for f in figs:
            !cp figs/$f /tmp/
In [10]: ls /tmp/*.png
/tmp/fig.png  /tmp/fitting.png  /tmp/python.png
```

4 NumPy

Previamente na Seção 2, vimos que listas podem armazenar qualquer tipo de dado. Tal flexibilidade acarreta perdas em desempenho, uma vez que, internamente, o interpretador Python precisa lidar com questões de posicionamento em memória e tipagem de dados. Em computação científica, a maior parte dos dados consiste em valores numéricos, geralmente de mesmo tipo (inteiro ou real). Ganhos de desempenho consideráveis podem ser obtidos se houver uniformidade no tipo de dado que está sendo considerado.

Um **array NumPy** é um coleção multidimensional e uniforme de elementos, isto é, todos os elementos ocupam o mesmo número de bytes em memória (WALT et al., 2011). Ele é a representação padrão para *arrays* n-dimensionais na SciPy Stack e é comumente utilizada na representação de vetores, matrizes, observações, sinais e imagens (*arrays* de até 32 dimensões podem ser representados).

NumPy pode ser importado no ambiente com o comando `import numpy`, como no exemplo abaixo:

```
In [1]:import numpy as np
       A = np.array([0,1,2,3])
       A
Out[1]:array([0, 1, 2, 3])
In [2]:A.shape = 2,2
       A
Out[2]:array([[0, 1],
              [2, 3]])
```

Se o interpretador IPython foi iniciado com a opção `pylab`, o módulo NumPy já estará carregado e não haverá necessidade, como no exemplo acima, do prefixo *np*.

NumPy evita iterações custosas do interpretador Python quando opera com *arrays*, utilizando, no lugar, eficientes operações **vetorizadas** implementadas em código de máquina. Como um exemplo simples, suponha que tenhamos um vetor com 10000 valores e desejemos computar o quadrado de cada valor. Utilizando uma lista, o interpretador Python iria usar iterações para realizar a computação para cada membro da lista:

```
In [3]:v = range(10000)
       %timeit [i**2 for i in v]
1000 loops, best of 3: 567 µs per loop
```

A função `range` produz uma **lista** com 10000 valores, de 0 a 9999. A computação levou em média, para a máquina em questão, 567 microssegundos. Abaixo, utilizamos a função `arange` do módulo NumPy, que devolve um *array* com os mesmos 10000 valores:

```
In [4]:v = arange(10000)
       %timeit v**2
100000 loops, best of 3: 8.95 µs per loop
```

Utilizando código nativo e vetorização, NumPy levou apenas 8,95 microssegundos para realizar toda a computação. Nos exemplos acima, a função "mágica" do IPython `%timeit` foi utilizada para calcular o desempenho do código. A operação `v**2` computa um novo array com todos os elementos de v ao quadrado (mais detalhes na Seção 4.6).

4.1 Criação de *arrays*

Arrays podem ser facilmente criados com a função *array*, que recebe como argumento uma lista contendo os dados que deverão ser armazenados. O número de dimensões de um array pode ser obtido pelo atributo `ndim` enquanto que as dimensões em si ficam armazenadas no atributo `shape`:

```
In [5]:v = array([0,1,2,3])
       v
Out[5]:array([0, 1, 2, 3])
In [6]:v.ndim
Out[6]:1
In [7]:v.shape
Out[7]:(4,)
```

A lista com os dados pode conter uma estrutura que permita ao método array inferir as dimensões pretendidas. Por exemplo, uma matriz 2D pode ser inicializada a partir de uma **lista de listas**:

```
In [8]:A = array([ [0,1,2], [3,4,5] ])
       A
Out[8]:array([[0, 1, 2],
              [3, 4, 5]])
In [9]: A.ndim
Out[9]:2
In [10]: A.shape
Out[10]: (2, 3)
```

Se for mais conveniente, é possível informar apenas uma lista com todos os elementos encadeados e as dimensões do array podem ser redefinidas posteriormente:

```
In [11]: data = range(12)
         data
Out[11]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
In [12]: B = array(data)
         B
Out[12]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
In [13]: B.ndim
```

```
Out[13]: 1
In [14]: B.shape
Out[14]: (12,)
In [15]: B[6]
Out[15]: 6
In [16]: B.shape = 3,4
In [17]: B.ndim
Out[17]: 2
In [18]: B.shape
Out[18]: (3, 4)
In [19]: B
Out[19]: array([[0, 1, 2, 3],
                [4, 5, 6, 7],
                [8, 9, 10, 11]])
In [20]: # Linha 1, coluna 2
         B[1,2]
Out[20]: 6
```

4.1.1 Utilitários para criação de *arrays*

Para criar um *array* contendo uma sequência com *n* elementos de `0` a $n-1$, podemos utilizar a função arange:

```
In [21]: v = arange(5)
         v
Out[21]: array([0, 1, 2, 3, 4])
```

A função `arange` também permite definir os elementos inicial e final e um passo entre elementos sucessivos. Por exemplo, para produzir uma sequência de 1 (inclusivo) até 8 (exclusivo) na qual os números inteiros são tomados a cada 2 elementos, pode-se utilizar:

```
In [22]: v = arange(1,8,2)
         v
Out[22]: array([1, 3, 5, 7])
```

A função `arange` trabalha com números em $\mathbb{Z}$. Para uma sequência de números reais, igualmente espaçada dentro de um intervalo, pode-se

utilizar `linspace`. Por exemplo, para obter 6 números reais igualmente espaçados no intervalo `[0,1]`, utiliza-se:

```
In [23]: v = linspace(0, 1, 6)
         v
Out[23]: array([0., 0.2, 0.4, 0.6, 0.8, 1.])
```

`zeros`, `ones`, `identity`. Para produzir matrizes de dimensões variadas com um valor fixo, podemos utilizar as funções `zeros` e `ones`, informando as dimensões desejadas na forma de uma tupla como argumento:

```
In [24]: A = np.zeros((4,5))
         A
Out[24]: array([[0.,  0.,  0.,  0.,  0.],
                [0.,  0.,  0.,  0.,  0.],
                [0.,  0.,  0.,  0.,  0.],
                [0.,  0.,  0.,  0.,  0.]])
In [25]: B = np.ones((3,3))
         B
Out[25]: array([[1.,  1.,  1.],
                [1.,  1.,  1.],
                [1.,  1.,  1.]])
```

Matrizes identidade são particularmente importantes em computação envolvendo álgebra linear. A função `identity` produz uma matriz quadrada do tamanho desejado, com todos os elementos em sua diagonal principal apresentando o valor 1, zero para todos os demais:

```
In [26]: I = identity(3)
         I
Out[26]: array([[1.,  0.,  0.],
                [0.,  1.,  0.],
                [0.,  0.,  1.]])
```

`random` Certas aplicações podem necessitar de valores aleatórios, por exemplo em testes com ruído sintético adicionado ao sinal. *Arrays* aleatórios podem ser produzidos utilizando-se o submódulo `random` da NumPy. Valores obtidos de uma distribuição uniforme no intervalo `[0,1]` são produzidos com `rand`:

```
In [27]: A = random.rand(4)
         A
Out[27]: array([0.31827175,  0.73058309,  0.86259259,  0.26156631])
```

Outra opção é utilizar uma distribuição Gaussiana através da função `randn`:

```
In [28]: A = np.random.randn(4)
         A
Out[28]: array([ 1.14767124,  0.65924363,  0.82154375,  0.15309107])
```

4.2 Tipos de dados

Diferente de listas, que podem armazenar elementos de tipos diferentes, *arrays* devem conter elementos de **mesmo tipo**, como especificado em `dtype`.

```
In [29]: v = np.array([1,2,3])
         v.dtype
Out[29]: dtype('int64')
```

Acima, vemos que o tipo padrão para inteiros adotado pelo sistema em uso no exemplo utiliza 64 bits. Similarmente, vemos que o tipo padrão para ponto flutuante também se baseia em 64 bits:

```
In [30]: v = np.array([1., 2., 3.])
         v.dtype
Out[30]: dtype('float64')
```

Alternativamente, podemos especificar exatamente o tipo desejado, dentre os disponíveis na NumPy. O exemplo abaixo cria um array com elementos em ponto flutuante com 32 bits, apesar do argumento de entrada consistir em uma lista de inteiros:

```
In [31]: v = np.array([1,2,3], dtype=float32)
         v
Out[31]: array([ 1.,  2.,  3.], dtype=float32)
In [32]: v.dtype
```



```
Out[32]: dtype('float32')
```

Utilitários para criação de *arrays* também possibilitam especificar o tipo pretendido:

```
In [33]: I = np.identity(3, dtype=np.uint8)
         I
Out[33]: array([[1, 0, 0],
                [0, 1, 0],
                [0, 0, 1]], dtype = uint8)
```

4.3 Carga de dados armazenados em arquivos

Na maior parte de seu trabalho, o pesquisador não irá inserir manualmente o conteúdo dos *arrays*, nem terá todas suas necessidades atendidas pelos utilitários de criação. São necessários mecanismos para a carga de dados armazenados em arquivo. Considere por exemplo o conteúdo do arquivo `populations.txt`:

```
In [34]: !cat ./data/populations.txt
# year      hare        lynx        carrot
1900        30e3        4e3         48300
1901        47.2e3      6.1e3       48200
1902        70.2e3      9.8e3       41500
1903        77.4e3      35.2e3      38200
1904        36.3e3      59.4e3      40600
1905        20.6e3      41.7e3      39800
1906        18.1e3      19e3        38600
1907        21.4e3      13e3        42300
1908        22e3        8.3e3       44500
1909        25.4e3      9.1e3       42100
1910        27.1e3      7.4e3       46000
1911        40.3e3      8e3         46800
1912        57e3        12.3e3      43800
1913        76.6e3      19.5e3      40900
1914        52.3e3      45.7e3      39400
1915        19.5e3      51.1e3      39000
1916        11.2e3      29.7e3      36700
1917        7.6e3       15.8e3      41800
1918        14.6e3      9.7e3       43300
```

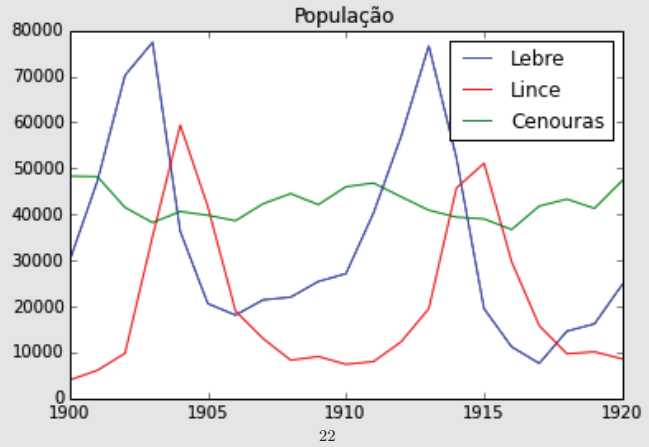
```
1919        16.2e3      10.1e3      41300
1920        24.7e3      8.6e3       47300
```

Esta tabela pode ser facilmente carregada pelo ambiente como um array NumPy com o uso da função `loadtxt`:

```
In [35]: populations = loadtxt('./data/populations.txt')
         populations
Out[35]: array([[ 1900.,  30000.,   4000.,  48300.],
                [ 1901.,  47200.,   6100.,  48200.],
                [ 1902.,  70200.,   9800.,  41500.],
                [ 1903.,  77400.,  35200.,  38200.],
                [ 1904.,  36300.,  59400.,  40600.],
                [ 1905.,  20600.,  41700.,  39800.],
                [ 1906.,  18100.,  19000.,  38600.],
                [ 1907.,  21400.,  13000.,  42300.],
                [ 1908.,  22000.,   8300.,  44500.],
                [ 1909.,  25400.,   9100.,  42100.],
                [ 1910.,  27100.,   7400.,  46000.],
                [ 1911.,  40300.,   8000.,  46800.],
                [ 1912.,  57000.,  12300.,  43800.],
                [ 1913.,  76600.,  19500.,  40900.],
                [ 1914.,  52300.,  45700.,  39400.],
                [ 1915.,  19500.,  51100.,  39000.],
                [ 1916.,  11200.,  29700.,  36700.],
                [ 1917.,   7600.,  15800.,  41800.],
                [ 1918.,  14600.,   9700.,  43300.],
                [ 1919.,  16200.,  10100.,  41300.],
                [ 1920.,  24700.,   8600.,  47300.]])
```

Como ilustração, abaixo segue uma representação gráfica do desenvolvimento das populações de cenouras, lebres e linces ao longo do tempo, contida no array `populations` (a produção de gráficos será apresentada na Seção 5):

```
In [36]: year = populations[:,0]
         hare = populations[:,1]
         lynx = populations[:,2]
         carrot = populations[:,3]
         plot(year, hare, 'b-')
         plot(year, lynx, 'r-')
         plot(year, carrot, 'g-')
         title(u'População')
         legend(('Lebre', 'Lince', 'Cenouras'))
Out[36]: <matplotlib.legend.Legend at 0x7f9ffe73b310>
```



De modo similar, `savetxt` pode ser utilizado para salvar um array em um **arquivo texto**. NumPy também possui uma funções save e load que respectivamente salvam e carregam *arrays* em um **formato binário** próprio.

4.4 Sub-arrays

Considere a matriz A abaixo (representada como um array NumPy):

```
In [38]: A
Out[38]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])
```

A matriz é bidimensional. Seus elementos podem ser indexados através de duas coordenadas, separadas por uma vírgula:

```
In [39]: A[2,3]
Out[39]: 23
```

Fragmentos do array podem ser obtidos pelo mesmo tipo de *slicing* visto anteriormente para listas Python. Suponha que desejemos obter apenas as colunas 3 e 4 na linha 4. Podemos utilizar *slicing* nas colunas de A:

```
In [40]: A[4,3:5]
Out[40]: array([43, 44])
```

Suponha agora que desejemos todas as linhas a partir da linha 4 e todas as colunas a partir da coluna 3:

```
In [41]: A[4:,3:]
Out[41]: array([[43, 44, 45],
                [53, 54, 55]])
```

Outros exemplos:

```
In [42]: A[:,2]
Out[42]: array([ 2, 12, 22, 32, 42, 52])
In [43]: A[2::2,::2]
Out[43]: array([[20, 22, 24],
                [40, 42, 44]])
```

Essas operações de *slicing* criam um **visão** (*view*) do *array* original, não uma cópia. Quando a visão é modificada, o array original é modificado também:

```
In [44]: B = A[0,3:5]
         B[0] = 99
         print B
         print A
[99  4]
[ [ 0  1  2 99  4  5]
  [10 11 12 13 14 15]
  [20 21 22 23 24 25]
  [30 31 32 33 34 35]
  [40 41 42 43 44 45]
  [50 51 52 53 54 55]]
```


Para evitar sobre-escrita, se necessário, podemos criar uma **cópia do array**. No exemplo abaixo, as matrizes A e B são representadas por *arrays* que não compartilham memória, não havendo sobre-escrita:

```
In [45]: A = array(data)
         A.shape = 6,6
         # Cópia
         B = A[0,3:5].copy()
         B[0] = 99
         print B
         print A
[99  4]
[ [ 0  1  2  3  4  5]
  [10 11 12 13 14 15]
  [20 21 22 23 24 25]
  [30 31 32 33 34 35]
  [40 41 42 43 44 45]
  [50 51 52 53 54 55]]
```

4.5 *Fancy indexing*

Podemos utilizar máscaras para selecionar os elementos de um array:

```
In [46]: A = array(data)
         A.shape = 6,6
         A
Out[46]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])
```

No exemplo abaixo, `mask` é um **array booleano** no qual cada elemento recebe o valor “Verdadeiro” (*True*) se e somente se seu elemento equivalente no *array* A for divisível por 3:

```
In [47]: mask = A % 3 == 0
         mask
Out[47]: array([[ True, False, False, True, False, False],
```

```
                [False, False, True, False, False, True],
                [False, True, False, False, True, False],
                [ True, False, False, True, False, False],
                [False, False, True, False, False, True],
                [False, True, False, False, True, False]],
dtype=bool)
```

O array booleano `mask` pode ser utilizado para selecionar elementos de A (no caso, apenas os elementos múltiplos de 3):

```
In [48]: A[mask]
Out[48]: array([ 0,  3, 12, 15, 21, 24, 30, 33, 42, 45, 51, 54])
```

Podemos também aplicar a máscara Booleana diretamente:

```
In [49]: B = A[A % 3 == 0]
         B # importante:- B é uma cópia
Out[49]: array([ 0,  3, 12, 15, 21, 24, 30, 33, 42, 45, 51, 54])
```

Fancy indexing também pode ser realizado a partir de um **array ou uma sequência de índices (na forma de listas ou tuplas)**. No exemplo abaixo, utilizamos a lista `[1,5,6,8]` como índice, recuperando assim os elementos nas posições 1, 5, 6 e 8 no array v :

```
In [50]: v = arange(0,100,10)
         v
Out[50]: array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
In [51]: v[[1,5,6,8]]
Out[51]: array([10, 50, 60, 80])
```

No exemplo abaixo, duas sequências são fornecidas, na forma de tuplas. A primeira indexa as linhas da matriz A, enquanto a segunta indexa as colunas:


```
In [52]: A
Out[52]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])
In [53]: A[(0,1,2,3,4),(1,2,3,4,5)]
Out[53]: array([ 1, 12, 23, 34, 45])
```

4.6 Operações com arrays

Escalar e array:

```
In [54]: v = arange(10)
         v
Out[54]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [55]: v + 1
Out[55]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
In [56]: 2**v
Out[56]: array([  1,   2,   4,   8,  16,  32,  64, 128, 256, 512])
```

Entre arrays, elemento por elemento:

```
In [57]: A = ones((3,3))
         B = 2 * ones((3,3))
         print A
         print B
[ [ 1.  1.  1.]
  [ 1.  1.  1.]
  [ 1.  1.  1.]]
[ [ 2.  2.  2.]
  [ 2.  2.  2.]
  [ 2.  2.  2.]]
In [58]: A + B
Out[58]: array([[ 3.,  3.,  3.],
                [ 3.,  3.,  3.],
                [ 3.,  3.,  3.]])
In [59]:A * B
```

```
Out[59]:array([ [ 2.,  2.,  2.],
                [ 2.,  2.,  2.],
                [ 2.,  2.,  2.]])
```

Importante: multiplicação de *arrays* não equivale à multiplicação de matrizes. A multiplicação de matrizes, como definida em álgebra, é obtida com a função `dot`:

```
In [60]: dot(A, B)
Out[60]: array([[ 6.,  6.,  6.],
                [ 6.,  6.,  6.],
                [ 6.,  6.,  6.]])
```

Transposição:

```
In [61]: A = arange(12).reshape(3,4)
         A
Out[61]: array([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]])
In [62]: A.T
Out[62]: array([[ 0,  4,  8],
                [ 1,  5,  9],
                [ 2,  6, 10],
                [ 3,  7, 11]])
```

Comparações:

```
In [63]: u = random.rand(5)
         u
Out[63]: array([ 0.63109122,  0.98298045,  0.5259016 ,  0.53315641,
0.91676396])
In [64]: v = random.rand(5)
         v
Out[64]: array([ 0.68171157,  0.57494436,  0.52487273,  0.25774938,
0.82496277])
In [65]: u > v
Out[65]: array([False,  True,  True,  True,  True], dtype=bool)
```

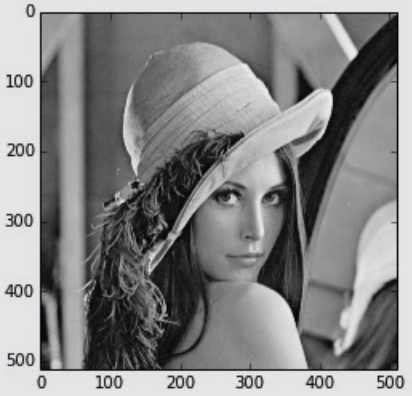

4.6.1 Álgebra linear

Operações de álgebra linear são implementadas em `numpy.linalg`. Porém, no geral, essas implementações **não são eficientes** e o módulo `scipy.linalg` deve ser preferido, por ser melhor otimizado. A Seção 6.1 apresenta mais informações sobre `scipy.linalg`.

4.7 Exemplo: imagens representadas como arrays

Imagens podem ser representadas como arrays nos quais cada elemento corresponde à intensidade luminosa naquela posição:

```
In [66]: from scipy.misc import lena
In [67]: I = lena()
         I
Out[67]: array([[162, 162, 162, …, 170, 155, 128],
                [162, 162, 162, …, 170, 155, 128],
                [162, 162, 162, …, 170, 155, 128],
                …,
                [ 43,  43,  50, …, 104, 100,  98],
                [ 44,  44,  55, …, 104, 105, 108],
                [ 44,  44,  55, …, 104, 105, 108]])
In [68]: imshow(I, cmap=cm.gray)
Out[68]: <matplotlib.image.AxesImage at 0x7f9ffd348090>
```



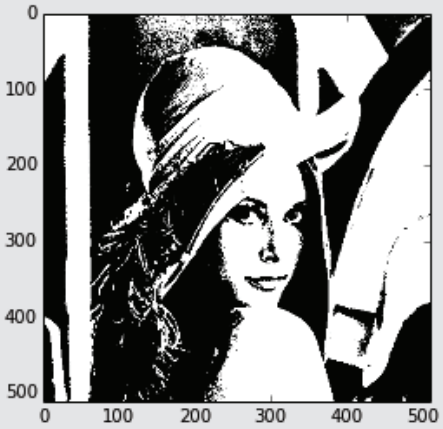
4.8 Exemplo: limiarização (*thresholding*)

Uma operação comum em processamento de imagens, chamada de limiarização, consiste em realizar o seguinte procedimento em cada pixel:

$$I'[x;y] = \begin{cases} 255 & \text{sse } I[x,y] > k, \\ 0 & \text{caso contrário} \end{cases}$$

o valor k é chamado de **limiar** (*threshold*). Tal operação pode ser facilmente implementada utilizando-se os recursos da NumPy.

```
In [70]: R = zeros_like(I) # Cria um array de zeros com as mesmas
dimensões de I
         R[I > 128] = 255
         imshow(R, cmap=cm.gray)
Out[70]: <matplotlib.image.AxesImage at 0x7f9ffd232b10>
```



4.9 Exemplo: cadeias de Markov e um modelo de clima muito simples

Considere um modelo bastante simplificado de predição climática. As probabilidades do dia ser *chuvoso* ou *ensolarado*, dado o clima do dia anterior, podem ser representadas a partir de uma **matriz de transição**:


```
In [71]: P = array([[0.9, 0.1],
                    [0.5, 0.5]])
```

A matriz `P` representa um modelo de clima no qual há uma chance de 90% de um dia ensolarado ser seguido por outro dia de sol e uma chance de 50% de um dia chuvoso se seguir a outro dia de chuva. As *colunas* podem ser rotuladas como *ensolarado* e *chuvoso* e as linhas na mesma ordem. Assim, `P[i,j]` é a probabilidade de um dia do tipo `i` ser seguido por um dia do tipo `j` (note que as linhas somam 1).

Considere que o tempo no primeiro dia foi ensolarado. Podemos representar este dia através de um **vetor de estado** x_0 no qual o campo *ensolarado* é 100% e o campo *chuvoso* é 0:

```
In [72]: x0 = array([1., 0])
```

A previsão do tempo para o dia seguinte pode ser dada por:

```
In [73]: x1 = x0.dot(P)
         x1
Out[73]: array([ 0.9,  0.1])
```

E no próximo:

```
In [74]: x2 = x1.dot(P)
         x2
Out[74]: array([ 0.86,  0.14])
```

E no seguinte:

```
In [75]: x3 = x2.dot(P)
         x3
Out[75]: array([ 0.844,  0.156])
```

5 Matplotlib

Matplotlib é um módulo para a criação de gráficos 2D e 3D criada por Hunter (2007). Sua sintaxe é propositalmente similar às funções de plotagem da MATLAB, facilitando o aprendizado de usuários que desejem replicar gráficos construídos naquele ambiente. Com uma grande comunidade de usuários, Matplotlib possui diversos tutoriais na web. Seu site oficial apresenta uma enorme galeria de exemplos que permite ao pesquisador rapidamente identificar o código necessário para o tipo de gráfico que pretende utilizar.

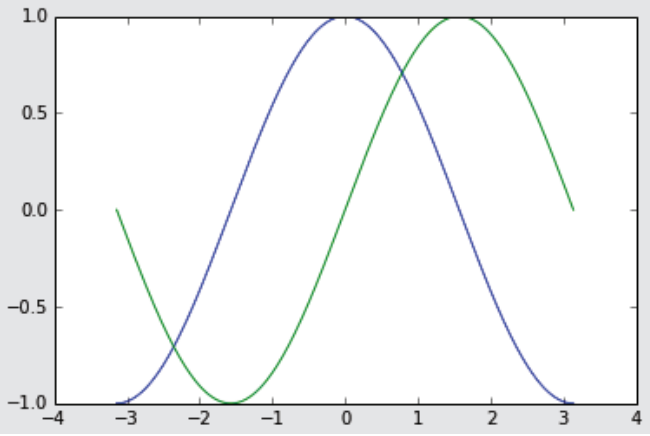
5.1 Exemplo: exibição de duas funções, sin(θ) e cos(θ)

Considere o domínio X, formado por 256 pontos no intervalo $[-\pi,\pi]$, e as funções $\cos(x)$ e $\sin(x)$:

```
In [1]:X = linspace(-pi, pi, 256)
       C = cos(X)
       S = sin(X)
```

O gráfico das duas funções pode ser facilmente exibido com a função `plot`:

```
In [2]:plot(X, C)
       plot(X, S)
Out[2]:[<matplotlib.lines.Line2D at 0x7feb255a0d10>]
```

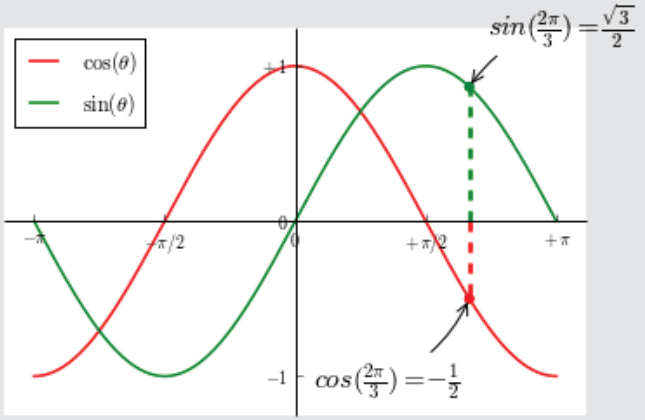


O exemplo acima apresenta um gráfico muito simples. Para ilustrar a grande variedade de personalizações fornecidas pela Matplotlib, é apresentado abaixo um exemplo mais complexo, uma versão modificada do código apresentado por Rougier, Müller e Varoquaux. O leitor interessado pode obter explicações detalhadas na Seção 1.4, *Matplotlib*: *plotting*, em Haenel et al. (2013).

```
In [3]:fig = figure()
       # Remover as bordas superior e inferior
       ax = gca()  # gca significa 'get current axis'
       ax.spines['right'].set_color('none')
       ax.spines['top'].set_color('none')
       # Mover os eixos e as marcas para o centro do gráfico
       ax.xaxis.set_ticks_position('bottom')
       ax.spines['bottom'].set_position(('data',0))
       ax.yaxis.set_ticks_position('left')
       ax.spines['left'].set_position(('data',0))
       # Define os intervalos exibidos nas ordenadas e abscissas
       xlim(-3.5, 3.5)
       ylim(-1.25, 1.25)
       # Indica o texto a ser utilizado nas marcas dos eixos
       xticks([-pi, -pi/2, 0, pi/2, pi], [r'$-\pi$', r'$-\pi/2$',
r'$0$', r'$+\pi/2$', r'$+\pi$'])
       yticks([-1, 0, +1], [r'$-1$', r'$0$', r'$+1$'])
       # Anotação de dois pontos de interesse: o seno e o cosseno de
2pi/3
       theta = 2 * pi / 3
       plot([theta, theta], [0, cos(theta)], color='red',
linewidth=2.5, linestyle="--")
       scatter([theta], [cos(theta)], 25, color='red')
       annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
                xy=(theta, sin(theta)), xycoords='data',
                xytext=(+10, +30), textcoords='offset points',
fontsize=16,
                arrowprops=dict(arrowstyle="->",
connectionstyle="arc3,rad=.2"))
       plot([theta, theta],[0, sin(theta)], color='green',
linewidth=2.5, linestyle="--")
       scatter([theta, ],[sin(theta), ], 25, color='green')
       annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$',
                xy=(theta, cos(theta)), xycoords='data',
```

```
                    xytext=(-90, -50), textcoords='offset points',
fontsize=16,
                    arrowprops=dict(arrowstyle="->",
connectionstyle="arc3,rad=.2"))
        # Exibe as funções
        plot(X, C, color="red", linewidth=1.5, linestyle="-",
label=r'$\cos(\theta)$')
        plot(X, S, color="green", linewidth=1.5, linestyle="-",
label=r'$\sin(\theta)$')
        # Adiciona a legenda
        legend(loc='upper left')
```

```
Out[3]:<matplotlib.legend.Legend at 0x7feb24b83110>
```



5.2 Armazenamento de figuras em arquivo

Uma das funções da Matplotlib é auxiliar os pesquisadores na preparação de gráficos para publicação em periódicos. Ao preparar um manuscrito, é comum o pesquisador se deparar com orientações como esta:

> Suas figuras deveriam ser preparadas com qualidade para publicação, utilizando aplicações capazes de gerar arquivos TIFF de alta resolução (1200 dpi para linhas e 300 dpi para arte colorida ou *half-tone*.
>
> In *Preparing Your Manuscript*, Oxford Journals

Exigências como a acima podem ser facilmente atendidas pela Matplotlib, que possui uma função `savefig` capaz de exportar o gráfico para um arquivo em disco, em diversos formatos, com resolução definida pelo pesquisador:

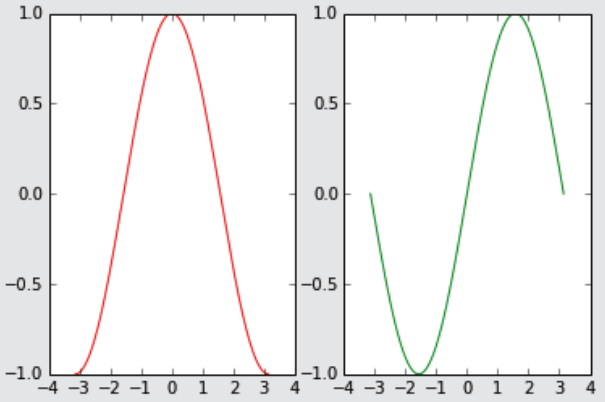
```
In [4]:fig.savefig('trig.tif', dpi=1200)
```

Na ausência de ilustrações *bitmap*, uma alternativa é armazenar a imagem em um formato **vetorial**, suportado por exemplos em arquivos PDF e EPS:

```
In [5]:fig.savefig('trig.pdf')
```

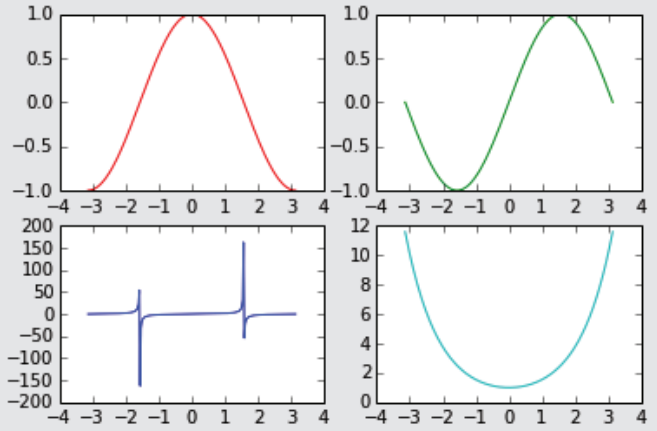
5.3 Subplots

```
In [6]:# 1 linha, 2 colunas, posição 1
       subplot(1, 2, 1)
       plot(X, C, 'r-')
       # 1 linha, 2 colunas, posição 2
       subplot(1, 2, 2)
       plot(X, S, 'g-')
```




```
Out[6]:[<matplotlib.lines.Line2D at 0x7feb24a619d0>]
In [7]:# 2 linhas, 2 colunas, posição 1
       subplot(2, 2, 1)
       plot(X, C, 'r-')
       # 2 linha, 2 colunas, posição 2
```

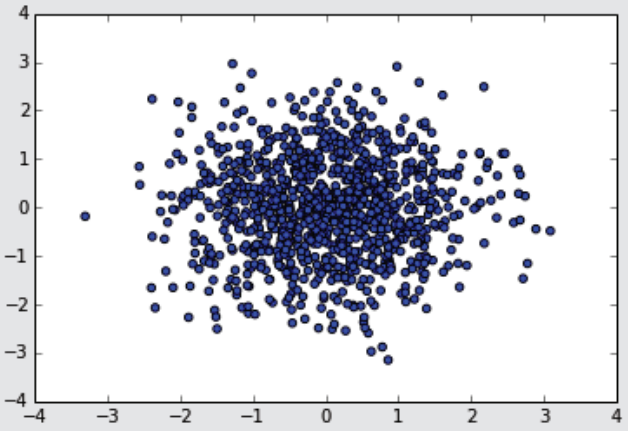
```
        subplot(2, 2, 2)
        plot(X, S, 'g-')
        # 2 linha, 2 colunas, posição 3
        subplot(2, 2, 3)
        plot(X, [tan(x) for x in X], 'b-')
        # 2 linha, 2 colunas, posição 4
        subplot(2, 2, 4)
        plot(X, [cosh(x) for x in X], 'c-')
Out[7]:[<matplotlib.lines.Line2D at 0x7feb1dfbcbd0>]
```



5.4 Outros tipos de gráficos

5.4.1 *Scatter plots*

```
In [8]:n = 1024
       X = random.normal(0,1,n)
       Y = random.normal(0,1,n)
       scatter(X,Y)
Out[8]:<matplotlib.collections.PathCollection at 0x7feb1ddfeb90>
```

Um clássico: *Iris Dataset* Este conjunto de dados é famoso na literatura de reconhecimento de padrões, sendo apresentado pela primeira vez por R. A. Fisher em 1950. Nele há 3 classes da planta *Iris*: *Iris Setosa*, *Iris Virginica* e *Iris Versicolor*. Cada classe possui 50 amostras com 4 medidas: comprimento e largura da sépala, comprimento e largura da pétala.

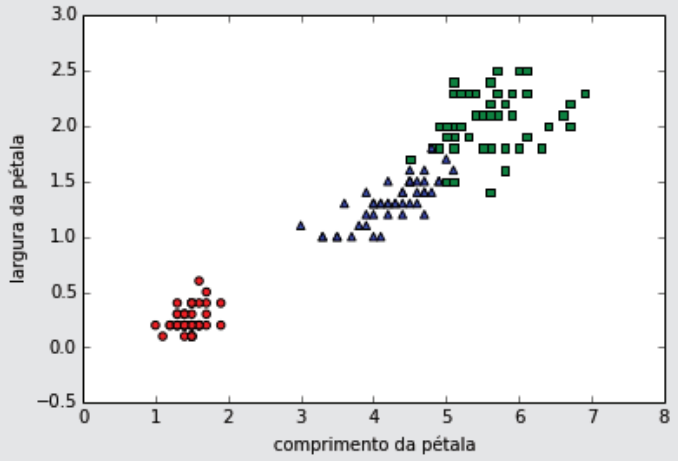
```
In [9]:!cat ./data/iris.data.txt
5.1      3.5       1.4      0.2      Iris-setosa
4.9      3.0      1.4      0.2      Iris-setosa
4.7      3.2      1.3      0.2      Iris-setosa
…
5.3      3.7      1.5      0.2      Iris-setosa
5.0      3.3      1.4      0.2      Iris-setosa
7.0      3.2      4.7      1.4      Iris-versicolor
6.4      3.2      4.5      1.5      Iris-versicolor
6.9      3.1      4.9      1.5      Iris-versicolor
…
5.1      2.5      3.0      1.1      Iris-versicolor
5.7      2.8      4.1      1.3      Iris-versicolor
6.3      3.3      6.0      2.5      Iris-virginica
5.8      2.7      5.1      1.9      Iris-virginica
7.1      3.0      5.9      2.1      Iris-virginica
…
6.2      3.4      5.4      2.3      Iris-virginica
5.9      3.0      5.1      1.8      Iris-virginica
```

O código abaixo carrega os dados das 3 classes de planta a partir do arquivo:

```
In [10]: iris_data = loadtxt('./data/iris.data.txt',
usecols=(0,1,2,3))
         iris_class = loadtxt('./data/iris.data.txt',
dtype='string')[:,4]
         setosa = iris_data[iris_class == 'Iris-setosa']
         virginica = iris_data[iris_class == 'Iris-virginica']
         versicolor = iris_data[iris_class == 'Iris-versicolor']
```

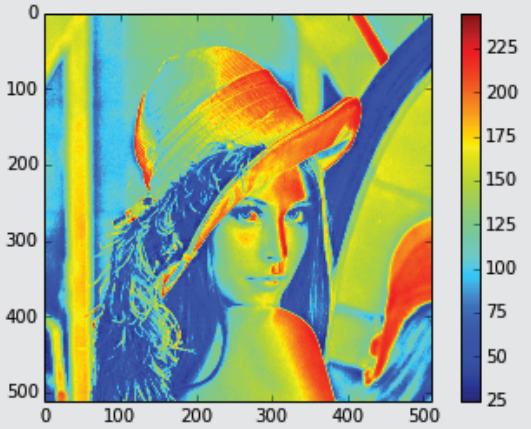
Podemos definir o símbolo e a cor utilizada em um *scatter plot*. No exemplo abaixo, círculos vermelhos representam os dados para *Setosa*, triângulos azuis representam *Versicolor* e quadrados verdes exibem o dados de *Virginica*:

```
In [11]: scatter(setosa[:,2], setosa[:,3], c='r', marker='o')
         scatter(virginica[:,2], virginica[:,3], c='g', marker='s')
         scatter(versicolor[:,2], versicolor[:,3], c='b',
marker='^')
         xlabel(u'comprimento da pétala')
         ylabel(u'largura da pétala')
Out[11]: <matplotlib.text.Text at 0x7feb1dd9dd50>
```



5.4.2 Imagens

```
In [12]: from scipy.misc import lena
         L = lena()
         imshow(L)
         colorbar()
Out[12]: <matplotlib.colorbar.Colorbar instance at 0x7feb4cc5c1b8>
```




```
In [13]: imshow(L, cmap=cm.gray)
         colorbar()
Out[13]: <matplotlib.colorbar.Colorbar instance at 0x7feb4cac2dd0>
```

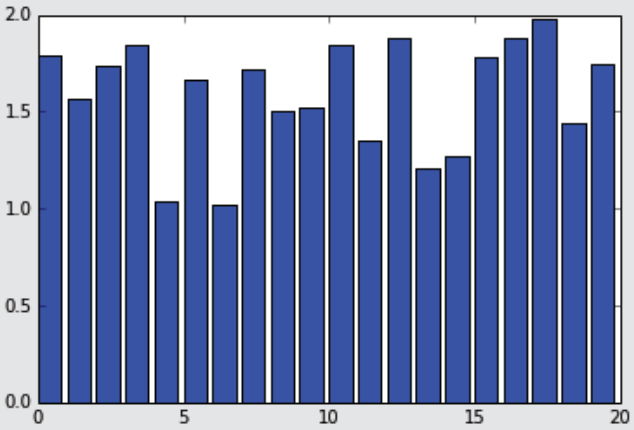
5.4.3 Barras

```
In [14]: x = arange(20)
         y = random.rand(20) + 1.
         print x
         print y
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[ 1.79041375  1.56426813  1.73814751  1.84023655  1.03542943
1.66087859
1.01735349  1.71556034  1.5047067   1.51906019  1.84412347
1.35404407
1.8778459   1.21032204  1.27278851  1.77880829  1.88269475
1.98243437
1.43757607  1.74274664]
```

```
In [15]: bar(x, y)
```

```
Out[15]: <Container object of 20 artists>
```



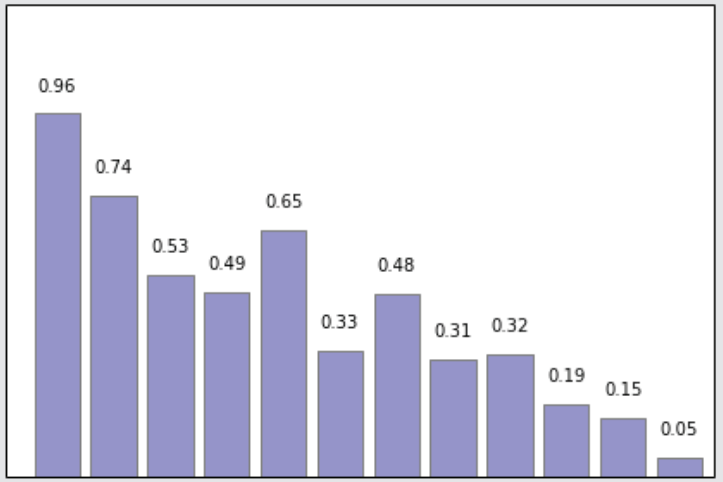
Um exemplo mais elaborado

```
In [16]: n = 12
         X = arange(n)
         Y = (1 - X / float(n)) * np.random.uniform(0.5, 1.0, n)

         axes([0.025, 0.025, 0.95, 0.95])
         bar(X, Y, facecolor='#9999ff', edgecolor='gray')

         for x, y in zip(X, Y):
             text(x + 0.4, y + 0.05, '%.2f' % y, ha='center',
va='bottom')
```



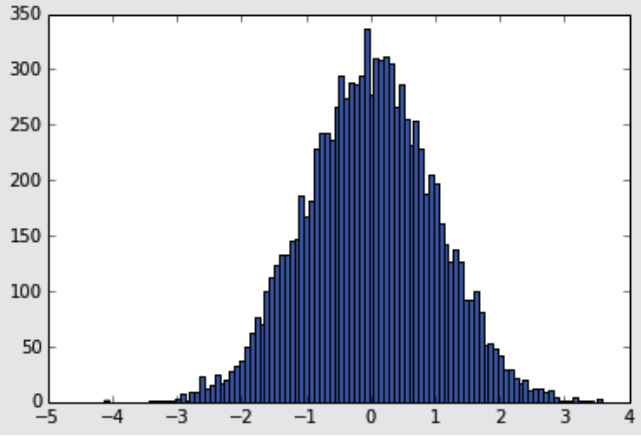
```
         xlim(-.5, n)
         xticks(())
         ylim(0, 1.25)
         yticks(())
```

```
Out[16]: ([], <a list of 0 Text yticklabel objects>)
```



5.4.4 Histogramas

```
In [17]: x = random.randn(10000)
         n, bins, patches = hist(x, 100)
```



5.5 A galeria da Matplotlib

Diversos exemplos de como utilizar a Matplotlib podem ser encontrados na galeria[11] em seu sítio web. Recomenda-se ao leitor também o tutorial *Matplotlib - 2D and 3D plotting in Python*, escrito por J. R. Johansson na forma de um IPython notebook[12].

6 A biblioteca SciPy

A biblioteca SciPy implementa uma vasta gama de algoritmos em computação científica, divididos em vários módulos:

- scipy.cluster - K-médias, vector quantization
- scipy.constants - constantes físicas e matemáticas
- scipy.fftpack - transformada de Fourier
- scipy.integrate - integração
- scipy.interpolate - interpolação
- scipy.io - IO, permite a leitura de arquivos .mat (MATLAB)
- scipy.linalg - álgebra linear
- scipy.ndimage - processamento de sinais -dimensionais
- scipy.odr - regressão
- scipy.optimize - otimização
  - BFGS, método de Newton, raízes de funções, simulated annealing,…
- scipy.signal - processamento de sinais
- scipy.sparse - matrizes esparsas
- scipy.spatial - algoritmos e estruturas de dados espaciais
  - KD-Trees, Voronoi, Delaunay e fecho convexo
- scipy.special - funções matemáticas diversas
  - Bessel, funções elípticas, hipergeométricas, parabólicas,…
- scipy.stats - estatística

SciPy pode ser vista como uma biblioteca equivalente à *GNU Scientific Library C/C++* (GSL) ou a várias *toolboxes* de MATLAB. SciPy faz uso da

[12] *Matplotlib gallery* <http:// matplotlib.sourceforge.net/ gallery.html>

[13] *Matplotlib - 2D and 3D plotting in Python* <http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scientic-python-lectures/master/Lecture-4-Matplotlib.ipynb>

NumPy para produzir implementações eficientes. O pesquisador deveria preferir implementações oferecidas pela SciPy, como argumentado por Haenel et al. (2013):

> Antes de implementar uma rotina, é importante checar se o processamento desejado já não se encontra implementado na SciPy. Como programadores, cientistas geralmente tendem a reinventar a roda, o que leva a código incorreto, não-otimizado e difícil de ser compartilhado e mantido. Em contraste, as rotinas da SciPy são otimizadas e testadas, e deveriam ser utilizadas sempre que possível.

Para ilustrar uso e potencial da biblioteca, nas próximas seções serão apresentadas algumas rotinas encontradas em três dos módulos da SciPy: `linalg` para álgebra linear, `optimize` para otimização contínua e `stats` para estatística.

6.1 Álgebra Linear - `scipy.linalg`

O módulo `scipy.linalg` fornece operações básicas de álgebra linear e baseia-se em bibliotecas nativas eficientes como BLAS e LAPACK. No exemplo abaixo, o determinante de uma matriz é computado:

```
In [8]:from scipy import linalg
       A = array([[1, 2],
       [3, 4]])
       linalg.det(A)
Out[8]:-2.0
```

Restrições envolvendo propriedades de matrizes são tratados como **exceções**. Python é uma linguagem que suporta exceções, que podem ser capturadas em tempo de execução e tratadas de acordo[14]. No exemplo abaixo, uma exceção, `ValueError`, é gerada ao tentarmos computar o determinante de uma matriz que não é quadrada:

```
In [9]:linalg.det(ones((3, 4)))
---------------------------------------------------------------------------
   ValueError                          Traceback (most recent call last)
```

[14] Ver *Errors and Exceptions* <https://docs.python.org/2/tutorial/errors.html> para detalhes.

```
       <ipython-input-9-737d1704a8e7> in <module>()
    ----> 1 linalg.det(ones((3, 4)))

       /usr/lib/python2.7/dist-packages/scipy/linalg/basic.pyc in
det(a, overwrite_a, check_finite)
       440          a1 = np.asarray(a)
       441      if len(a1.shape) != 2 or a1.shape[0] != a1.shape[1]:
    --> 442          raise ValueError('expected square matrix')
       443      overwrite_a = overwrite_a or _datacopied(a1, a)
       444      fdet, = get_flinalg_funcs(('det',), (a1,))

       ValueError: expected square matrix
```

6.1.1 Inversão de matrizes

```
In [10]: A = array([[1, 2],
                    [3, 4]])
         Ainv = linalg.inv(A)
         dot(A, Ainv)
Out[10]: array([[  1.00000000e+00,   0.00000000e+00],
                [  8.88178420e-16,   1.00000000e+00]])
```

Caso se tente obter a inversa de uma **matriz singular**, uma exceção `LinAlgError` é gerada:

```
In [11]: A = array([[3, 2],
                    [6, 4]])
         linalg.inv(A)
---------------------------------------------------------------------------
    LinAlgError                    Traceback (most recent call last)

         <ipython-input-11-467e3ab89c8b> in <module>()
           1 A = array([[3, 2],
           2            [6, 4]])
    ----> 3 linalg.inv(A)

         /usr/lib/python2.7/dist-packages/scipy/linalg/basic.pyc in
inv(a, overwrite_a, check_finite)
         381         inv_a, info = getri(lu, piv, lwork=lwork,
overwrite_lu=1)
         382     if info > 0:
    -->  383         raise LinAlgError("singular matrix")
         384     if info < 0:
```



```
         385         raise ValueError('illegal value in %d-th
argument of internal '

         LinAlgError: singular matrix
```

6.1.2 Decomposição em valores singulares

A **decomposição em valores singulares**, *singular value decomposition* (SVD) é uma fatorização especial de uma matriz real A com diversas aplicações práticas. A fatorização tem a forma

$$A = U \sum V^T$$

sendo $\sum$ uma matriz diagonal de números reais, U e V matrizes unitárias ($UU^T = I$). Tal fatorização apresenta as seguintes propriedades:

- as colunas de U correspondem aos autovetores de AA^T;
- as colunas de V correspondem aos autovetores de A^TA;
- os autovalores de AA^T e A^TA correspondem aos valores da diagonal de $\sum$.

Aplicação: resolução de sistemas homogêneos $Ax = 0$

```
In [12]: A = array([[ 0.84499381, -0.46944101],
                    [ 0.6248284 , -0.34712689],
                    [ 0.46448266, -0.25804592],
                    [ 0.64654437, -0.35919131],
                    [ 0.73794547, -0.4099697 ],
                    [ 0.8193507 , -0.45519483],
                    [ 0.92559289, -0.51421827],
                    [ 0.43297558, -0.24054199],
                    [ 0.35868939, -0.19927188]])
In [13]: U, s, VT = linalg.svd(A)
         V = VT.T
```

Se o sistema $Ax = 0$ tem solução, então o menor valor singular é zero.

```
In [14]: print s
         print abs(s[1]) < 1.0e-7
  [  2.32393669e+00   8.86916086e-09]
  True
```

Nesse caso, a solução se encontra no 2º valor singular à direita, ou seja, a última coluna de V:

```
In [15]: x = V[:,-1]
         print x
[ 0.48564293  0.87415728]
In [16]: dot(A, x)
Out[16]: array([-5.76358272e-09,  -2.36029274e-09,   9.10016895e-10,
                 4.39043324e-09,   3.21596083e-09,   1.09242354e-09,
                -1.15040977e-10,  -1.93379981e-09,   2.11648343e-09])
```

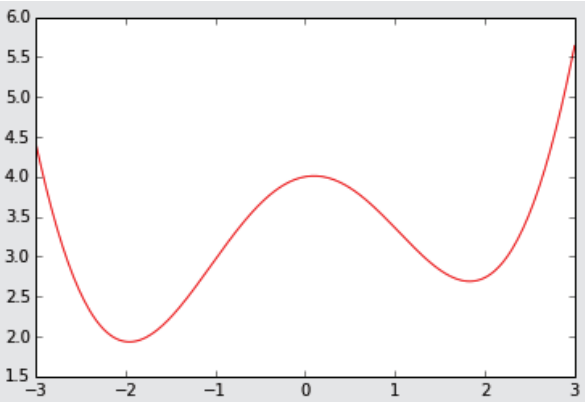
SVD possui diversas outras aplicações como a resolução de sistemas lineares super-determinados, isto é, sistemas com mais equações que incógnitas, e minimização por mínimos quadrados (GOLUB; LOAN, 2012).

6.2 Otimização - `scipy.optimize`

Otimização consiste em encontrar uma solução numérica que minimiza uma função. Este módulo fornece algoritmos para minimização de funções, *fitting* e busca de raízes (zeros de uma função). Como exemplo, considere a seguinte função:

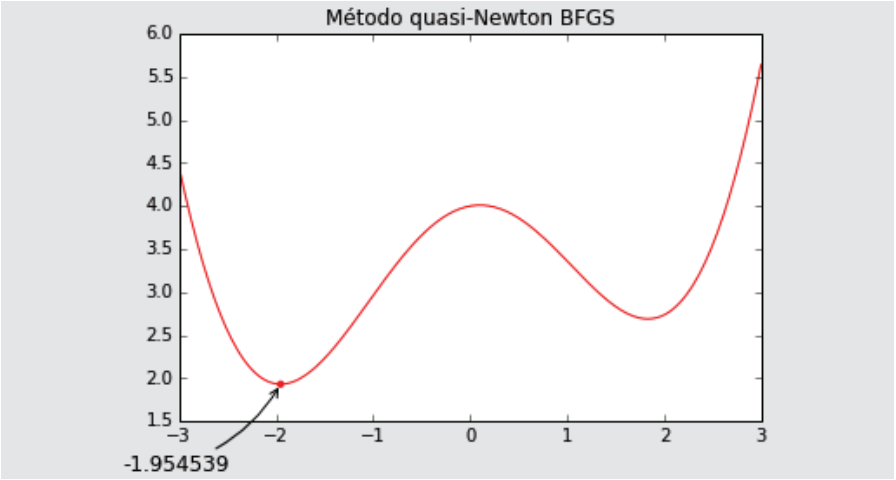
$$f(x) = x^2 + 0.2x + \cos(x).$$

```
In [17]: def f(x):
             return x**2 + 0.2*x + 4 * cos(x)
         x = linspace(-3, 3, 200)
         plot(x, f(x), 'r-')
         xlim((-3, 3))
Out[17]: (-3, 3)
```

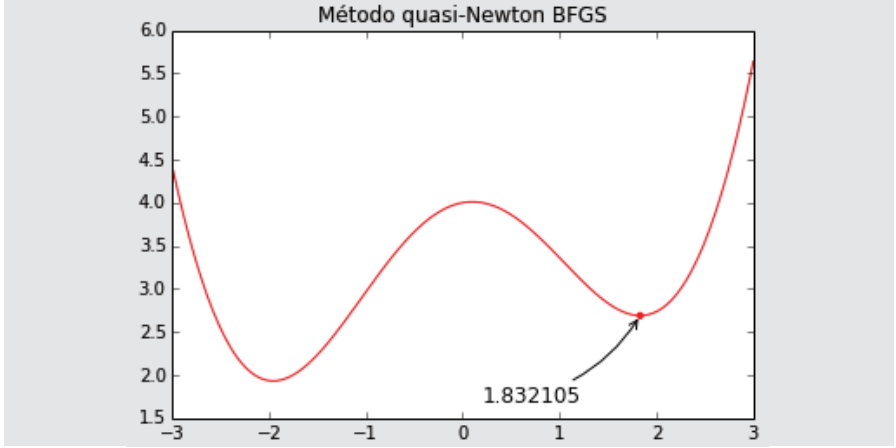
Uma alternativa para obter o mínimo de uma função é utilizar o método Broyden–Fletcher–Goldfarb–Shanno (BFGS), um método iterativo de otimização:

```
In [18]: from scipy.optimize import fmin_bfgs
         xmin = fmin_bfgs(f, 0)
         xmin
Optimization terminated successfully.
         Current function value: 1.931741
         Iterations: 3
         Function evaluations: 27
         Gradient evaluations: 9
Out[18]: array([-1.95453947])
In [19]: def show_opt_result(x, f, xmin, opt_title=None):
             plot(x, f(x), 'r-')
             xlim((-3,3))
             scatter([xmin],[f(xmin)], 10, color='red')
             annotate('%f' % xmin,
                      xy=(xmin, f(xmin)), fontsize=12,
xycoords='data',
                      xytext=(-90, -50), textcoords='offset points',
                      arrowprops=dict(arrowstyle="->",
connectionstyle="arc3,rad=.2"))
             if opt_title != None:
                 title(opt_title)
         show_opt_result(x, f, xmin, u'Método quasi-Newton BFGS')
```



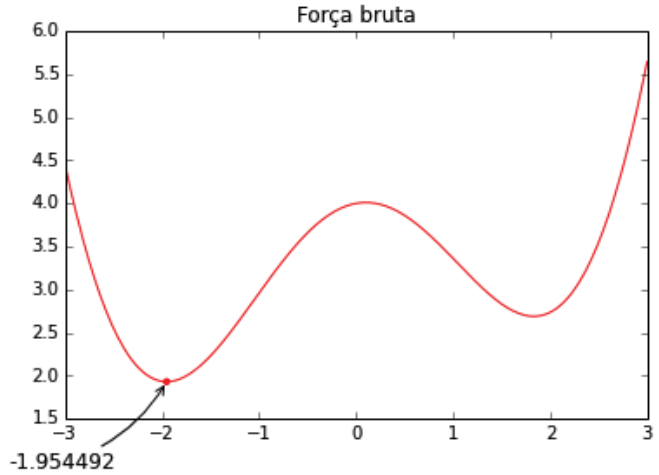
Um problema com este método é que dependendo do ponto inicial o processo de otimização pode ficar “preso” em um **mínimo local**. No exemplo anterior, o algoritmo foi inicializado em $x = 0$ (o segundo argumento de `fmin_bfgs`). Eis o resultado se a inicialização fosse $x = 1$:

```
In [20]: xmin = fmin_bfgs(f, 1, disp=0)
         xmin
         show_opt_result(x, f, xmin, u'Método quasi-Newton BFGS')
```



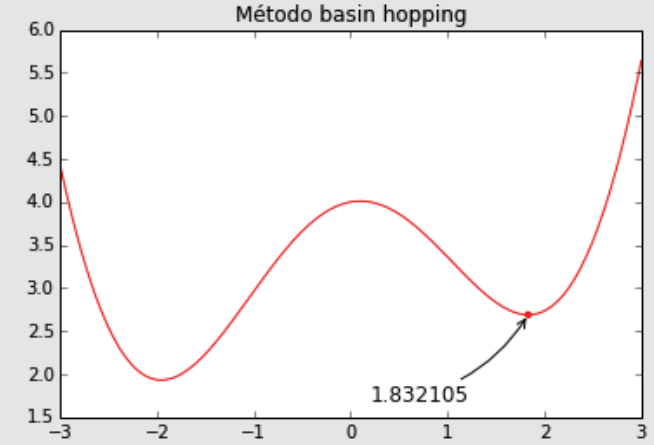
Podemos computar o mínimo global por **força bruta**, isto é, calculando o valor de $f(x)$ para *todo* x em uma certa *grade* (conjunto de valores em um intervalo espaçado).

```
In [21]: from scipy.optimize import brute
         grid = (-3, 3, 0.1)
         xmin = brute(f, (grid,))
         show_opt_result(x, f, xmin, u'Força bruta')
```

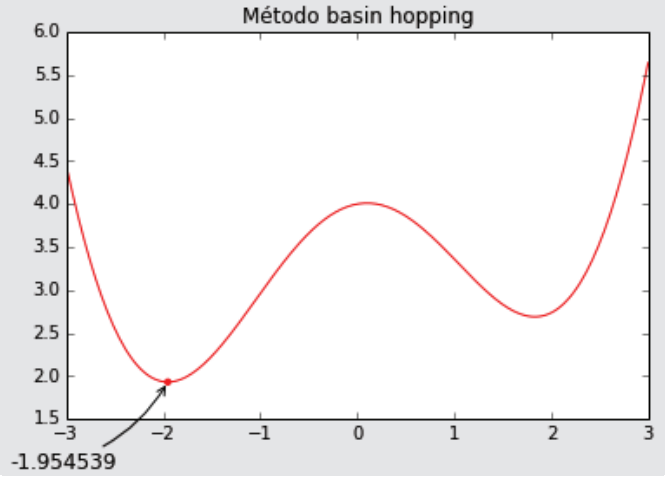


Obviamente, se a grade for muito grande, a execução de `brute` se torna **muito lenta**. Outros algoritmos podem ser utilizados como, por exemplo, *basin hopping*. Com um número suficiente de iterações, o método pode encontrar o mínimo global:

```
In [22]: from scipy.optimize import basinhopping
         # Número de iterações não é o suficiente
         xmin = basinhopping(f, 1, niter=200)
         xmin
Out[22]:     nfev: 3267
                fun: 2.6896496507852756
                  x: array([ 1.83210529])
            message: ['requested number of basinhopping iterations
completed successfully']
               njev: 1089
                nit: 200
In [23]: show_opt_result(x, f, xmin.x, u'Método basin hopping')
```



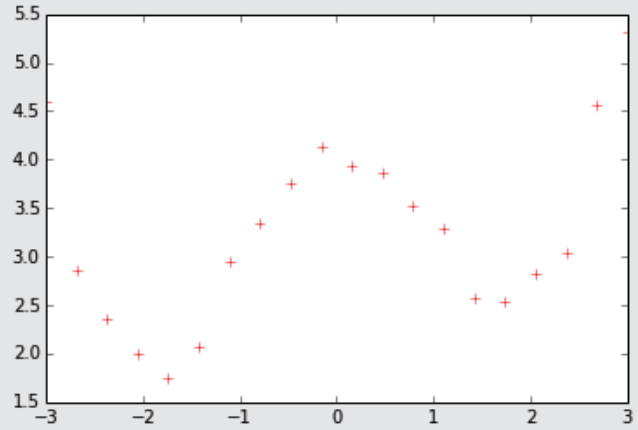

```
In [24]: # Novamente, com um maior número de iterações
         xmin = basinhopping(f, 1, niter=250)
         xmin
Out[24]:     nfev: 4176
                fun: 1.9317406957346788
                  x: array([-1.95453946])
            message: ['requested number of basinhopping iterations
completed successfully']
               njev: 1392
                nit: 250
In [25]: show_opt_result(x, f, xmin.x, u'Método basin hopping')
```



6.2.1 Fitting

Suponha que tenhamos obtido uma amostra de $f(x)$ com algum *ruído*:

```
In [26]: Xdata = linspace(-3, 3, num=20)
         Ydata = f(Xdata) + 0.2 * np.random.randn(Xdata.size)
         plot(Xdata, Ydata, 'r+')
         xlim((-3,3))
Out[26]: (-3, 3)
```



Suponha também que nós tenhamos um bom **modelo paramétrico**, mas com os parâmetros desconhecidos. Aqui, nosso modelo é da forma $\theta_1 x^2 + \theta_2 x + \theta_3 cos(x)$:

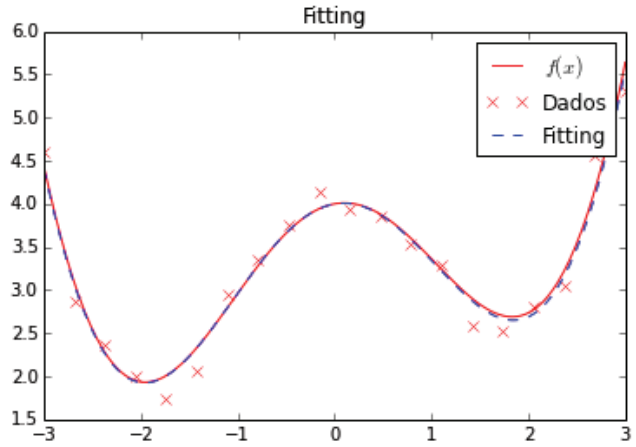
```
In [27]: def model(x, theta1, theta2, theta3):
             return theta1 * x**2 + theta2 * x + theta3 * cos(x)
```

Nós podemos estimar os parâmetros θ_1, θ_2 e θ_3 através do método de **mínimos quadrados**:

```
In [28]: from scipy.optimize import curve_fit
In [29]: palpite = [1,1,1]
         theta, theta_cov = curve_fit(model, Xdata, Ydata, palpite)
         theta
Out[29]: array([ 0.99306504,  0.19043778,  3.99791708])
```

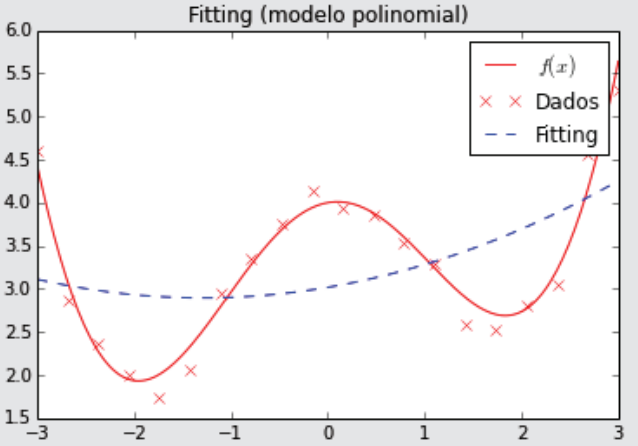
Note como o resultado acima é muito próximo da nossa função verdadeira, $f(x) = x^2 + 0.2x + 4\cos(x)$.

```
In [30]: X = linspace(-3, 3, num=200)
         plot(X, f(X), 'r-', label=r'$f(x)$')
         plot(Xdata, Ydata, 'rx', label=r'Dados')
         theta1, theta2, theta3 = theta
         Y_model = [model(x, theta1, theta2, theta3) for x in X]
         plot(X, Y_model, 'b--', label=r'Fitting')
         xlim((-3,3))
         title(u'Fitting')
         legend()
Out[30]: <matplotlib.legend.Legend at 0x7fc099b48b90>
```



O resultado obtido é satisfatório, contudo, o **modelo escolhido era perfeito**, isto é, apresentava a mesma forma paramétrica da verdadeira função $f(x)$. E se o modelo proposto fosse menos preciso? Abaixo, temos o resultado obtido por um modelo polinomial $\theta_1 x^2 + \theta_2 x + \theta_3$:

```
In [31]: def model2(x, theta1, theta2, theta3):
             return theta1 * x**2 + theta2 * x + theta3
In [32]: palpite = [1,1,1]
         theta, theta_cov = curve_fit(model2, Xdata, Ydata, palpite)
         theta
Out[32]: array([ 0.07386207,  0.19043777,  3.01696373])
In [33]: X = linspace(-3, 3, num=200)
         plot(X, f(X), 'r-', label=r'$f(x)$')
         theta1, theta2, theta3 = theta
         Y_model = [model2(x, theta1, theta2, theta3) for x in X]
         plot(X, Y_model, 'b--', label=r'Fitting')
         xlim((-3,3))
         title(u'Fitting (modelo polinomial)')
         legend()
Out[33]: <matplotlib.legend.Legend at 0x7fc08f8ce310>
```

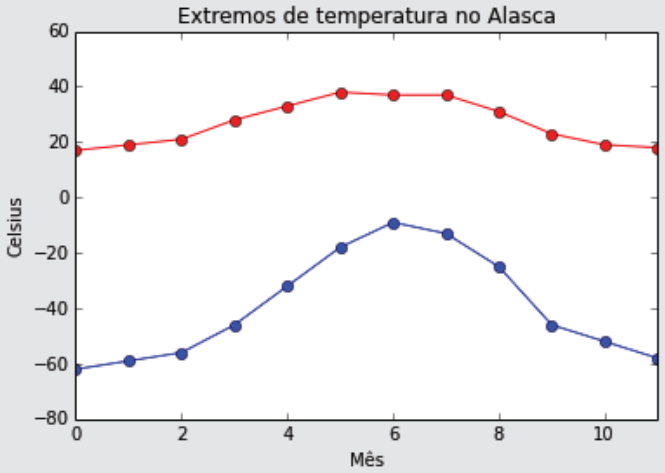


6.2.2 Exemplo - determinação de uma função para temperatura anual

Os vetores abaixo correspondem às temperaturas extremas do Alasca para cada mês (em ºC):

```
In [34]: tmax = [17,  19,  21,  28,  33,  38, 37,  37,  31,  23,
19,  18]
         tmin = [-62, -59, -56, -46, -32, -18, -9, -13, -25, -46,
-52, -58]
         month = arange(12)
In [35]: plot(month, tmin, 'bo-')
         plot(month, tmax, 'ro-')
         xlim(0,11)
         ylim(-80,60)
         ylabel(r'Celsius')
         xlabel(u'Mês')
         title(r'Extremos de temperatura no Alasca')
```

```
Out[35]: <matplotlib.text.Text at 0x7fc08f5f4f10>
```

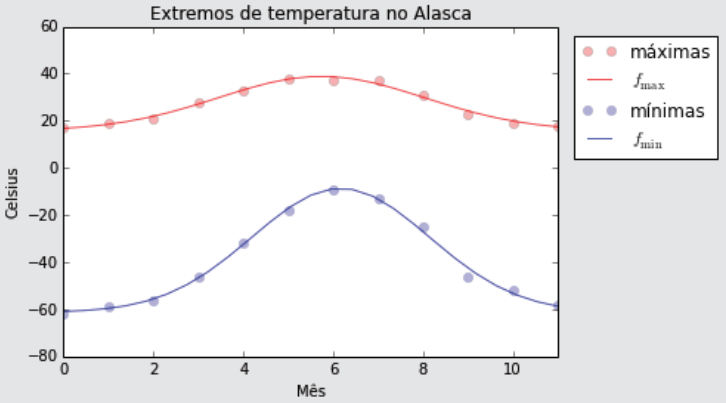


A partir desses dados, gostaríamos de estimar funções para prever a temperatura ao longo do ano. O formato se "sino" dos gráficos acima sugere que uma função Gaussiana poderia produzir um bom *fitting*:

```
In [36]: import scipy.optimize as opt

         def f(x, a, b, c, d):
             return a * exp(-(x-b)**2/(2*c**2)) + d

In [37]: theta_min, theta_min_cov = opt.curve_fit(f, month, tmin,
[10., 6, 2, -60])
         theta_min

Out[37]: array([ 52.64929193,   6.16452601,   1.9759001 ,
-61.32093597])

In [38]: theta_max, theta_max_cov = opt.curve_fit(f, month, tmax,
[10., 6, 2, -60])
         theta_max

Out[38]: array([ 23.09208263,   5.73581613,   2.2944347 ,
15.77932609])

In [39]: X = linspace(0, 11, 25)
         a, b, c, d = theta_max
         plot(month, tmax, 'ro', linewidth=3, alpha=0.3,
label=u'máximas')
         plot(X, [f(x, a, b, c, d) for x in X], 'r-', alpha=0.7,
label=r'$f_{\max}$')
```



```
         a, b, c, d = theta_min
         plot(month, tmin, 'bo', linewidth=3, alpha=0.3,
label=u'mínimas')
         plot(X, [f(x, a, b, c, d) for x in X], 'b-', alpha=0.7,
label=r'$f_{\min}$')
         ylabel(r'Celsius')
         xlabel(u'Mês')
         xlim(0,11)
         ylim(-80,60)
         legend(bbox_to_anchor=(1.34,1))
         title(r'Extremos de temperatura no Alasca')
Out[39]: <matplotlib.text.Text at 0x7fc08f76ff90>
```



6.3 Estatística - `scipy.stats`

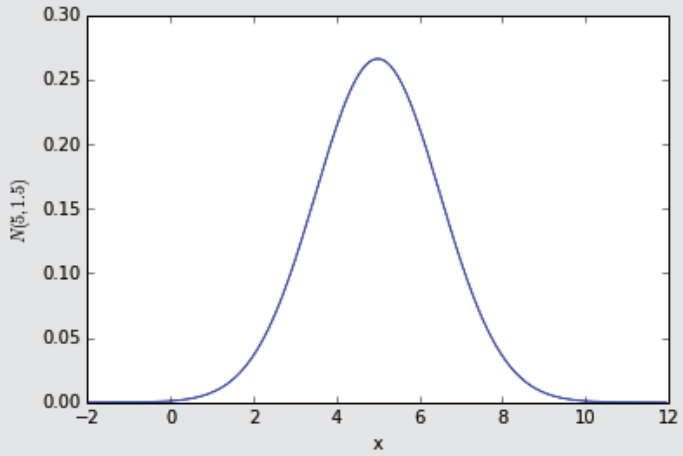
6.3.1 Funções de densidade de probabilidade

```
In [40]: from scipy import stats
```

O módulo `stats` fornece diversas funções de densidade de probabilidade (pdf). Abaixo, são computadas as probabilidades para todo ponto x em um certo domínio X, tomando-se uma distribuição normal com média igual a 5 e desvio padrão igual a 1,5.:

```
In [41]: X = arange(-2, 12, 0.05)
         plot(X, [stats.norm.pdf(x, 5., 1.5) for x in X])
         xlabel('x')
```

```
         ylabel(r'$N(5, 1.5)$')
Out[41]:<matplotlib.text.Text at 0x7fc08f3c5250>
```



6.3.2 *Fitting*

Uma atividade frequente em análise estatística é estimar os parâmetros de uma distribuição a partir de dados observados. Os métodos fit das várias distribuições encontradas em stats são capazes de realizar a estimação dos parâmetros por máxima verossimilhança, *maximum likelihood* estimate (MLE). Como exemplo, considere os dados de comprimento da sépala para Iris setosa:

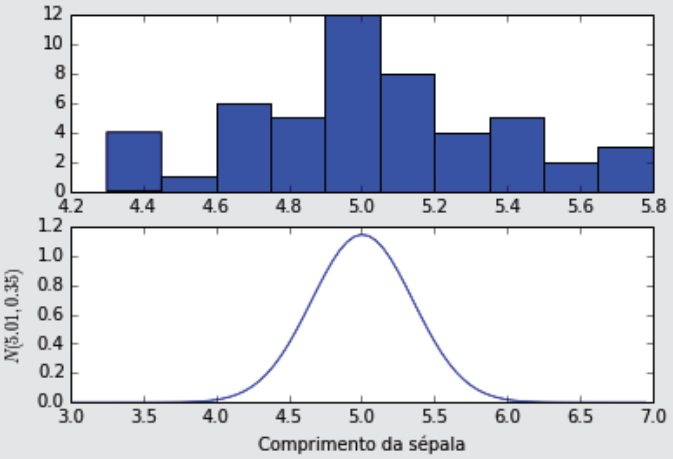
```
In [42]: iris_data = loadtxt('./data/iris.data.txt',
usecols=(0,1,2,3))
         iris_class = loadtxt('./data/iris.data.txt',
dtype='string')[:,4]
         setosa = iris_data[iris_class == 'Iris-setosa']
```

O exemplo abaixo ilustra a estimação para a distribuição normal:

```
In [43]: sep_len = setosa[:,0]
         mu, sig = stats.norm.fit(sep_len)
         mu, sig
Out[43]: (5.0059999999999993, 0.34894698737773899)
In [44]: subplot(2,1,1)
```



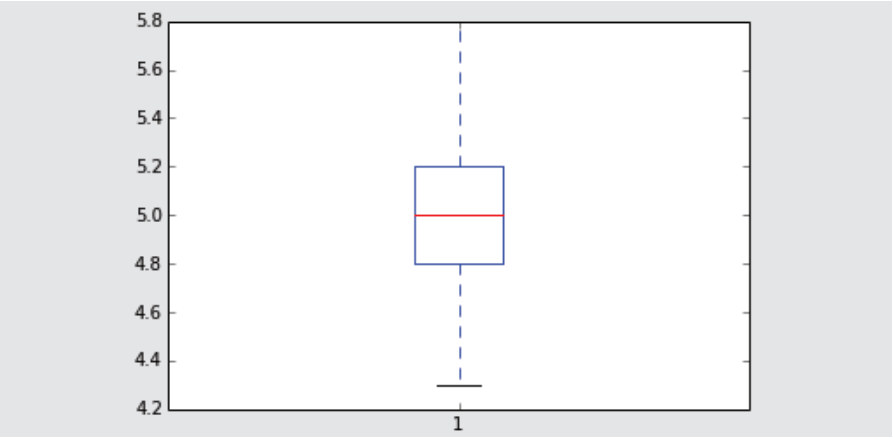
```
         n, pan, binstches = hist(sep_len, 10)
         subplot(2,1,2)
         X = arange(3, 7, 0.05)
         plot(X, [stats.norm.pdf(x, mu, sig) for x in X])
         xlabel(u'Comprimento da sépala')
         ylabel(r'$N(%.2f, %.2f)$' % (mu, sig))
Out[44]: <matplotlib.text.Text at 0x7fc08d34f550>
```



6.3.3 Percentis

Vamos olhar mais uma vez os dados sobre o comprimento de sépala em Iris setosa:

```
In [45]: boxplot(sep_len)
Out[45]: {'boxes': [<matplotlib.lines.Line2D at 0x7fc08d27c950>],
          'caps': [<matplotlib.lines.Line2D at 0x7fc08d290c90>,
           <matplotlib.lines.Line2D at 0x7fc08d27c310>],
          'fliers': [<matplotlib.lines.Line2D at 0x7fc08d036610>,
           <matplotlib.lines.Line2D at 0x7fc08d036fd0>],
          'medians': [<matplotlib.lines.Line2D at 0x7fc08d27cf90>],
          'whiskers': [<matplotlib.lines.Line2D at 0x7fc08d290350>,
           <matplotlib.lines.Line2D at 0x7fc08d2905d0>]}
```



Abaixo, computamos a mediana utilizando median da NumPy:

```
In [46]: np.median(sep_len)
Out[46]: 5.0
```

A mediana obviamente coincide com o percentil 50:

```
In [47]: stats.scoreatpercentile(sep_len, 50)
Out[47]: 5.0
```

Qualquer percentil pode ser recuperado:

```
In [48]: stats.scoreatpercentile(sep_len, 90)
Out[48]: 5.4100000000000001
In [49]: stats.scoreatpercentile(sep_len, 15)
Out[49]: 4.5999999999999996
```

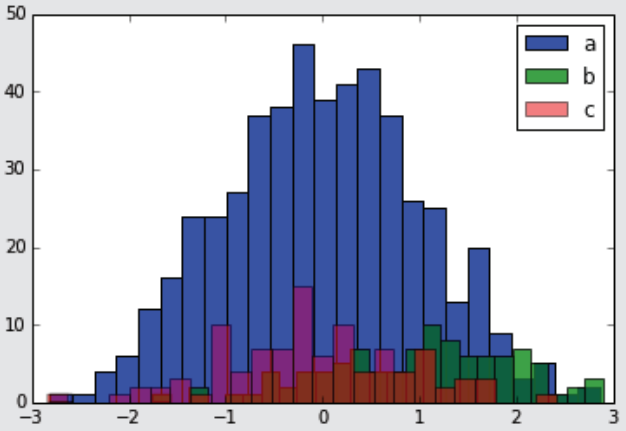
6.3.4 Testes estatísticos

O `teste t`-Student toma duas amostras independentes e verifica a hipótese de que a média das duas populações (assumindo a mesma variância) são iguais (hipótese nula - *null-hypothesis*). O teste avalia se a média esperada difere significativamente entre as duas amostras. Implementado na rotina `tttest_ind`, o teste também computa o **p-valor**. Se o valor do

p-valor for menor que um limiar (por exemplo, 1%), rejeita-se a hipótese de médias iguais.

Considere como exemplo três amostras a, b, e c. As amostras a e c são produzidas a partir de uma distribuição normal $N(0,1)$ enquanto que é oriunda de uma distribuição $N(1,1)$. Note que todas as distribuições têm a mesma variâncias, mas a distribuição de b difere das outras duas em relação à média:

```
In [50]: a = random.normal(0, 1, size=500)
         b = random.normal(1, 1, size=100)
         c = random.normal(0, 1, size=100)
In [51]: n, pan, binstches = hist(a, 25, label='a')
         n, pan, binstches = hist(b, 25, alpha=0.75, label='b')
         n, pan, binstches = hist(c, 25, alpha=0.5, label='c')
         legend()
Out[51]: <matplotlib.legend.Legend at 0x7fc08d2df090>
```



Como esperado, a e b diferem segundo o teste (note o baixo p-valor):

```
In [52]: stats.ttest_ind(a, b)
Out[52]: (array(-9.133939657615127), 1.012363940213494e-18)
```

O teste corretamente informa que não podemos descartar a hipótese nula para a e c:

```
In [53]: stats.ttest_ind(a, c)
Out[53]: (array(0.8294178095763991), 0.40719898095416118)
```

7 Comentários finais

O ambiente de computação científica em Python rivaliza outras soluções como MATLAB e R. Em particular, os recursos providos pelos IPython notebooks têm atraído a atenção de diversos pesquisadores devido à facilidade com que notas de pesquisa e código podem ser mantidos.

É importante notar que diversos outros módulos Python para computação científica estão disponíveis, embora não sejam considerados (até o momento) parte da SciPy Stack. Entre eles podemos citar estes:

- PyMC[15] módulo destinado à programação probabilística e análise Bayesiana (DAVIDSON-PILON, 2013).
- Pylearn[16] é um módulo destinado a aprendizado de máquina com ênfase nos recentes métodos de *deep learning* (GOODFELLOW et al., 2013).
- NetworX[17] provê funcionalidade para criação, manipulação e análise de redes complexas (grafos).

Devido aos esforços de sua grande e ativa comunidade de usuários, o ambiente científico Python deverá continuar uma opção competitiva em computação científica por vários anos.

8 Referências

DAVIDSON-PILON, C. **Probabilistic Programming and Bayesian Methods for Hacker**. 2013. Disponível em: <http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/>.

[15] PyMC. Disponível em: <http://pymc-devs.github.io/pymc>.
[16] Pylearn2. Disponível em: <http://deeplearning.net/software/pylearn2>.
[17] NetworX. Disponível em: <https://networkx.github.io>.

DHAR, V. Data science and prediction. **Communications of the ACM**, v. 56, n. 12, p. 64-73, dez. 2013. ISSN 00010782.

GOLUB, G. H.; LOAN, C. F. V. **Matrix computations**. [S.l.]: JHU Press, 2012.

GOODFELLOW, I. J. et al. Pylearn2: a machine learning research library. **arXiv preprint arXiv**:1308.4214, 2013. Disponível em: <http://arxiv.org/abs/1308.4214>.

HAENEL, V.; GOUILLART, E.; VAROQUAUX, G. (Ed.). **Python Scientific Lecture Notes**. 2013. Disponível em: <http://scipy-lectures.github.com>.

HEY, A. J. et al. The fourth paradigm: data-intensive scientific discovery. Microsoft Research Redmond, WA, 2009.

HUNTER, J. D. Matplotlib: A 2D Graphics Environment. **Computing in Science & Engineering**, v. 9, n. 3, p. 90-95, 2007. ISSN 1521-9615.

OLIPHANT, T. E. **Python for Scientific Computing**. Computing in Science & Engineering, v. 9, n. 3, p. 10-20, 2007. ISSN 1521-9615. Disponível em: <http://scitation.aip.org/content/aip/journal/cise/9/3/10.1109/MCSE.2007.58>.

PEREZ, F.; GRANGER, B. E. IPython: A System for Interactive Scientific Computing. **Computing in Science & Engineering**, v. 9, n. 3, p. 21-29, 2007. ISSN 1521-9615.

PEREZ, F.; GRANGER, B. E.; HUNTER, J. D. Python: An Ecosystem for Scientific Computing. **Computing in Science & Engineering**, v. 13, n. 2, p. 13-21, mar. 2011. ISSN 1521-9615. Dispon´ıvel em: <http://scitation.aip.org/content/aip/journal/cise/13/2/10.1109/MCSE.2010.119>.

POPPER, K. **Conjectures and Refutations**: The Growth of Scientific Knowledge. [S.l.]: Taylor & Francis, 2014. (Routledge Classics). ISBN 9781135971373.

SHEN, H. Interactive notebooks: Sharing the code. **Nature**, v. 515, n. 7525, p. 151-152, nov. 2014. ISSN 0028-0836. Disponível em: <http://www.nature.com/doifinder/10.1038/515151a>.

VANDEWALLE, P.; KOVACEVIC, J.; VETTERLI, M. Reproducible research in signal processing. **IEEE Signal Processing Magazine**, v. 26, n. 3, p. 37-47, maio 2009. ISSN 1053-5888.

WALT, S. van der; COLBERT, S. C.; VAROQUAUX, G. The NumPy Array: A Structure for Efficient Numerical Computation. **Computing in Science & Engineering**, v. 13, n. 2, p. 22-30, mar. 2011. ISSN 1521-9615.

Embrapa

Informática Agropecuária

Ministério da
Agricultura, Pecuária
e Abastecimento

GOVERNO FEDERAL
BRASIL
PAÍS RICO É PAÍS SEM POBREZA

CGPE 11887