

Matemática Aplicada à Engenharia

Nelson Luís Dias

Departamento de Engenharia Ambiental e
Lemma — Laboratório de Estudos em Monitoramento e
Modelagem Ambiental
Universidade Federal do Paraná

30 de setembro de 2011

Sumário

1	Ferramentas computacionais	11
1.1	Antes de começar a trabalhar,	11
1.2	Python	13
1.2.1	Strings e Inteiros	13
1.2.2	Números de ponto flutuante “reais” e “complexos”	15
1.2.3	Obtenção de uma curva de permanência	16
1.2.4	Arquivos texto e arquivos binários	19
1.3	Gnuplot	22
1.3.1	Ajuste de uma função	24
1.4	Python de novo: projeto probabilístico	26
1.5	Maxima	30
2	Um pouco de polinômios, integrais, séries . . .	33
2.1	Integração numérica: motivação	33
2.2	A regra do trapézio	37
2.3	Aproximação de integrais com séries: a função erro	42
3	Solução numérica de equações diferenciais ordinárias	49
3.1	Solução numérica de equações diferenciais ordinárias	49
3.1.1	Solução numérica; método de Euler	50
3.1.2	Um método implícito, com tratamento analítico	54
3.1.3	Runge-Kutta	54
4	Solução numérica de equações diferenciais parciais	61
4.1	Advecção pura: a onda cinemática	61
4.2	Difusão pura	71
4.3	Difusão em 2 Dimensões: ADI, e equações elíticas	83
A	Dados de vazão média anual e vazão máxima anual, Rio dos Patos, 1931–1999	89

Lista de Tabelas

1.1	Vazões Máximas Anuais ($\text{m}^3 \text{s}^{-1}$) no Rio dos Patos, PR, 1931–1999 .	16
2.1	Estimativas numéricas de I e seus erros relativos δ	38

Lista de Figuras

1.1	Convenções do terminal <code>postscript eps monochrome 'Times-Roman'</code> 18	24
1.2	FDA da vazão máxima anual, Rio dos Patos	25
1.3	Ajuste de uma FDA Weibull aos dados de vazão máxima anual do Rio dos Patos	27
2.1	Integração numérica de uma função	35
2.2	A regra do trapézio, com $n = 4$ e $n = 8$ trapézios.	38
2.3	Função $\text{erf}(x)$ calculada por integração numérica, com <code>trapepsilon</code> e $\epsilon = 1 \times 10^{-6}$, <i>versus</i> a <code>erf</code> pré-definida em Gnuplot.	44
2.4	Função $\text{erf}(x)$ calculada com série de Taylor, com <code>erf_1</code> , <i>versus</i> a <code>erf</code> pré-definida em Gnuplot.	45
3.1	Solução da equação (3.1).	50
3.2	Comparação da solução analítica da equação (3.1) com a saída de <code>sucesso.py</code> , para $\Delta x = 0,01$	53
3.3	Comparação da solução analítica da equação (3.1) com a saída de <code>sucesso.py</code> , para $\Delta x = 0,5$	53
3.4	Comparação da solução analítica da equação (3.1) com a saída de <code>sucimp.py</code> , para $\Delta x = 0,5$	55
3.5	Comparação da solução analítica da equação (3.1) com a saída de <code>euler2.py</code> , para $\Delta x = 0,5$	56
3.6	Comparação da solução analítica da equação (3.1) com a saída de <code>rungek4.py</code> , para $\Delta x = 0,5$	59
4.1	Condição inicial da equação 4.3.	61
4.2	Solução numérica produzida por <code>onda1d-ins.py</code> , para $t = 250\Delta t$, $500\Delta t$ e $750\Delta t$	63
4.3	Solução numérica produzida por <code>onda1d-lax.py</code> , para $t = 500\Delta t$, $1000\Delta t$ e $1500\Delta t$	70
4.4	Solução numérica produzida pelo esquema <i>upwind</i> , para $t = 500\Delta t$, $1000\Delta t$ e $1500\Delta t$	71
4.5	Solução analítica da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$	74

4.6	Solução numérica com o método explícito (4.35) (círculos) <i>versus</i> a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.	77
4.7	Solução numérica com o método implícito (4.39) (círculos) <i>versus</i> a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.	80
4.8	Solução numérica com o método de Crank-Nicholson ((4.45)) (círculos) <i>versus</i> a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.	83
4.9	Solução analítica da equação da difusão bidimensional, para $t = 0$, $t = 0$, $t = 0,1$, $t = 0,2$ e $t = 0,3$	87
4.10	Solução numérica da equação da difusão bidimensional com o esquema ADI, para $t = 0$, $t = 0$, $t = 0,1$, $t = 0,2$ e $t = 0,3$	88

Lista de Listagens

1.1	<code>binint.py</code> — exemplo de <i>strings</i> , e inteiros.	13
1.2	<code>floats.py</code> — exemplo de uso de <code>float</code> e <code>complex</code>	15
1.3	<code>patos-medmax.dat</code> — vazões média e máxima anuais, Rio dos Patos	17
1.4	<code>fqiemp.py</code> — cálculo de uma FDA empírica	18
1.5	<code>fqiemp.dat</code> — FDA empírica da vazão máxima anual no Rio dos Patos.	19
1.6	<code>bintext.py</code> — Exemplo de arquivo texto e arquivo binário	20
1.7	<code>writearr.py</code> — Escreve um arquivo binário contendo 3 “linhas”, cada uma das quais com um <i>array</i> de 10 <code>float</code> ’s.	21
1.8	<code>readarr.py</code> — Escreve um arquivo binário contendo 3 “linhas”, cada uma das quais com um <i>array</i> de 10 <code>float</code> ’s.	22
1.9	<code>figemp.plt</code> — programa para plotar a FDA empírica do Rio dos Patos	23
1.10	<code>weibull.plt</code> — como ajustar uma FDA analítica (Weibull) aos dados da FDA empírica	25
1.11	<code>interp.py</code> — módulo com função para calcular uma interpolação linear.	28
1.12	<code>ppbemp.py</code> — projeto probabilístico a partir da distribuição empírica	29
1.13	<code>mulambdak.max</code> — cálculo da média da distribuição Weibull em em função de seus parâmetros λ e k	31
2.1	<code>achapol.max</code> — Polinômio com propriedades definidas	34
2.2	Saída de <code>achapol.max</code>	34
2.3	<code>passaquad.max</code> — parábola $h(x) = ax^2 + bx + c$ passando por (1, $f(1)$), (3, $f(3)$) e (5, $f(5)$).	36
2.4	Saída de <code>passaquad.max</code>	37
2.5	<code>numint.py</code> — Integração numérica, regra do trapézio	39
2.6	<code>quadraver1.py</code> — Integração numérica de $f(x)$ com 8 trapézios	39
2.7	<code>numint.py</code> — Integração numérica ineficiente, com erro absoluto pré- estabelecido	40
2.8	<code>quadraver2.py</code> — Integração numérica ineficiente de $f(x)$ com $\epsilon =$ 0,0001	40
2.9	<code>numint.py</code> — Integração numérica eficiente, com erro absoluto pré- estabelecido	41
2.10	<code>quadraver3.py</code> — Integração numérica eficiente de $f(x)$ com $\epsilon =$ 0,000001	41
2.11	<code>vererf.py</code> — Cálculo da função erro por integração numérica	43
2.12	<code>vererf.plt</code> — Plotagem da função erro por integração numérica <i>ver-</i> <i>sus</i> a $\text{erf}(x)$ pré-definida em Gnuplot	43
2.13	Cálculo de $\text{erf}(x)$ com uma série de Taylor.	46
2.14	<code>vererf1.py</code> — Cálculo da função erro com série de Taylor entre 0 e 3.	46

2.15	<code>vererf1.plt</code> — Plotagem da função erro calculada com série de Taylor <i>versus</i> a $\text{erf}(x)$ pré-definida em Gnuplot	47
2.16	<code>erfs.py</code> — Cálculo de $\text{erf}(x)$ com série de Taylor, limitado a no máximo 43 termos	48
3.1	<code>resolve-eqdif</code> — Solução de uma EDO com Maxima	49
3.2	<code>fracasso.py</code> — Um programa com o método de Euler que não funciona	51
3.3	Saída de <code>fracasso.py</code>	51
3.4	<code>sucesso.py</code> — Um programa com o método de Euler que funciona .	52
3.5	<code>sucimp.py</code> — Método de Euler implícito	55
3.6	<code>euler2</code> — Um método explícito de ordem 2	57
3.7	<code>rungek4</code> — Método de Runge-Kutta, ordem 4	58
4.1	<code>onda1d-ins.py</code> — Solução de uma onda 1D com um método explícito instável	63
4.2	<code>surf1d-ins.py</code> — Seleciona alguns intervalos de tempo da solução numérica para plotagem	64
4.3	<code>onda1d-lax.py</code> — Solução de uma onda 1D com um método explícito laxtável	68
4.4	<code>surf1d-lax.py</code> — Seleciona alguns intervalos de tempo da solução numérica para plotagem	69
4.5	<code>difusao1d-ana.py</code> — Solução analítica da equação da difusão	73
4.6	<code>divisao1d-ana.py</code> — Seleciona alguns instantes de tempo da solução analítica para visualização	74
4.7	<code>difusao1d-exp.py</code> — Solução numérica da equação da difusão: método explícito.	76
4.8	<code>divisao1d-exp.py</code> — Seleciona alguns instantes de tempo da solução analítica para visualização	77
4.9	<code>alglin.py</code> — Exporta uma rotina que resolve um sistema tridiagonal, baseado em Press et al. (1992)	80
4.10	<code>difusao1d-imp.py</code> — Solução numérica da equação da difusão: método implícito.	81
4.11	<code>difusao1d-ckn.py</code> — Solução numérica da equação da difusão: esquema de Crank-Nicholson.	84
	<code>patos-medmax.dat</code>	89

1

Ferramentas computacionais

Neste capítulo nós fazemos uma introdução muito rápida a 3 ferramentas computacionais.

Python é uma *linguagem de programação*, razoavelmente clássica, e da qual nós vamos explorar apenas uma parte chamada de *programação procedural*. A escolha de Python deve-se ao fato de ela ser uma linguagem fácil de aprender, e de existirem muitas *bibliotecas* de rotinas em Python que tornam muitas das tarefas de Matemática Aplicada fáceis de implementar em um computador.

Gnuplot é uma linguagem para gerar gráficos. Ela é bastante popular, fácil de usar, e muito flexível. Nós vamos gerar todos os nossos gráficos bidimensionais (gráficos “ $x \times y$ ”) com Gnuplot.

Maxima é uma linguagem de processamento simbólico. Da mesma maneira que nós faremos contas com Python, nós faremos *álgebra* com Maxima. Os usos que faremos de Maxima estarão longe de explorar todo o seu potencial. Nós vamos apenas calcular algumas integrais, algumas séries de Taylor, e resolver algumas equações diferenciais ordinárias; entretanto, a rapidez com que faremos isto justifica amplamente o seu uso.

Infelizmente, a versão de um programa de computador faz diferença, e às vezes faz *muita* diferença. As versões que eu utilizei para este texto foram:

Python: 2.6.5

Gnuplot: 4.4

Maxima: 5.21.1

Neste momento, existem duas linhagens de Python convivendo: 2.x, e 3.x. O futuro é Python 3.x, e eu procurei usar uma sintaxe tão próxima de 3.x quanto possível. Isto é facilitado por comandos do tipo

```
[ from __future__ import ],
```

que você encontrará diversas vezes ao longo do texto. Se você (já) estiver usando Python 3.x, remova estes comandos do início de seus arquivos .py.

1.1 – Antes de começar a trabalhar,

Você precisará de algumas condições de “funcionamento”. Eis os requisitos fundamentais:

- 1) Saber que sistema operacional você está usando.
- 2) Saber usar a linha de comando, ou “terminal”, onde você datilografa comandos que são em seguida executados.
- 3) Saber usar um *editor* de texto, e salvar seus arquivos com codificação ISO-8859-1.
- 4) Certificar-se de que você tem Python instalado.
- 5) Certificar-se de que você tem Numpy (um módulo de Python que deve ser instalado à parte, e que vamos utilizar seguidamente) instalado.
- 6) Certificar-se de que você tem Gnuplot instalado.
- 7) Certificar-se de que você tem Maxima instalada.

Atenção! Um editor de texto não é um processador de texto. Um editor de texto não produz letras de diferentes tamanhos, não cria tabelas, e não insere figuras. Um editor de texto reproduz o texto que você datilografa, em geral com um tipo de largura constante para que as colunas e espaços fiquem bem claros. Um editor de texto que “vem” com Windows chama-se *notepad*, ou *bloco de notas* nas versões em Português; um excelente substituto chama-se *notepad2* (<http://www.flos-freeware.ch/notepad2.html>). Em Linux, editores de texto simples são o *gedit*, e o *kate*. Programadores mais experientes costumam preferir o *vim*, ou o *emacs*. Estes dois últimos possuem versões para os 3 sistemas operacionais mais comuns hoje em dia: Windows, Linux e MacOS.

Quando você estiver praticando o uso das ferramentas computacionais descritas neste texto, suas tarefas invariavelmente serão:

- 1) Criar o arquivo com o programa em Python, Gnuplot, ou Maxima, usando o editor de texto, e salvá-lo em codificação ISO-8859-1.

Se não for óbvio para você, procure descobrir como você pode configurar o seu editor para salvar em ISO-8859-1. Se isto for impossível para você, trabalhe normalmente, mas não use acentos: em lugar de *impossível*, digite *impossivel*.

- 2) Ir para a linha de comando.
- 3) Executar o programa digitando o seu nome (*e não clicando!*), possivelmente precedido por *python*, *gnuplot*, ou *maxima*.
- 4) Verificar se o resultado está correto.
- 5) Se houver erros, voltar para 1), e reiniciar o processo.

Neste texto, eu vou partir do princípio de que todas estas condições estão cumpridas por você, mas não vou detalhá-las mais: em geral, sistemas operacionais, editores de texto e ambientes de programação variam com o gosto do freguês: escolha os seus preferidos, e bom trabalho!

Listagem 1.1: `binint.py` — exemplo de *strings*, e inteiros.

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  a = 'açafrão'                                # a é uma string
6  print(a)                                     # imprime a na tela
7  print(len(a))                               # imprime o número de caracteres de a
8  i = 2**32 - 1                                # i é o maior int s/sinal que cabe em 32 bits
9  b = unicode(i)                              # converte i na string correspondente b
10 print(b)                                    # imprime b na tela
11 print('---a:')                             # separa a saída do primeiro for
12 for c in a:                                  # p/ cada c de a, imprime seu valor unicode
13     print('ord(', c, ')_=_', ord(c))
14 print('---b:')                             # separa a saída do segundo for
15 for c in b:                                  # p/ cada c de b, imprime seu valor unicode
16     print('ord(', c, ')_=_', ord(c))
17 print(unichr(227))                          # imprime o caractere unicode no 227

```

Exercícios Propostos

1.1 Você já devia estar esperando por isto: o que é um sistema operacional? Qual é o sistema operacional que você usa?

1.2 Como saber se Python está instalado?

1.2 – Python

Python reconhece os seguintes tipos “básicos” de variáveis: *strings*, números inteiros, números de ponto flutuante, e números complexos.

1.2.1 – Strings e Inteiros

Strings, ou cadeias de caracteres, são criaturas do tipo `'abacaxi'`, e números inteiros são criaturas do tipo `-1`, `0`, e `32767`. O *tipo* das strings que vamos usar chama-se `unicode` em Python 2.x, e `str` em Python 3.x. O tipo dos números inteiros chama-se `int` em Python.

A listagem 1.1 mostra o conteúdo do arquivo `binint.py` com alguns exemplos simples do uso de inteiros e *strings*. A legenda de cada listagem se inicia sempre com o nome do arquivo correspondente (após um pouco de hesitação, eu decidi não disponibilizar estes arquivos; para testar os exemplos deste texto, você precisará digitar os arquivos novamente: isto o (a) forçará a praticar programação). A listagem é um retrato fiel do arquivo, com duas exceções: as *palavras reservadas* de Python estão sublinhadas na listagem (mas não no arquivo), e os espaços em branco *dentro das strings* estão enfatizados pelo símbolo `_`.

Atenção: os números que aparecem à esquerda da listagem não fazem parte do arquivo. Em Python, um comentário inicia-se com `#`, e prossegue até o fim de linha. A maior parte dos comandos de `binint.py` está explicada nos próprios comentários. Alguns comentários (sem intenção de trocadilho) adicionais, por linha:

- 1 Este comentário especial torna o arquivo executável em Linux. O caminho `/usr/bin/python` pode variar com a instalação, e mais ainda com o sistema operacional.

- 2 Este comentário informa que o arquivo está codificado em iso-8859-1, não é óbvio?
- 3 Magia negra: tudo que é escrito entre aspas passa a ser uma *string* do tipo `unicode`.
- 4 Magia negra: `print` deixa de ser uma declaração, e passa a ser uma função. Se você não entendeu nada, não esquite.
- 8 O operador `**` significa exponenciação. Nesta linha, a variável `i` torna-se um `int`, e recebe o valor $2^{32} - 1$. Este é o maior valor *sem sinal* que um inteiro pode assumir, já que o maior inteiro que cabe em 32 bits é

$$\begin{aligned}
 &11111111111111111111111111111111 \text{ (binário)} = \\
 &1 \times 2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + \dots + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = \\
 &4294967295 \text{ (decimal)} .
 \end{aligned}$$

- 12-13 O comando `for` percorre em ordem um objeto *iterável* (no caso, a *string* 'açafrão' é iterável, com `a[0] == 'a'`, `a[1] == 'ç'`, etc.). O *corpo* do `for` tem que ser indentado, e na listagem a indentação é de 3 espaços. Desta forma, no caso da linha 13, o comando


```
[ print('ord(',c,') = ', ord(c)) ]
```

 é executado 7 vezes, uma para cada caractere de `a`. A volta à indentação anterior na linha 14 define o fim do `for`.

Vamos agora à saída do programa `binint.py`:

```

açafrão
7
4294967295
--- a:
ord( a ) = 97
ord( ç ) = 231
ord( a ) = 97
ord( f ) = 102
ord( r ) = 114
ord( ã ) = 227
ord( o ) = 111
--- b:
ord( 4 ) = 52
ord( 2 ) = 50
ord( 9 ) = 57
ord( 4 ) = 52
ord( 9 ) = 57
ord( 6 ) = 54
ord( 7 ) = 55
ord( 2 ) = 50
ord( 9 ) = 57
ord( 5 ) = 53
ã

```

`print` imprime com o formato apropriado inteiros e *strings* (e muito mais: quase tudo, de alguma forma!). O maior inteiro que cabe em 32 bits sem sinal é 4294967295 (como já vimos acima). A posição do caractere `a` na tabela Unicode é 97; a posição do caractere `ç` é 231; a posição do caractere `4` é 52. Finalmente, o caractere Unicode de número 227 é o `ã`.

Se você estiver vendo uma saída diferente da mostrada acima, com caracteres estranhos, não se assuste (demais): o seu arquivo `binint.py` e o terminal dentro do

Listagem 1.2: floats.py — exemplo de uso de float e complex.

```

1 #!/usr/bin/python
2 # -*- coding: iso-8859-1 -*-
3 from __future__ import unicode_literals
4 from __future__ import print_function
5 from math import pi, e, sin          # pi, e, e seno
6 from cmath import sin as csin        # o seno de um número complexo vem com outro
7                                     # nome, para não confundir
8 from cmath import sqrt as csqrt     # a raiz quadrada de um número complexo vem
9                                     # com outro nome, para não confundir
10 print('pi=_',pi)                    # imprime o valor de pi
11 print('e=_',e)                      # imprime o valor de e
12 print('sen(pi/2)=',sin(pi/2))        # imprime sen(pi/2)
13 i = csqrt(-1.0)                     # neste programa, i == sqrt(-1)
14 print('sen(i)=====',csin(i))        # imprime sen(i)

```

qual você está executando este programa certamente estão utilizando codificações diferentes (veja se o apêndice ?? pode ajudar).

1.2.2 – Números de ponto flutuante “reais” e “complexos”

Em primeiro lugar, um esclarecimento: no computador, não é possível representar todos os números $x \in \mathbb{R}$ do conjunto dos reais, mas apenas um *subconjunto* dos racionais \mathbb{Q} . Linguagens de programação mais antigas, como FORTRAN, ALGOL e PASCAL, mesmo assim chamavam estes tipos de REAL. A partir de C, e continuando com Python, em muitas linguagens passou-se a usar o nome mais adequado float.

Python vem com uma grande quantidade de *módulos* predefinidos, e você pode adicionar seus próprios módulos. Deles, importam-se variáveis e funções (e outras coisas) úteis. Nosso primeiro exemplo do uso de números de ponto flutuante (float) e “complexos” (complex) não podia ser mais simples, na listagem 1.2

Eis a saída de floats.py:

```

pi = 3.14159265359
e = 2.71828182846
sen(pi/2) = 1.0
sen(i) = 1.17520119364j

```

Os comentários importantes seguem-se. Lembre-se de que na maioria dos casos você está vendo *aproximações racionais* de números reais, e que o computador não pode lidar com todos os números reais.

Como era de se esperar, $\sin \pi/2 = 1$. Por outro lado, o seno de i é um número puramente imaginário, e vale $\approx 1,17520119364i$. Note que Python usa a letra j para indicar um valor imaginário (e não i). Na linha 13, eu “forcei a barra”, e criei a variável que, em notação matemática se escreveria $i = \sqrt{-1}$. Mas eu *também* poderia ter simplesmente eliminado esta linha, e substituído a linha 14 por

```
[ print('sen(i) = ',csin(1j)) ] .
```

Note que existem *dois* módulos, `math` e `cmath`, para variáveis “reais” e “complexas”, respectivamente. Em *ambos*, existe uma função denominada `sin`, que calcula o seno. Para poder usar estas duas funções *diferentes* em meu programa `floats.py`, eu *rebatizei* a função `sin` complexa de `csin`, no ato da importação do módulo, com o mecanismo `from ... import ... as ...` (linha 6).

Tabela 1.1: Vazões Máximas Anuais ($\text{m}^3 \text{s}^{-1}$) no Rio dos Patos, PR, 1931–1999

Ano	Vaz Máx	Ano	Vaz Máx	Ano	Vaz Máx	Ano	Vaz Máx
1931	272.00	1951	266.00	1971	188.00	1991	131.00
1932	278.00	1952	192.10	1972	198.00	1992	660.00
1933	61.60	1953	131.80	1973	252.50	1993	333.00
1934	178.30	1954	281.00	1974	119.00	1994	128.00
1935	272.00	1955	311.50	1975	172.00	1995	472.00
1936	133.40	1956	156.20	1976	174.00	1996	196.00
1937	380.00	1957	399.50	1977	75.40	1997	247.50
1938	272.00	1958	152.10	1978	146.80	1998	451.00
1939	251.00	1959	127.00	1979	222.00	1999	486.00
1940	56.10	1960	176.00	1980	182.00		
1941	171.60	1961	257.00	1981	134.00		
1942	169.40	1962	133.40	1982	275.00		
1943	135.00	1963	248.00	1983	528.00		
1944	146.40	1964	211.00	1984	190.00		
1945	299.00	1965	208.60	1985	245.00		
1946	206.20	1966	152.00	1986	146.80		
1947	243.00	1967	92.75	1987	333.00		
1948	223.00	1968	125.00	1988	255.00		
1949	68.40	1969	135.60	1989	226.00		
1950	165.00	1970	202.00	1990	275.00		

Exercícios Propostos

1.3 Usando Python, converta 7777 da base 10 para a base 2. Sugestão: estude a documentação em www.python.org, e encontre a rotina pré-definida (*built-in*) que faz isto.

1.4 Como se faz para concatenar as strings "bom" e "demais"?

1.2.3 – Obtenção de uma curva de permanência

Uma função distribuição acumulada (FDA) de probabilidade é uma função que nos informa qual é a probabilidade de que uma variável aleatória Q assuma um valor menor ou igual que um certo nível q . Os valores de Q variam de experimento para experimento. Por exemplo, se Q é a vazão máxima diária em um rio em um ano qualquer, o valor observado de Q varia de ano para ano.

A tabela 1.1 dá os valores da vazão máxima anual para o Rio dos Patos, PR, estação ANA (Agência Nacional de Águas do Brasil) 64620000, entre 1931 e 1999.

Provavelmente, a maneira mais simples de se *estimar* uma FDA a partir de um conjunto de dados é supor que os dados representam a totalidade das possibilidades, e que as observações são equiprováveis (em analogia com os 6 únicos resultados possíveis do lançamento de um dado não-viciado). No caso da tabela 1.1, se q_i é a vazão máxima do i -ésimo ano, teríamos que a probabilidade de ocorrência de q_i é

$$P\{Q = q_i\} = \frac{1}{n}, \quad (1.1)$$

Listagem 1.3: `patos-medmax.dat` — vazões média e máxima anuais, Rio dos Patos

1931	21.57	272.00
1932	25.65	278.00
1933	4.76	61.60
1934	11.46	178.30
1935	28.10	272.00

onde n é o número de observações. Mas a FDA *por definição* é

$$F(q) = P\{Q \leq q\}. \quad (1.2)$$

Para obtê-la, é preciso considerar os valores iguais ou menores que o valor de corte q . Portanto, nós devemos primeiro *ordenar* os q_i 's, de tal maneira que

$$q_0 \leq q_1 \leq \dots \leq q_{n-1}.$$

Note que a ordenação não altera (1.1). Após a ordenação, o cálculo de $F(q_i)$ é trivial:

$$F(q_i) = \sum_{k=0}^i \frac{1}{n} = \frac{i+1}{n}. \quad (1.3)$$

(1.3) é chamada de distribuição acumulada *empírica* de probabilidade. Em Hidrologia, muitas vezes (1.3) é denominada *curva de permanência*. O resultado $(i+1)/n$ é denominado uma *posição de plotagem*. Por diversos motivos, existem muitas outras posições de plotagem possíveis para a FDA empírica. Uma muito popular é $(i+1)/(n+1)$. A discussão detalhada de posições de plotagem deve ser feita em um curso de Probabilidade e Estatística, e não aqui, onde (1.3) serve (apenas) como um exemplo motivador.

Os dados da tabela 1.1 estão digitados no arquivo `patos-medmax.dat` (Apêndice A). Este arquivo contém 3 colunas contendo, respectivamente, o ano, a vazão média do ano, e a vazão máxima do ano. A listagem 1.3 mostra as 5 primeiras linhas do arquivo (que possui 69 linhas).

O programa `fqiemp.py`, mostrado na listagem 1.4, calcula a curva de permanência, ou FDA empírica, para as vazões máximas anuais do Rio dos Patos. Esta é, simplesmente, uma tabela de duas colunas: a vazão observada (em ordem crescente), e o valor de $(i+1)/n$.

Como antes, os comentários em `fqiemp.py` explicam muito do que está acontecendo. Mas há necessidade de explicações adicionais:

- 5 Faz a divisão funcionar como em Python 3.x. Em Python 2.x, `3/4 == 0`, e `3.0/4 == 0.75`. Em Python 3.x, `/` sempre produz um resultado do tipo `float` (ou `complex`); e um novo operador `//` sempre produz divisão inteira.
- 6 A função pré-definida `open` abre o arquivo `patos-medmax.dat`, descrito acima e exemplificado na listagem 1.3. O segundo argumento de `open`, a *string* `'rt'`, diz duas coisas: com `r`, que se trata de uma operação de *leitura*, (*read*), de um arquivo que já existe: isto garante que `patos-medmax.dat` não será modificado por `fqiemp.py`; com `t`, que se trata de um arquivo *texto*. Um arquivo texto é um arquivo formado por linhas, sendo cada linha uma *string*. As linhas são separadas por caracteres (invisíveis no editor de texto) de fim de linha, que convencionalmente nós indicamos por `\n`. Um arquivo texto é um objeto *iterável*, que pode ser acessado linha a linha.

Listagem 1.4: fqiemp.py — cálculo de uma FDA empírica

```

1 #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  from __future__ import division
6  fin = open('patos-medmax.dat','rt')      # abre o arquivo de dados (entrada)
7  qmax = []                               # uma lista vazia
8  for linha in fin:                       # loop nas linhas do arquivo
9      campo = linha.split()              # separa os campos
10     print(campo)                        # para ver campo a campo na tela
11     qi = float(campo[2])                 # a vazão é o terceiro campo
12     qmax.append(qi)                     # adiciona qi à lista qmax
13 fin.close()                             # fecha o arquivo de entrada
14 fou = open('fqiemp.dat','wt')           # abre o arquivo de saída
15 qmax.sort()                             # ordena a lista
16 n = len(qmax)                           # tamanho da lista
17 for i in range(n):                      # loop nos elementos da lista ordenada
18     qi = qmax[i]                         # vazão
19     Fi = (i+1)/n                         # posição de plotagem
20     fou.write('□8.2f□8.6f\n' % (qi,Fi)) # imprime uma linha
21 fou.close()                             # fim de papo

```

7 Nesta linha, `qmax` é inicializado como uma *lista* vazia. Uma lista é uma sequência de objetos quaisquer. Dada uma lista `a`, seus elementos são `a[0]`, `a[1]`, etc.. Uma lista `a` com n objetos vai de `a[0]` até `a[n-1]`.

8 Este é o *loop* de leitura do programa. `linha` é uma *string* que contém em cada iteração uma das linhas de `patos-medmax.dat`.

9 É preciso separar uma linha (veja a listagem 1.3) em seus 3 campos. O *método*¹ `split` separa a string `linha` em 3 strings, e as coloca em uma lista `campo == [campo[0], campo[1], campo[2]]`, usando os espaços em branco como separadores (que são eliminados).

10 Cada um dos `campo`'s agora é ele mesmo uma *string*, com os valores separados. As 5 primeiras linhas que aparecem na tela devido ao comando `print` da linha 10 são:

```

['1931', '21.57', '272.00']
['1932', '25.65', '278.00']
['1933', '4.76', '61.60']
['1934', '11.46', '178.30']
['1935', '28.10', '272.00']

```

11 No entato, estes campos ainda são *strings*, e não *float*'s. Aqui o terceiro campo é convertido em um *float* e vai para a variável `qi`.

12 Finalmente, `qi` é incluída na lista `qmax`.

13 É bom estilo de programação fechar cada arquivo previamente aberto.

14 Agora o programa abre o arquivo de saída, `fqiemp.dat`. `w` significa abertura para escrita (*write*); e `t` que o arquivo será texto.

¹Em Python, um método é uma rotina que “pertence” a uma variável ou um tipo (a rigor, a uma classe, mas este não é um curso de programação)

Listagem 1.5: `fqiemp.dat` — FDA empírica da vazão máxima anual no Rio dos Patos.

```
56.10 0.014493
61.60 0.028986
68.40 0.043478
75.40 0.057971
92.75 0.072464
```

- 15 `sort` é um *método* pré-definido para qualquer lista, que a ordena (por *default* em ordem crescente).
- 16 `len` é uma função pré-definida que retorna o número de elementos da lista.
- 17 *loop* para impressão no arquivo de saída.
- 18 Obtém o *i*-ésimo elemento de `qmax`, colocado na variável `qi`, que é re-utilizada para este fim.
- 19 Calcula a posição de plotagem, utilizando o operador `/` para dividir dois `int`'s e gerar um resultado correto do tipo `float` (veja comentário sobre a linha 5).
- 20 `write` é um método do arquivo `fou`. `write` sempre escreve seu único argumento, que *tem* que ser uma *string*, no arquivo. Trata-se portanto do problema inverso do *loop* de leitura, que transformava *strings* em `float`'s: agora precisamos transformar `float`'s em uma *string*. É para isto que serve o operador `%`: ele tem à sua esquerda uma *string* com os campos de formatação especiais `%8.2f` e `%8.6f`; à sua direita uma *tupla*² (`qi,Fi`) com tantos elementos quantos são os campos de formatação. O primeiro elemento será substituído no primeiro campo de formatação por uma *string* com 8 caracteres, sendo 2 deles para casas decimais. O segundo elemento será substituído no segundo campo de formatação por uma *string* com 8 caracteres, sendo 6 deles para casas decimais. A *string* resultante, que precisa conter explicitamente o caractere de fim de linha `\n`, será escrita no arquivo de saída.

As primeiras 5 linhas do arquivo de saída `fqiemp.dat` são mostradas na listagem 1.5.

Exercícios Propostos

1.5 Dado um número inteiro `p` lido do terminal, escreva um programa Python para procurar o índice `i` de uma lista `a` de 10 números inteiros definida internamente no programa tal que `a[i] == p`, e imprimir o resultado.

1.2.4 – Arquivos texto e arquivos binários

Arquivos texto são legíveis por seres humanos. Qualquer representação interna é primeiramente traduzida para uma *string* de caracteres antes de ser escrita em um arquivo texto. Arquivos binários em geral armazenam informação com a mesma representação interna utilizada pelo computador para fazer contas, etc..

²Em Python uma *tupla* é uma lista imutável

Listagem 1.6: `bintext.py` — Exemplo de arquivo texto e arquivo binário

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import print_function
4  i = 673451
5  bdet = bin(i)
6  print(bdet)
7  fot = open('i.txt', 'wt')
8  fot.write('%6d' % i)
9  fot.close()
10 from struct import pack
11 b = pack('i', i)
12 fob = open('i.bin', 'wb')
13 fob.write(b)
14 fob.close()
15 print(type(i))
16 print(type(b))
17 from os.path import getsize
18 print('tamanho do arquivo txt = %d bytes' % getsize('i.txt'))
19 print('tamanho do arquivo bin = %d bytes' % getsize('i.bin'))

```

Como sempre, um exemplo vale por mil palavras. Na listagem 1.6, temos o programa `bintext.py`, que produz dois arquivos: o arquivo texto `i.txt`, e o arquivo binário `i.bin`. Cada um destes arquivos contém o número inteiro `i == 673451`.

Eis aqui a dissecação de `bintext.py`:

- 5 A função pré-definida `bin` produz a representação do objeto na base 2.
- 8 Escreve a variável `i` no arquivo `i.txt`, usando um campo de 6 caracteres, que é o tamanho necessário para escrever `i` na base 10.
- 10 A única maneira de ler ou escrever um arquivo binário em Python é por meio de *byte strings*: strings em que cada elemento é um *byte* (8 bits). O módulo `struct` provê a conversão de, e para, *byte strings*.
- 11 Converte `i` em uma *byte string*. Como a representação interna de `i` é em 4 bytes, o tamanho de `b` será de 4 bytes.
- 15–16 Verifica os tipos de `i` e `b`.
- 18–19 Mostra o tamanho de cada um dos arquivos gerados.

Finalmente, a saída de `bintext.py` é

```

0b10100100011010101011
<type 'int'>
<type 'str'>
tamanho do arquivo txt = 6 bytes
tamanho do arquivo bin = 4 bytes

```

Conte os 32 bits na primeira linha (após o prefixo `0b`, que indica a representação na base 2): eles correspondem aos 4 *bytes* que custa ao programa para guardar o número inteiro 673451. Repare que o arquivo binário é menor que o arquivo texto. *Em geral*, arquivos binários tendem a ser menores (para a mesma quantidade de informação). A outra grande vantagem é que a leitura e a escritura de arquivos binários é muito mais *rápida*, porque não há necessidade de traduzir a informação de, e para, *strings*.

Listagem 1.7: `writearr.py` — Escreve um arquivo binário contendo 3 “linhas”, cada uma das quais com um *array* de 10 *float*’s.

```

1 #!/usr/bin/python
2 # -*- coding: iso-8859-1 -*-
3 from numpy.random import rand
4 fob = open('a.bin','wb')      # abre um arq binário para escrita
5 for k in range(3):           # loop em 3 "linhas"
6     a = rand(10)              # gera um array com 10 números aleatórios
7     a.tofile(fob)             # escreve uma "linha"
8 fob.close()

```

Na prática, arquivos binários estão invariavelmente associados ao uso de *arrays*, pelo menos em Engenharia. O módulo Numpy³, que *não vem com Python*, e *necessita ser instalado à parte*, proporciona um tipo chamado **array**, e permite manipular com boa eficiência computacional vetores e matrizes em Python. O tipo permite na prática substituir *listas* (tipo **list**); ademais, tudo ou quase tudo que funciona com listas também funciona com *arrays*. Neste texto, nós faremos a partir de agora amplo uso de Numpy e de seu tipo **array**. A referência completa de Numpy está disponível em domínio público (Oliphant, 2006), podendo também ser comprada pela Internet.

Além de definir *arrays*, Numpy também proporciona seus próprios métodos e funções para ler e escrever de e para arquivos binários. Vamos então dar dois exemplos muito simples, porém muito esclarecedores.

Primeiro, um programa para *escrever* um *array*. A listagem 1.7 mostra o programa `writearr.py`. O programa usa uma rotina disponível em **numpy**, `rand`, para devolver um *array* com 10 números aleatórios entre 0 e 1. `writearr.py` repete por 3 vezes a geração de **a** e a sua escritura no arquivo binário **a.bin**: a escritura utiliza o método `tofile`. Repare que `tofile` é um método do *array* **a**; ele não precisa ser importado, pois ele “já faz parte” da variável **a** a partir do momento em que ela é criada. `writearr` roda silenciosamente: não há nenhuma saída na tela. No entanto, se procurarmos no disco o arquivo gerado, teremos algo do tipo

```

>ls -l a.bin
-rw-r--r-- 1 nldias nldias 240 2011-08-28 14:08 a.bin

```

O arquivo gerado, **a.bin**, possui 240 bytes. Em cada uma das 3 iterações de `writearr.py`, ele escreve 10 *float*’s no arquivo. Cada *float* custa 8 *bytes*, de modo que em cada iteração 80 *bytes* são escritos. No final, são 240.

É importante observar que o arquivo **a.bin** *não possui estrutura*: ele “não sabe” que dentro dele mora o *array* **a**; ele é, apenas, uma “linguiça” de 240 *bytes*. Cabe a você, programadora ou programador, interpretar, ler e escrever corretamente o arquivo.

Prosseguimos agora para ler o arquivo binário gerado. Isto é feito com o programa `readarray.py`, mostrado na listagem 1.8.

O programa importa a rotina `fromfile` de Numpy, a partir da qual 3 instâncias de **a** são lidas do arquivo **a.bin** e impressas com formato na tela. Eis a sua saída:

```

0.2327 0.6117 0.7713 0.5942 0.1799 0.3156 0.1473 0.4299 0.0870 0.0846
0.4301 0.9779 0.0322 0.4833 0.6097 0.4387 0.0639 0.1399 0.4350 0.7737
0.5809 0.0382 0.6567 0.8062 0.8427 0.2511 0.2897 0.5785 0.2892 0.0385

```

³numpy.scipy.org

Listagem 1.8: `readarr.py` — Escreve um arquivo binário contendo 3 “linhas”, cada uma das quais com um *array* de 10 float’s.

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import print_function
4  from numpy import fromfile
5  from sys import stdout          # para escrever na tela
6  fib = open('a.bin','rb')        # abre o arquivo binário
7  for k in range(3):              # loop em 3 "linhas"
8      a = fromfile(fib,float,10)  # lê um array com 10 floats
9      for e in a:                  # imprime com formato na tela
10         stdout.write('_%5.4f' % e)
11         stdout.write('\n')
12 fib.close()

```

O que vemos são os números aleatórios das 3 instâncias de *a* escritas pelo programa `writearr.py`.

1.3 – Gnuplot

Mas como é a “cara” da função distribuição acumulada empírica? Nós gostaríamos de “ver” a função, isto é, plotá-la. É para isto que serve Gnuplot. Um programa Gnuplot especifica o arquivo de onde vêm os dados; que colunas queremos utilizar; que símbolos ou tipos de linhas, bem como espessuras ou tamanhos nós desejamos, e o nome do arquivo de saída (na falta de um nome para o arquivo de saída, uma tela com o gráfico deverá se abrir para visualização interativa). Via de regra, o arquivo de saída de um programa Gnuplot *não* é um arquivo texto, mas um arquivo com uma descrição gráfica qualquer. Extensões comuns de nomes de arquivos que indicam imagens são `.jpg`, `.png`, `.eps`, e `.emf` (esta última em Windows). Gnuplot permite gerar arquivos de imagem em qualquer um destes formatos (e muitos outros!). Neste texto, nós vamos sempre gerar pares de arquivos (*quase* idênticos) `.eps` e `.emf`. Em Linux, um arquivo `a.eps` é facilmente conversível no arquivo `a.pdf` mediante o comando

```
[ epstopdf a.eps ] ;
```

em Windows, um arquivo `a.emf` está pronto para ser visualizado (com um clique), inserido em documentos, etc..

Nós já temos a FDA empírica da vazão máxima anual no Rio dos Patos, no arquivo `fqiemp.dat`; agora, nós vamos plotá-la com o programa Gnuplot `fqiemp.plt`:

Como sempre, é preciso explicar:

- 5 Gera um arquivo de saída no padrão *Encapsulated Postscript*, em preto-e-branco, com tipo *Times Roman*, no tamanho 18pt.
- 6 O nome do arquivo de saída.
- 7 Gera um arquivo que mostra as convenções utilizadas neste “terminal”. Este arquivo pode ser visto na figura [1.1](#)
- 8 O nome do arquivo de saída.
- 9 Formato do eixo *y*, que usa a mesma convenção de Python (a bem da verdade, de C).

Listagem 1.9: `fqiemp.plt` — programa para plotar a FDA empírica do Rio dos Patos

```

1  set encoding iso_8859_1
2  # -----
3  # no mundo de linux
4  # -----
5  set terminal postscript eps monochrome 'Times-Roman' 18
6  set output 'test.eps'
7  test
8  set output 'fqiemp.eps'
9  set format y '%3.1f'
10 set grid
11 set xlabel 'Vazão_máxima_anual(m3/s)'
12 set ylabel 'FDA_empírica'
13 plot 'fqiemp.dat' using 1:2 notitle with points pt 7
14 # -----
15 # agora no mundo de Windows
16 # -----
17 set terminal emf monochrome 'Times_New_Roman' 18
18 set output 'test.emf'
19 test
20 set output 'fqiemp.emf'
21 replot
22 exit

```

- 10 Mostra o reticulado da escala.
- 11 Nome do eixo x .
- 12 Nome do eixo y .
- 13 Nome do arquivo de dados a ser utilizado; colunas para x e y ; use pontos, com símbolo 7 (veja a convenção na figura 1.1): gera o arquivo `fqiemp.eps`.
- 17 Mude o tipo de arquivo de saída para `emf`, usando o tipo *Times New Roman*, no tamanho 18 pt.
- 18 O nome do arquivo de saída.
- 19 O mesmo que a linha 7.
- 20 O nome do arquivo de saída.
- 21 Plota novamente, usando todas as definições que não tiverem sido mudadas (só mudamos o “terminal” de saída); agora, gera o arquivo `fqiemp.emf`.

Finalmente, podemos ver a FDA empírica gerada por `fqiemp.plt`, na figura 1.2.

Uma última observação de cunho estético. Neste texto, as figuras que são geradas por programas Gnuplot descritos explicitamente, e cujas listagens `.plt` são dadas no texto, utilizam o tipo *postscript Times-Roman*. Olhe bem para o tipo utilizado nas figuras 1.1 e 1.2: este *não* é o tipo utilizado no restante do texto (por exemplo, compare com as legendas).

Já as figuras geradas para fim ilustrativo utilizam o mesmo tipo que o texto; veja por exemplo as figuras 2.1, e ??.

Utilizar um tipo de letra diferente nas figuras e no texto é ligeiramente inconsistente, do ponto de vista tipográfico. Entretanto, a escolha de *Times-Roman* para

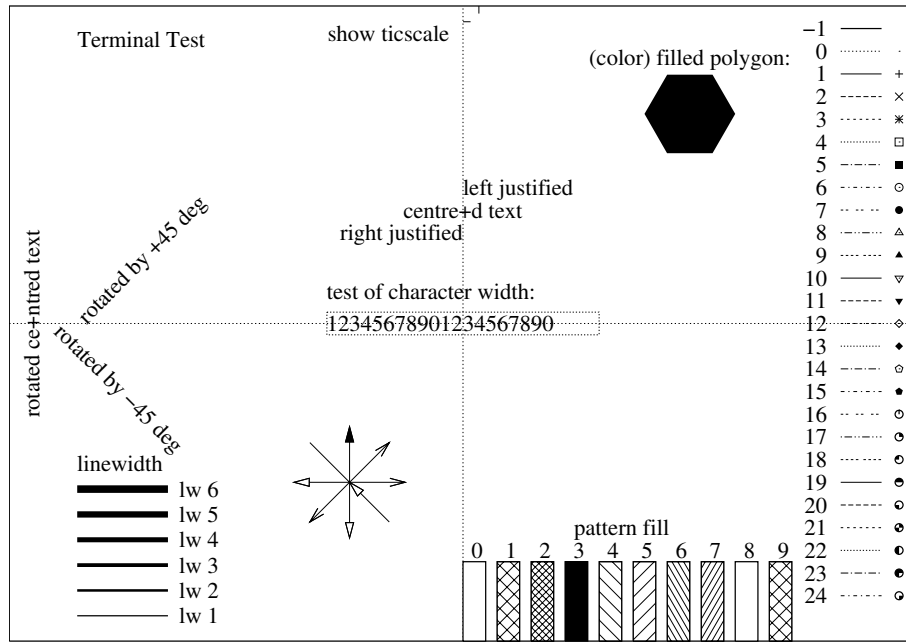


Figura 1.1: Convenções do terminal `postscript eps monochrome 'Times-Roman'` 18

a saída em *postscript*, e de *Times New Roman* para a saída em EMF, foi feita em nome da simplicidade e da universalidade. É praticamente impossível não existir o tipo *Times-Roman* em seu sistema operacional Linux, e praticamente impossível não existir o tipo *Times New Roman* em seu sistema operacional Windows. Na preparação de um texto mais bem-elaborado tipograficamente você deve, se possível, utilizar o mesmo tipo nas figuras e no texto. Mas isto, além de ser outra história, é do gosto do freguês.

1.3.1 – Ajuste de uma função

A definição da FDA empírica em (1.3) é muito limitada: é óbvio que nenhum destes 69 valores da tabela 1.1, ao contrário do que acontece no jogo de dados, jamais se repetirá. Também é óbvio que valores *maiores* do que máximo de $660 \text{ m}^3 \text{ s}^{-1}$ poderão ocorrer no futuro (em relação a 1999 ...), e isto é uma poderosa fonte de crítica à posição de plotagem ingênua que nós utilizamos em (1.3): ela prevê que a probabilidade de uma vazão máxima anual maior do que $660 \text{ m}^3 \text{ s}^{-1}$ é nula!

Em suma, é nossa esperança que uma relação analítica, uma *fórmula*, faça um trabalho no mínimo um pouco melhor do que a FDA empírica foi capaz de fazer para *prever* as probabilidades da vazão máxima anual no Rio dos Patos — mas espere para ver o resultado!

Vamos tentar *ajustar* uma fórmula aos dados. A fórmula que escolheremos é a da distribuição de Weibull,

$$F(q) = 1 - \exp\left(-(q/\lambda)^k\right). \quad (1.4)$$

Não existe *absolutamente* nenhum motivo transcendental para utilizar (1.4): ela apenas atende aos requisitos fundamentais de uma FDA (tende a 0 em $x \rightarrow -\infty$; e a 1 em $x \rightarrow +\infty$). A rigor, a Weibull é definida apenas para $q \geq 0$, que é o que se espera de qualquer vazão em rios — nenhum rio tem vazão negativa; nenhum rio corre do mar para suas cabeceiras.

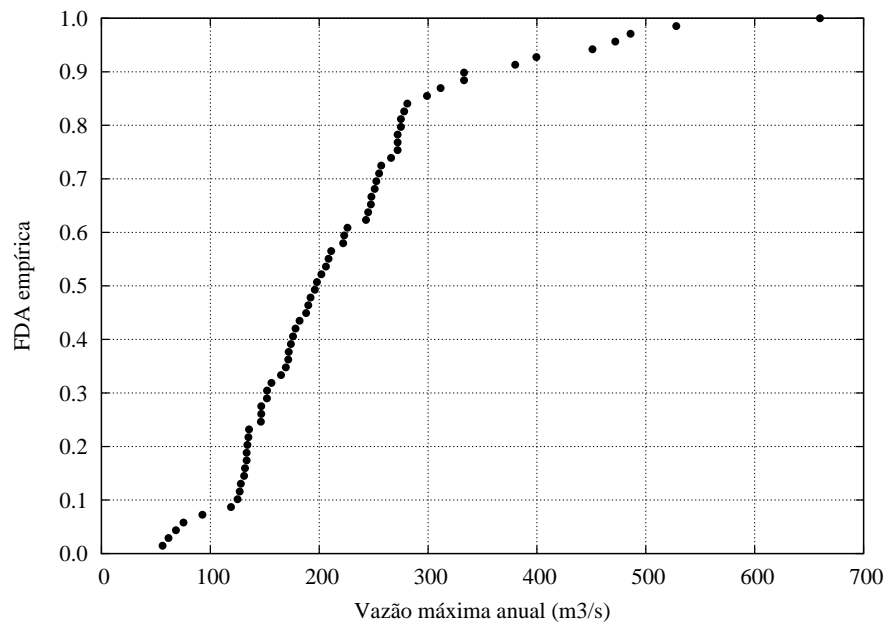


Figura 1.2: FDA da vazão máxima anual, Rio dos Patos

Listagem 1.10: `weibull.plt` — como ajustar uma FDA analítica (Weibull) aos dados da FDA empírica

```

1  set encoding iso_8859_1
2  # -----
3  # no mundo de linux
4  # -----
5  set terminal postscript eps monochrome 'Times-Roman' 18
6  # -----
7  # ajusta lambda e k
8  # -----
9  lambda = 1000.0
10 k = 1.0
11 F(x) = 1.0 - exp(-((x/lambda)**k))
12 fit F(x) 'fqieimp.dat' using 1:2 via lambda,k
13 set output 'weibullfit.eps'
14 set format y '%3.1f'
15 set grid
16 set xlabel 'Vazão máxima anual (m3/s)'
17 set ylabel 'FDA'
18 set xrange [0:1000]
19 set yrange [0:1]
20 set xtics 0,100
21 set ytics 0,0.1
22 set samples 10000
23 plot 'fqieimp.dat' using 1:2 notitle with points pt 7, \
24      F(x) with lines notitle lt 1 lw 3
25 # -----
26 # agora no mundo de Windows
27 # -----
28 set terminal emf monochrome 'Times New Roman' 18
29 set output 'weibullfit.emf'
30 replot
31 exit

```

No programa `weibullfit.plt`, mostrado na listagem 1.10, veja como Gnuplot lida facilmente com o ajuste dos valores de λ e de k aos dados de que dispomos. Gnuplot produz informações na tela sobre o ajuste requisitado pelo comando `fit`; as últimas linhas da saída em tela são

```
After 11 iterations the fit converged.
final sum of squares of residuals : 0.0604353
rel. change during last iteration : -6.18025e-06

degrees of freedom      (FIT_NDF)                : 67
rms of residuals        (FIT_STDFIT) = sqrt(WSSR/ndf) : 0.0300336
variance of residuals   (reduced chisquare) = WSSR/ndf : 0.000902019

Final set of parameters          Asymptotic Standard Error
=====
lambda      = 233.486            +/- 1.283            (0.5495%)
k           = 2.62023            +/- 0.05594           (2.135%)

correlation matrix of the fit parameters:

                lambda k
lambda      1.000
k          -0.346  1.000
```

donde $\lambda = 233,486$, $k = 2,62023$. Na listagem 1.10, comentários adicionais são:

- 9–10 Os valores iniciais de `lambda` e `k` *influenciam* nos valores finais encontrados — ou não. Por exemplo, se mudarmos a linha 9 para `[lambda = 1.0]`, o comando `fit` da linha 12 *não consegue ajustar a curva* $F(x)$; muitas vezes você precisará fazer várias tentativas com os valores iniciais até ter sucesso.
- 11 É possível definir funções que serão depois plotadas.
- 22 É possível aumentar o número de pontos utilizados para traçar curvas, quando a aparência das mesmas for “quebrada”.
- 23 É possível aumentar a espessura das curvas, com `lw` (*line width*): o número a seguir especifica a espessura desejada: veja a figura 1.1.

A figura 1.3 mostra o gráfico gerado: note como a cauda da direita, ou seja, os valores maiores de vazão máxima, são *mal ajustados*: $F(q)$ *subestima* o valor de q para probabilidades altas (próximas de 1): o uso da Weibull para este conjunto de dados seria desastroso, uma vez que as maiores cheias seriam subestimadas!

Exercícios Propostos

1.6 Usando Gnuplot, plote em tons de cinza a área debaixo da curva $y = x$ para $x = 2$. Quanto vale esta área?

1.4 – Python de novo: projeto probabilístico

De fato, um *projeto probabilístico* é a definição de um valor de corte q cuja probabilidade de excedência seja menor que um certo risco predeterminado α . Suponha que, para o Rio dos Patos, nós desejemos calcular o valor q da vazão máxima anual

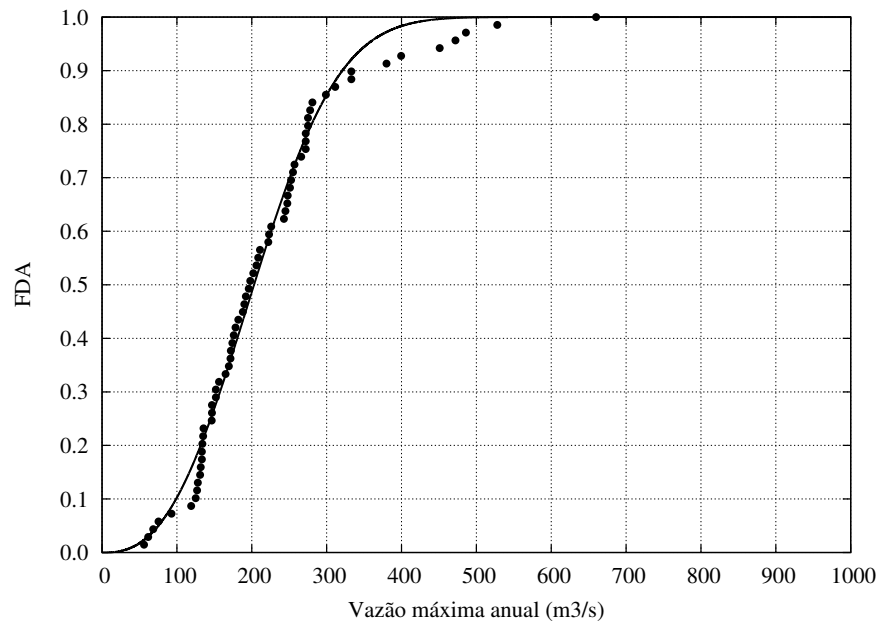


Figura 1.3: Ajuste de uma FDA Weibull aos dados de vazão máxima anual do Rio dos Patos

cuja probabilidade de excedência em um ano qualquer seja $\alpha = 0.05$ (5%). Isto significa que precisamos resolver a equação

$$\begin{aligned} P\{Q > q\} &= 1 - P\{Q \leq q\} = \alpha, \\ 1 - F(q) &= \alpha, \\ F(q) &= 1 - \alpha, \end{aligned} \tag{1.5}$$

para q .

Quando a $F(q)$ é a FDA empírica, não existe uma equação para se resolver, mas sim uma tabela na qual devemos interpolar o valor desejado de q . Primeiramente nós precisamos escrever uma *função* que, dado um valor `xc`, procure o intervalo em que ele se encontra na coluna `x` da tabela, e em seguida calcule o `yc` correspondente interpolando linearmente entre os valores da coluna `y` para o mesmo intervalo. O *módulo* `interp`, arquivo `interp.py`, mostrado na listagem 1.11, faz isto.

`interp` não é um programa: ele é um módulo que contém uma única função (que possui também o nome `interp`). Esta rotina estará disponível para qualquer programa Python por meio de uma declaração do tipo

```
[from interp import interp].
```

A dissecação do módulo e da função `interp` é

6 Em Python, uma função é definida com a palavra reservada `def`.

9 `assert` significa “assegure-se de que”; `==` é um operador lógico que retorna `True` se os operandos forem iguais.

17–18 `iu` é para “upper”, `il` é para “lower”: são os índices do intervalo que contém `xc`. O intervalo se inicia igual ao intervalo da tabela toda.

Listagem 1.11: `interp.py` — módulo com função para calcular uma interpolação linear.

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  from __future__ import division
6  def interp(xc, x, y):
7      nx = len(x)
8      ny = len(y)
9      assert nx == ny          # x e y devem ser listas com tamanhos iguais
10     if nx < 2:                # deve haver pelo menos 2 pontos
11         exit("interp: n = %d deve ser >= 2\n" % nx)
12     if (xc < x[0]) or (xc > x[nx-1]): # xc deve estar no intervalo da lista x
13         exit("xc = %lf fora de alcance: %f\n" % (xc, x[0], x[nx-1]))
14     # -----
15     # inicia uma busca binária
16     # -----
17     iu = nx - 1
18     il = 0
19     while (iu - il > 1):
20         im = (iu + il)//2
21         if (xc <= x[im]):
22             iu = im
23         else:
24             il = im
25     # -----
26     # interpola linearmente, e retorna
27     # -----
28     dx = (x[iu] - x[il])      # delta x
29     dy = (y[iu] - y[il])      # delta y
30     m = dy/dx                 # coeficiente angular
31     return (y[il] + m * (xc - x[il])) # y interpolado

```

19–24 Este é um algoritmo de *busca binária*: o índice `iu` vai diminuindo, ou o índice `il` vai aumentando, até que a diferença entre os dois seja igual a 1. Estude-o, e compreenda-o.

28–31 Uma interpolação linear padrão. Lembre-se da equação da reta em Geometria Analítica.

Agora que temos uma função de interpolação linear, podemos escrever um programa que a utilize. Este programa, `ppbemp.py` (“projeto probabilístico a partir da distribuição empírica”) mostrado na listagem 1.12, simplesmente lê a tabela com a FDA empírica (que, como sabemos, está no arquivo `fqiemp.dat`), e interpola um valor `qproj` nesta tabela. O programa é muito óbvio, e não necessita de maiores explicações.

A vazão com risco de 5% de excedência calculada é $q_{20} = 462,55 \text{ m}^3 \text{ s}^{-1}$. Como $5\% = 1/20$, é comum em Hidrologia (e em outras disciplinas em que se utiliza critérios de risco) dizer que q_{20} é a “vazão de 20 anos de tempo de recorrência”.

Para projetar a partir da $F(x)$ analítica (isto é, da Weibull com os valores de k e de λ encontrados anteriormente), basta resolver (1.5) analiticamente:

Listagem 1.12: ppbemp.py — projeto probabilístico a partir da distribuição empírica

```

1 #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  from __future__ import division
6  fin = open('fqimp.dat', 'rt')          # abre o arquivo de dados (entrada)
7  qq = []                               # tabela inicialmente vazia
8  FF = []                               # tabela inicialmente vazia
9  for linha in fin:                     # loop nas linhas do arquivo
10     campo = linha.split()             # separa os campos
11     (qi, Fi) = (float(campo[0]), float(campo[1]))
12     qq.append(qi)
13     FF.append(Fi)
14  fin.close()                          # fecha o arquivo de entrada
15  from interp import interp            # importa interp
16  qproj = interp(0.95, FF, qq)         # projeto para risco de exc. de 5%
17  print("qproj = %8.2f\n" % qproj)     # fim de papo

```

$$\begin{aligned}
 F(q_{20}) &= 0,95, \\
 1 - \exp\left(-(q_{20}/\lambda)^k\right) &= 0,95, \\
 \exp\left(-(q_{20}/\lambda)^k\right) &= 0,05, \\
 -(q_{20}/233,486)^{2,62023} &= -2.99573, \\
 q_{20}/233,486 &= 2.99573^{1/2,62023} = 1.52004, \\
 q_{20} &= 354.908 \blacksquare
 \end{aligned}$$

Observe que este é um valor consideravelmente menor que o da própria distribuição empírica! Como já havíamos comentado baseados na análise visual da figura 1.3, a distribuição ajustada *subestima* as vazões com tempos de recorrência altos, e não deve ser utilizada para projetos probabilísticos: às vezes uma tabela de dados usada corretamente é melhor que um modelo matemático ajustado com ferramentas computacionais sofisticadas!

Além disso, o método que utilizamos para ajustar a FDA da Weibull, usando o comando `fit` de Gnuplot, é incomum: em Probabilidade e Estatística, existem outros métodos para o ajuste de distribuições de probabilidade, notadamente o *método dos momentos* e o *método da máxima verossimilhança*. O uso de outros métodos de ajuste, e de outras distribuições de probabilidade, provavelmente produziria resultados muito melhores.

Exercícios Propostos

1.7 Usando Gnuplot (é claro), plote a vazão média anual do Rio dos Patos em função do tempo.

1.8 Com Gnuplot, ajuste uma reta às vazões médias em função do tempo no Rio dos Patos, e plote novamente. Você acha que a vazão média do Rio dos Patos está mudando com o tempo?

1.5 – Maxima

Maxima é a linguagem de processamento simbólico mais antiga que existe: ela se chamava, quando foi criada, MACSYMA. A mudança de nome ocorreu quando o código se tornou livre. Maxima é capaz de fazer álgebra, derivar, integrar, resolver equações diferenciais, calcular transformadas de Laplace, etc., *analiticamente*. Por exemplo: você sabe quanto é $(a+b)^4$? Não? Digitando **maxima** na linha de comando você obterá

```
Maxima 5.21.1 http://maxima.sourceforge.net
using Lisp GNU Common Lisp (GCL) GCL 2.6.7 (a.k.a. GCL)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1)
```

Continuando a digitar, você encontrará:

```
(%i1) (a+b)^4 ;

(%o1)
              4
          (b + a)

(%i2) expand(%);

(%o2)
      4      3      2 2      3      4
b  + 4 a b  + 6 a  b  + 4 a  b + a

(%i3) quit();
```

O comando

`[quit();]`

o devolverá para a linha de comando.

Vamos fazer algo um pouco mais sofisticado: suponha que você deseje calcular a *média* de população da distribuição Weibull, que nós encontramos na seção anterior, em função de seus dois *parâmetros* λ e k . Por definição, a média de uma distribuição cuja FDA é $F(x)$ é

$$\mu = \int_{x \in \mathcal{D}} x \frac{dF}{dx} dx, \quad (1.6)$$

onde \mathcal{D} é o domínio de validade da variável aleatória. No caso da Weibull, $x \geq 0$, donde

$$\mu = \int_0^\infty x \frac{d}{dx} [1 - \exp(-(x/\lambda)^k)] dx. \quad (1.7)$$

A derivada ficou intencionalmente indicada: estou supondo que nós somos preguiçosos, e não queremos calculá-la manualmente, nem à integral. A listagem 1.13 mostra uma maneira de calcular μ , em função de λ e de k , com Maxima.

As explicações se seguem:

- 1–2 Declara que **k** e **lam** são variáveis reais. O símbolo **\$** no fim de cada linha omite a “resposta” de Maxima; isto é útil quando executamos uma série de comandos de Maxima em “batch” (em série) a partir de um arquivo **.max**, e não estamos interessados em ver o resultado de cada um deles. Em seções interativas, o normal é encerrar cada comando com ‘;’.
- 3–4 Maxima suporá que **k** e **lam** são positivos em seus cálculos.
- 5 Sem a linha 5, Maxima vai parar e perguntar se **k** é **integer** — apesar da linha 1!

Listagem 1.13: `mulambdak.max` — cálculo da média da distribuição Weibull em em função de seus parâmetros λ e k

```

1 declare ( [k], real)$
2 declare ( [lam], real)$
3 assume ( k > 0)$
4 assume ( lam > 0)$
5 declare ( [k], noninteger)$
6 F : 1 - exp(-(x/lam)^k)$
7 fdp : diff(F,x)$
8 mu : integrate(x*fdp,x,0,inf) ;

```

6 Define F ($F(x)$): note que não é $F(x)$! Note também que a *atribuição*, em Maxima, *não* é feita com ‘=’ (como em Python ou Gnuplot), mas sim com ‘:=’; ‘=’ fica reservado para igualdade.

7 armazena a derivada de $F(x)$, que em probabilidade se chama *função densidade de probabilidade*, na variável `fdp`.

8 Calcula a integral definidora da média.

O resultado de “rodar” `mulambdak.max`, com o comando

`[maxima -b mulambdak.max]`

é

```

Maxima 5.21.1 http://maxima.sourceforge.net
using Lisp GNU Common Lisp (GCL) GCL 2.6.7 (a.k.a. GCL)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1)                                     batch(mulambdak.max)

read and interpret file: #p/home/nldias/work/graduacao/matap/aulas/mulambdak.max
(%i2)                                     declare([k], real)
(%i3)                                     declare([lam], real)
(%i4)                                     assume(k > 0)
(%i5)                                     assume(lam > 0)
(%i6)                                     declare([k], noninteger)
                                     x      k
(%i7)                                     F : 1 - exp(- (---) )
                                     lam
(%i8)                                     fdp : diff(F, x)
(%i9)                                     mu : integrate(x fdp, x, 0, inf)
                                     k + 1
(%o9)                                     gamma(-----) lam
                                     k
(%o9)                                     mulambdak.max

```

Na linha (%o9), `gamma` significa a função gama $\Gamma(x)$, que nós vamos encontrar muitas vezes neste curso, mas que não vem ao caso detalhar agora. O resultado analítico que nós obtivemos com Maxima é

$$\mu = \lambda \Gamma\left(\frac{k+1}{k}\right). \quad (1.8)$$

Exercícios Propostos

1.9 Com Maxima, calcule a derivada de

$$f(x) = \ln(\sin(e^x)).$$

1.10 Com Maxima, calcule

$$\int_1^{\infty} \frac{1}{x^{3/2}} dx.$$

1.11 Com Maxima, obtenha as raízes de

$$x^3 + \frac{3}{2}x^2 - \frac{29}{2}x + 15 = 0.$$

2

Um pouco de polinômios, integrais, séries . . .

2.1 – Integração numérica: motivação

Suponha que você deseje traçar uma curva com as seguintes propriedades:

- passar pelos pontos $(0, 0)$, $(1, 5/2)$, $(2, 7/2)$ e $(4, 4)$;
- possuir derivada igual a 0 em $x = 4$.

Existem muitas curvas com estas propriedades, mas uma candidata natural é um polinômio de grau 5, uma vez que existem 5 propriedades — ou graus de liberdade — na lista acima. Portanto,

$$\begin{aligned}f(x) &= ax^4 + bx^3 + cx^2 + dx + e, \\g(x) &= \frac{df}{dx}, \\f(0) &= 0, \\f(1) &= 5/2, \\f(2) &= 7/2, \\f(4) &= 4, \\g(4) &= 0.\end{aligned}$$

Agora, podemos obter facilmente a, b, c, d, e com Maxima, com o programa `achapol.max` da listagem 2.1.

A dissecação de `achapol.max` é a seguinte:

- 1–2 Define $f(x)$ e $g(x)$.
- 3–7 Define o valor de $f(x)$ em $x = 0, 1, 2, 4$, e o valor de $g(x)$ em $x = 4$, em função de a, b, c, d, e .
- 8 Resolve um sistema linear 5×5 em a, b, c, d, e .

A saída de `achapol.max` é mostrada na listagem 2.2

Portanto,

$$f(x) = -\frac{1}{48}x^4 + \frac{13}{48}x^3 - \frac{17}{12}x^2 + \frac{11}{3}x. \quad (2.1)$$

Listagem 2.1: achapol.max — Polinômio com propriedades definidas

```

1 f : a*x^4 + b*x^3 + c*x^2 + d*x + e ;
2 g : diff(f,x);
3 eq1 : f,x=0 ;
4 eq2 : f,x=1 ;
5 eq3 : f,x=2 ;
6 eq4 : f,x=4 ;
7 eq5 : g, x=4 ;
8 solve ([eq1 = 0, eq2 = 5/2, eq3 = 7/2, eq4 = 4, eq5 = 0], [a,b,c,d,e]) ;

```

Listagem 2.2: Saída de achapol.max

```

Maxima 5.21.1 http://maxima.sourceforge.net
using Lisp GNU Common Lisp (GCL) GCL 2.6.7 (a.k.a. GCL)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1)                                     batch(achapol.max)

read and interpret file: #p/home/nldias/work/graduacao/matap/aulas/achapol.max

(%i2)                                     f : e + d x + c x + b x + a x
                                     4       3       2       4
(%o2)                                     a x + b x + c x + d x + e
(%i3)                                     g : diff(f, x)
                                     3       2
(%o3)                                     4 a x + 3 b x + 2 c x + d
(%i4)                                     ev(eq1 : f, x = 0)
(%o4)                                     e
(%i5)                                     ev(eq2 : f, x = 1)
(%o5)                                     e + d + c + b + a
(%i6)                                     ev(eq3 : f, x = 2)
(%o6)                                     e + 2 d + 4 c + 8 b + 16 a
(%i7)                                     ev(eq4 : f, x = 4)
(%o7)                                     e + 4 d + 16 c + 64 b + 256 a
(%i8)                                     ev(eq5 : g, x = 4)
(%o8)                                     d + 8 c + 48 b + 256 a
(%i9) solve([eq1 = 0, eq2 = -, eq3 = -, eq4 = 4, eq5 = 0], [a, b, c, d, e])
                                     1       13       17       11
(%o9) [[a = - --, b = --, c = - --, d = --, e = 0]]
                                     48       48       12       3
(%o9)                                     achapol.max

```

Suponha agora que você deseje calcular

$$I = \int_1^5 f(x) dx.$$

É claro que esta integral pode ser calculada analiticamente com Maxima:

```

(%i1) [ a : -1/48, b : 13/48, c : -17/12, d : 11/3] ;
(%o1) [- --, --, - --, --]
          48 48 12 3
(%i2) f : a*x^4 + b*x^3 + c*x^2 + d*x ;
          4       3       2
(%o2)   x  13 x  17 x  11 x
        - -- + ---- - ---- + ----
        48  48  12  3
(%i3) integrate(f,x,1,5) ;
          1321
(%o3)   ----
          90

```

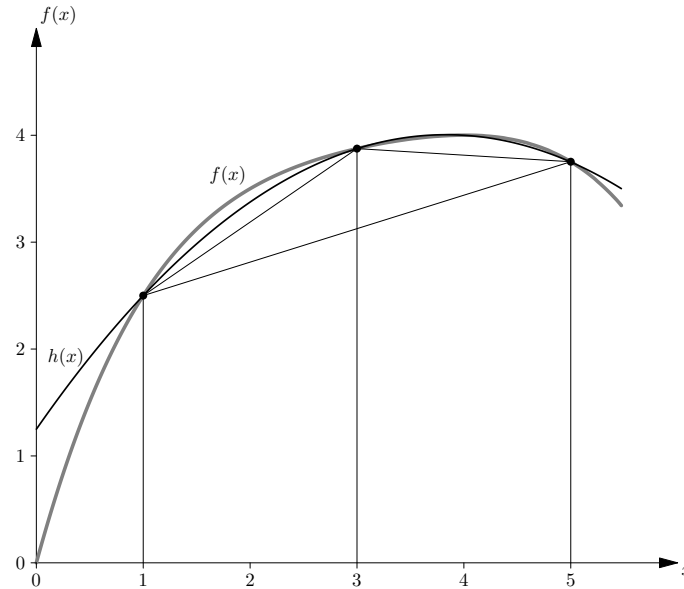


Figura 2.1: Integração numérica de uma função

```
(%i4) bfloat(%);
(%o4) 1.467777777777778b1
```

Logo, $I = 1321/90 \approx 14,6778$ (atenção para o arredondamento). Entretanto, nem todas as integrais podem ser calculadas assim, por uma série de motivos:

- a função pode não possuir uma integral em forma fechada,
- a função pode ser definida por pontos, mas não por uma fórmula,
- a função pode ser difícil de integrar analiticamente, e você pode querer ter uma idéia inicial do valor da integral,

etc..

Suponha portanto que você *não soubesse* que $I = 1321/90$, mas fosse capaz de calcular em princípio tantos pontos de $f(x)$ quantos fossem necessários: qual a sua aposta para o valor de I ?

Considere a figura 2.1: uma das aproximações mais óbvias — mas também mais grosseiras — é substituir a função por uma reta ligando os pontos $(1, f(1))$ e $(5, f(5))$. A área do *trapézio* é a nossa primeira aproximação para a integral:

$$I_0 = \frac{f(1) + f(5)}{2} \times 4 = 12,50.$$

Nada mal, considerando o valor verdadeiro 14,6778! Mas pode ser melhorada, com o uso de dois trapézios. Para isto, basta calcular $f(3)$ e somar as áreas dos dois trapézios resultantes:

$$I_1 = \frac{f(1) + f(3)}{2} \times 2 + \frac{f(3) + f(5)}{2} \times 2 = 14,000.$$

Note que estamos *muito* próximos do valor verdadeiro — com apenas 2 trapézios. Mas existe uma alternativa analítica mais inteligente do que trapézios: como temos 3 pontos, nós podemos aproximar a curva por uma parábola do 2º grau passando

por estes 3 pontos. O programa `passaquad.max`, mostrado na listagem 2.3, faz este trabalho, e acha os coeficientes a, b, c da parábola

$$h(x) = ax^2 + bx + c$$

que passa por $(1, f(1))$, $(3, f(3))$ e $(5, f(5))$. No embalo, `passaquad.max` redefine $h(x)$ com os coeficientes encontrados, e já calcula a integral de $h(x)$ entre 1 e 5. A parábola h também é mostrada na figura 2.1.

Listagem 2.3: `passaquad.max` — parábola $h(x) = ax^2 + bx + c$ passando por $(1, f(1))$, $(3, f(3))$ e $(5, f(5))$.

```

1 f : (-1/48)*x^4 + (13/48)*x^3 - (17/12)*x^2 + (11/3)*x ;
2 y1 : f,x=1$
3 y3 : f,x=3$
4 y5 : f,x=5$
5 h : a*x^2 + b*x + c$
6 eq1 : ev(h,x=1) = y1 ;
7 eq2 : ev(h,x=3) = y3 ;
8 eq3 : ev(h,x=5) = y5 ;
9 solve( [eq1, eq2, eq3],[a,b,c]) ;
10 abc : % ;
11 h : h,abc ;
12 integrate(h,x,1,5);

```

A saída de `passaquad.max` é mostrada na listagem 2.4.

Com os coeficientes a, b, c de $h(x)$, nós podemos calcular — analiticamente — nossa próxima aproximação:

$$\begin{aligned}
 I_2 &= \int_1^2 h(x) dx \\
 &= \int_1^2 \left[-\frac{3}{16}x^2 + \frac{23}{16}x + \frac{5}{4} \right] dx \\
 &= \frac{29}{2} = 14,5000.
 \end{aligned}$$

Até agora nós avaliamos 3 alternativas de integração numérica de $f(x)$: com um trapézio, com dois trapézios, e com uma parábola. A tabela 2.1 dá um resumo dos resultados alcançados. O erro relativo de cada estimativa é

$$\delta = \frac{I_k - I}{I}. \quad (2.2)$$

Uma única parábola foi capaz de estimar I com um erro relativo ligeiramente superior a 1%. Um caminho geral para a integração numérica está aberto: aumentar o número de “elementos” de integração (no nosso caso foram trapézios) e/ou aumentar a ordem do polinômio aproximador da função por um certo número de pontos. Este é o conteúdo da próxima seção.

Exercícios Propostos

2.1 Se $f(x) = \sin x$, qual é a estimativa de $I = \int_0^\pi f(x) dx = 2$ com *um* trapézio ?

2.2 Provavelmente, você não está muito contente com o resultado do Problema 2.1.

Listagem 2.4: Saída de `passaquad.max`

```

batching /home/nldias/work/graduacao/matap/aulas/passaquad.max

      2      3      4
      11 x    17 x    13 x    (- 1) x
(%i2)  f : ---- - ---- + ---- + ----
          3      12      48      48

          4      3      2
          x    13 x    17 x    11 x
(%o2)  - -- + ---- - ---- + ----
        48      48      12      3

(%i3)  ev(y1 : f, x = 1)
(%i4)  ev(y3 : f, x = 3)
(%i5)  ev(y5 : f, x = 5)

          2
          h : c + b x + a x
(%i6)  eq1 : ev(h, x = 1) = y1
          5
          c + b + a = -
(%o7)
          2
          eq2 : ev(h, x = 3) = y3
          31
          c + 3 b + 9 a = --
(%o8)
          8
          eq3 : ev(h, x = 5) = y5
          15
          c + 5 b + 25 a = --
(%o9)
          4
solve([eq1, eq2, eq3], [a, b, c])
(%i10)
          3      23      5
          [[a = - --, b = --, c = -]]
          16      16      4
(%i11)  abc : %
          3      23      5
(%o11)  [[a = - --, b = --, c = -]]
          16      16      4
(%i12)  ev(h : h, abc)
          2
          3 x    23 x    5
(%o12)  - ---- + ---- + -
          16      16      4
(%i13)  integrate(h, x, 1, 5)
          29
          --
          2
(%o13)  passaquad.max

```

- a) Aproxime a integral I do Problema 2.1 com dois trapézios, entre $x = 0$ e $\pi/2$, e entre $x = \pi/2$ e π .
- b) Aproxime a integral pela integral da parábola quadrática $g(x) = ax^2 + bx + c$ passando pelos pontos $(0, 0)$, $(\pi/2, 1)$ e $(\pi, 0)$.
- c) Aproxime a integral de $f(x)$ pela integral da parábola cúbica $h(x) = ax^3 + bx^2 + cx + d$ passando pelos mesmos pontos acima, e com $h'(\pi/2) = 0$.

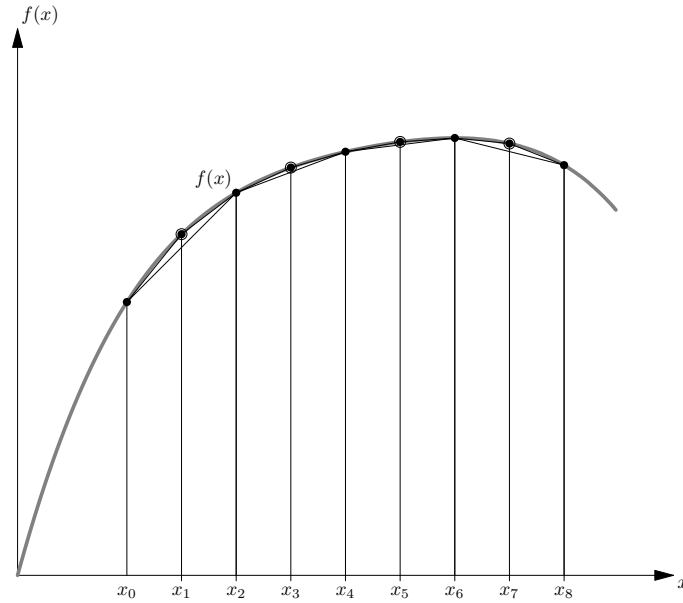
2.3 Com Gnuplot, plote $f(x) = \sin x$ e a aproximação $g(x)$ obtida no Problema 2.2.

2.2 – A regra do trapézio

Vamos começar a melhorar nossas estimativas de I pelo método de “força bruta”, de aumentar o número de trapézios. Isto nos levará ao método talvez mais simples de integração numérica que vale a pena mencionar, denominado “Regra do Trapézio”.

Tabela 2.1: Estimativas numéricas de I e seus erros relativos δ .

Integral	Valor	δ
Exato	14,6778	0
Um trapézio	12,5000	0,1483
Dois trapézios	14,0000	0,0461
Uma parábola	14,5000	0,0121

Figura 2.2: A regra do trapézio, com $n = 4$ e $n = 8$ trapézios.

A figura 2.2 mostra a mesma função $f(x)$ da seção anterior; agora, entretanto, nós desenhamos 4 e 8 trapézios sob a curva $f(x)$.

É evidente que a área sob $f(x)$ está agora muito bem aproximada com $n = 8$ trapézios. O seu valor pode ser estimado por

$$I_3 = \sum_{i=1}^n \frac{f(x_{i-1}) + f(x_i)}{2} \Delta x, \quad (2.3)$$

com

$$x_0 = 1, \quad (2.4)$$

$$x_n = 5, \quad (2.5)$$

$$\Delta x = \frac{x_n - x_0}{n}. \quad (2.6)$$

Prosseguindo, (2.3) pode ser re-escrita:

$$\begin{aligned} I_3 &= \left[\frac{f(x_0) + f(x_1)}{2} + \frac{f(x_1) + f(x_2)}{2} + \dots + \frac{f(x_{n-1}) + f(x_n)}{2} \right] \Delta x \\ &= \left[f(x_0) + 2(f(x_1) + f(x_2) + \dots + f(x_{n-1})) + f(x_n) \right] \frac{\Delta x}{2} \\ &= (S_e + 2S_i) \frac{\Delta x}{2}, \end{aligned} \quad (2.7)$$

com

$$S_e = f(x_0) + f(x_n), \quad (2.8)$$

$$S_i = \sum_{i=1}^{n-1} f(x_i). \quad (2.9)$$

Esta seção vale um *módulo* de Python, que nós vamos denominar `numint`. Uma implementação razoavelmente eficiente da regra do trapézio é mostrada nas primeiras 21 linhas de `numint.py`, na listagem 2.5

Listagem 2.5: `numint.py` — Integração numérica, regra do trapézio

```

1 #!/usr/bin/python
2 # -*- coding: iso-8859-1 -*-
3 from __future__ import unicode_literals
4 from __future__ import print_function
5 from __future__ import division
6 def trapezio(n,a,b,f):
7     '''
8     trapezio(n,a,b,f): integra f entre a e b com n trapézios
9     '''
10    deltax = (b-a)/n
11    Se = f(a) + f(b)           # define Se
12    Si = 0.0                   # inicializa Si
13    for k in range(1,n):      # calcula Si
14        xk = a + k*deltax
15        Si += f(xk)
16    I = Se + 2*Si              # cálculo de I
17    I *= deltax
18    I /= 2
19    return I

```

O programa `quadraver1.py` calcula a integral de $f(x)$ com 8 trapézios (listagem 2.6).

Listagem 2.6: `quadraver1.py` — Integração numérica de $f(x)$ com 8 trapézios

```

1 #!/usr/bin/python
2 # -*- coding: iso-8859-1 -*-
3 from __future__ import unicode_literals
4 from __future__ import print_function
5 from __future__ import division
6 from numint import trapezio
7 def f(x):
8     return ((-1/48)*x**4 + (13/48)*x**3 + (-17/12)*x**2 + (11/3)*x)
9 I3 = trapezio(8,1,5,f)
10 print('I3 = %8.4f\n' % I3)

```

A saída de `quadraver1.py` é $I_3 \approx 14,6328$. O erro está agora na 2ª casa decimal, e o erro relativo é $\delta = 0,0031$, ou 0,3%.

O “problema” com `numint.trapezio` é que nós não temos uma idéia do erro que estamos cometendo, porque, se estamos utilizando integração numérica, é porque não conhecemos o valor exato de I ! Um primeiro remédio para este problema é ficar recalculando a regra do trapézio com um número dobrado de trapézios, até que a diferença *absoluta* entre duas estimativas sucessivas fique abaixo de um valor estipulado. Isto é implementado, *de forma muito ineficiente*, na próxima rotina

do módulo `numint` (listagem 2.7: note a continuação da numeração de linhas dentro do *mesmo arquivo* `numint.py`), denominada `trapepsilonlento`, e mostrada na listagem 2.7.

Listagem 2.7: `numint.py` — Integração numérica ineficiente, com erro absoluto pré-estabelecido

```

20 def trapepsilonlento(epsilon,a,b,f):
21     '''
22     trapepsilonlento(epsilon,a,b,f): calcula a integral de f entre a e b com
23     erro absoluto epsilon, de forma ineficiente
24
25     '''
26     eps = 2*epsilon           # estabelece um erro inicial grande
27     n = 1                    # um único trapézio
28     Iv = trapezio(1,a,b,f)    # primeira estimativa, "velha"
29     while eps > epsilon:      # loop
30         n *= 2                # dobra o número de trapézios
31         In = trapezio(n,a,b,f) # estimativa "nova", recalculada do zero
32         eps = abs(In - Iv)     # calcula o erro absoluto
33         Iv = In                # atualiza a estimativa "velha"
34     return (In,eps)

```

O programa `quadraver2.py` calcula a integral de $f(x)$ com erro absoluto estipulado menor que 0,0001, e imprime a estimativa da integral, o erro absoluto e o erro relativo (em relação ao valor exato conhecido) encontrados: $I_4 = 14,677777$, $\epsilon = 0,00003$, $\delta = 0,00000075$. Com 4 casas decimais, este é um resultado exato!

Listagem 2.8: `quadraver2.py` — Integração numérica ineficiente de $f(x)$ com $\epsilon = 0,0001$

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  from __future__ import division
6  from numint import trapepsilonlento
7  def f(x):
8      return ((-1/48)*x**4 + (13/48)*x**3 + (-17/12)*x**2 + (11/3)*x)
9  (I4,eps) = trapepsilonlento(0.0001,1,5,f)
10 print('I4=□%8.5f□□eps=□%8.5f' % (I4,eps))
11 II = 14.677777777777777
12 delta = (I4 - II)/II
13 print('delta=□%10.8f' % delta)

```

O problema com `trapepsilonlento` é que todos os pontos que já haviam sido calculados por `trapezio` são recalculados em cada iteração (verifique). Nosso último esforço, a rotina `trapepsilon` em `numint.py`, corrige este problema, reaproveitando todos os cálculos. Volte um pouco à figura 2.2: nela, nós vemos a integração numérica de $f(x)$ com $n = 4$ e depois com $n = 8$ trapézios. Repare que S_i para $n = 4$ é parte do valor de S_i para $n = 8$. De fato, para $n = 4$, $S_i = f(x_2) + f(x_4) + f(x_6)$ (note que os índices já estão definidos para o caso $n = 8$). Esta soma já foi calculada na integral com 4 trapézios, e não precisa ser recalculada. O que nós precisamos fazer agora é somar $f(x_1) + f(x_3) + f(x_5) + f(x_7)$ ao S_i que já tínhamos. S_e permanece o mesmo, e depois basta aplicar (2.7). Isto é feito na última rotina de `numint`, denominada `trapepsilon`, na listagem 2.9.

Listagem 2.9: numint.py — Integração numérica eficiente, com erro absoluto pré-estabelecido

```

35 def trapepsilon(epsilon,a,b,f):
36     '''
37     trapepsilon(epsilon,a,b,f): calcula a integral de f entre a e b com
38     erro absoluto epsilon, de forma eficiente
39     '''
40     eps = 2*epsilon                # estabelece um erro inicial grande
41     n = 1                          # n é o número de trapézios
42     Se = f(a) + f(b)              # Se não muda
43     deltax = (b-a)/n              # primeiro deltax
44     dx2 = deltax/2                # primeiro deltax/2
45     Siv = 0.0                     # Si "velho"
46     Iv = Se*dx2                   # I "velho"
47     while eps > epsilon:           # executa o loop pelo menos uma vez
48         Sin = 0.0                 # Si "novo"
49         n *= 2                     # dobra o número de trapézios
50         deltax /= 2                # divide deltax por dois
51         dx2 = deltax/2             # idem para dx2
52         for i in range(1,n,2):     # apenas os ímpares...
53             xi = a + i*deltax      # pula os ptos já calculados!
54             Sin += f(xi)           # soma sobre os novos ptos internos
55             Sin = Sin + Siv         # aproveita todos os ptos já calculados
56             In = (Se + 2*Sin)*dx2  # I "novo"
57             eps = abs(In - Iv)      # calcula o erro absoluto
58             Siv = Sin              # atualiza Siv
59             Iv = In                # atualiza Iv
60     return (In,eps)

```

O programa `quadraver3.py` (listagem 2.10) calcula a integral de $f(x)$ com erro absoluto estipulado menor que 0,000001, e imprime a estimativa da integral, o erro absoluto e o erro relativo (em relação ao valor exato conhecido) encontrados: $I_5 = 14.6777776$, $\epsilon = 0.0000005$, $\delta = 0.00000001$. Note que I_5 é exato até a 6ª casa decimal, conforme estipulado.

Listagem 2.10: quadraver3.py — Integração numérica eficiente de $f(x)$ com $\epsilon = 0,000001$

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  from __future__ import division
6  from numint import trapepsilon
7  def f(x):
8      return ((-1/48)*x**4 + (13/48)*x**3 + (-17/12)*x**2 + (11/3)*x)
9  (I5,eps) = trapepsilon(0.000001,1,5,f)
10 print('I5= %12.7f f_{eps}= %12.7f' % (I5,eps))
11 II = 14.677777777777777
12 delta = (I5 - II)/II
13 print('delta= %12.8f' % delta)

```

Exercícios Propostos

2.4 Usando Python e `numint.trapezio`, aproxime $I = \int_0^\pi \sin(x) dx$ pela regra do trapézio com 10 trapézios.

2.5 Usando Python e `numint.trapepsilon`, aproxime $I = \int_0^\pi \sin(x) dx$ pela regra do trapézio com precisão absoluta menor ou igual a 1×10^{-5} .

2.6 Dada $f(x)$ definida no intervalo $[x_0, x_0 + 2h]$, deduza a regra de Simpson:

$$\int_{x_0}^{x_0+2h} f(x) dx \approx \frac{h}{3} [f_0 + 4f_1 + f_2],$$

com $f_n = f(x_n)$, $x_n = x_0 + nh$, interpolando a função $g(x) = ax^2 + bx + c$ através dos pontos (x_0, f_0) , (x_1, f_1) e (x_2, f_2) e calculando sua integral.

2.7 Dobrando o número de pontos de 2 para 4, obtenha a regra de Simpson para 5 pontos,

$$\int_{x_0}^{x_0+4h} f(x) dx \approx \frac{h}{3} [f_0 + 4f_1 + 2f_2 + 4f_3 + f_4];$$

generalize:

$$\int_{a \equiv x_0}^{b \equiv x_0 + 2nh} f(x) dx \approx \frac{h}{3} [f_0 + 4f_1 + 2f_2 + \dots + 2f_{2n-2} + 4f_{2n-1} + f_{2n}].$$

2.8 Estenda `numint` com uma rotina `simpson` para calcular a regra de Simpson com $2n$ intervalos, e uma rotina `simpepsilon` para calcular uma integral numérica pela regra de Simpson com precisão estipulada. Baseie-se em `trapezio` e `trapepsilon`.

2.3 – Aproximação de integrais com séries: a função erro

Integrações numéricas tais como as mostradas na seção 2.2 podem ser muito “custosas” em termos do número de operações de ponto flutuante necessárias. Algumas vezes, é possível ser mais inteligente. Um exemplo disto foi o cálculo de I_2 com uma única parábola quadrática¹ no fim da seção 2.1, que foi capaz de baixar o erro relativo para pouco mais de 1%.

Considere agora o cálculo de uma integral particularmente importante, a *função erro*, definida por

$$\operatorname{erf}(x) \equiv \frac{2}{\sqrt{\pi}} \int_{u=0}^x e^{-u^2} du. \quad (2.10)$$

A função erro está definida em Gnuplot e em Maxima, mas não em Python < 2.7. Como obtê-la? Uma maneira “força bruta” é utilizar `trapepsilon` com uma precisão razoável (digamos, 10^{-6}), gerar um arquivo, e plotar o resultado para “ver a cara” da $\operatorname{erf}(x)$. O programa `vererf.py` (listagem 2.11) calcula a função em um grande número de pontos; gera um arquivo de dados `vererf.dat`, e então o programa `vererf.plt` (listagem 2.12) plota o resultado, mostrado na figura 2.3.

Observe que `vererf.plt` utiliza a função `erf` pré-definida em Gnuplot e plota uma comparação: a linha contínua é a `erf` de Gnuplot, e os pontos são os valores obtidos por `vererf.py`.

Existe uma maneira mais inteligente de calcular $\operatorname{erf}(x)$: ela se baseia em integrar a *série de Taylor* do integrando, e^{-u^2} . Maxima permite calcular os primeiros termos:

```
(%i1) taylor(exp(-u^2), u, 0, 10) ;
              4      6      8      10
              2      u      u      u      u
(%o1) T/      1 - u + -- - -- + -- - --- + . . .
              2      6      24     120
```

¹Sim! existem parábolas cúbicas, quárticas, etc..

Listagem 2.11: `vererf.py` — Cálculo da função erro por integração numérica

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  from __future__ import division
6  from math import exp, sqrt, pi
7  from numint import trapepsilon
8  def f(x):
9      return exp(-x*x)
10  xmin = 0.0                # de 0
11  xmax = 3.0                # a 3
12  nx = 100                 # em 100 passos
13  dx = (xmax - xmin)/nx    # de dx
14  erf = 0.0                # erf(0) = 0
15  fou = open('vererf.dat','wt') # arquivo de saída
16  xl = xmin                # limite inferior a partir de xmin
17  fou.write('%8.6f_8.6f\n' % (xl,erf)) # erf(0) = 0
18  for k in range(nx):      # loop
19      xu = xl + dx          # novo limite superior
20      (I,eps) = trapepsilon(1.0e-6,xl,xu,f) # integra mais uma fatia
21      erf = erf + (2.0/sqrt(pi))*I          # acumula erf
22      fou.write('%8.6f_8.6f\n' % (xu,erf)) # imprime até aqui
23      xl = xu               # atualiza limite inferior
24  fou.close()              # fecha o arquivo de saída

```

Listagem 2.12: `vererf.plt` — Plotagem da função erro por integração numérica *versus* a $\text{erf}(x)$ pré-definida em Gnuplot

```

1  set encoding iso_8859_1
2  # -----
3  # no mundo de linux
4  # -----
5  set terminal postscript eps monochrome 'Times-Roman' 18
6  set output 'vererf.eps'
7  set format y '%3.1f'
8  set grid
9  set xlabel 'x'
10 set ylabel 'erf(x)'
11 set xrange [0:3]
12 set yrange [0:1]
13 set xtics 0,0.5
14 set ytics 0,0.1
15 plot 'vererf.dat' using 1:2 title 'trapepsilon' with points pt 7 ps 0.75,\
16      erf(x) title 'Gnuplot' with lines lt 1 lw 2
17 # -----
18 # agora no mundo de Windows
19 # -----
20 set terminal emf monochrome 'Times_New_Roman' 18
21 set output 'vererf.emf'
22 replot
23 exit

```

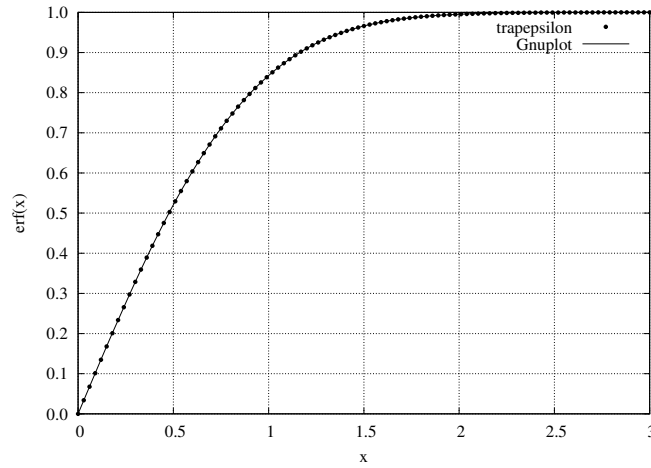


Figura 2.3: Função $\text{erf}(x)$ calculada por integração numérica, com `trapepsilon` e $\epsilon = 1 \times 10^{-6}$, *versus* a `erf` pré-definida em Gnuplot.

A expansão é em torno de $u = 0$, e é feita até o 10º termo. Não é muito difícil reconhecer nos denominadores os fatoriais de 0, 1, 2, 3, 4 e 5, e as potências dos dobros destes valores nos expoentes de u . Em geral,

$$e^{-u^2} = \sum_{n=0}^{\infty} (-1)^n \frac{u^{2n}}{n!}. \quad (2.11)$$

Portanto,

$$\begin{aligned} \int_0^x e^{-u^2} du &= \int_0^x \sum_{n=0}^{\infty} (-1)^n \frac{u^{2n}}{n!} du \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \int_0^x u^{2n} du \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \frac{x^{2n+1}}{2n+1} \\ &= \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)n!}. \end{aligned} \quad (2.12)$$

Cuidado: não é toda série que permite a troca impune da ordem de integração e da soma (infinita) da série. Em princípio, devemos procurar os teoremas relevantes que nos permitem esta troca de ordem. Admitindo que está tudo bem, entretanto, nós conseguimos a série de Taylor de $\text{erf}(x)$!

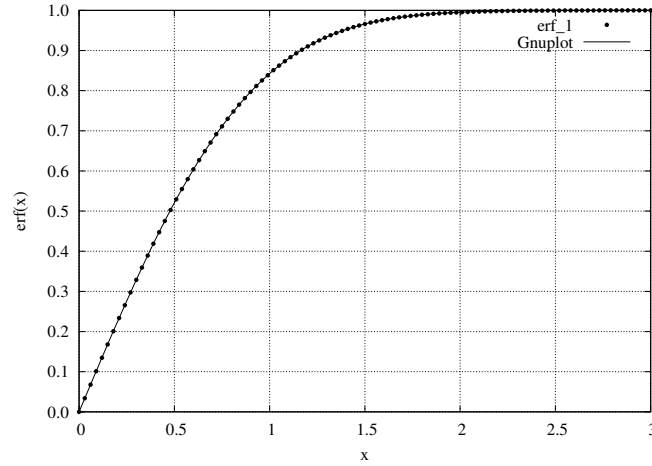


Figura 2.4: Função $\text{erf}(x)$ calculada com série de Taylor, com `erf_1`, *versus* a `erf` pré-definida em Gnuplot.

Ainda falta encontrar uma maneira computacionalmente eficiente de passar do termo $n - 1$ para o termo n . Vamos a isto:

$$\begin{aligned}
 \frac{x^{2n+1}}{(2n+1)n!} &\equiv \frac{A_n}{B_n C_n} \\
 &= \frac{(-1)^n x^{2(n-1+1)+1}}{(2(n-1+1)+1)(n(n-1)!)} \\
 &= \frac{(-1)^{n-1} x^{2(n-1)+1} \times [-x^2]}{(2(n-1)+1+[2])([n](n-1)!)} \\
 &= \frac{A_{n-1} \times (-x^2)}{(B_{n-1}+2)(C_{n-1} \times n)} \tag{2.13}
 \end{aligned}$$

Em outras palavras, o numerador A_n de cada novo termo da série é o anterior vezes $-x^2$. O denominador é mais complicado; ele é formado pelo produto $B_n C_n$, e as relações completas de recursão são

$$\begin{aligned}
 A_n &= x^{2n+1} = A_{n-1} \times (-x^2), \\
 B_n &= 2n+1 = B_{n-1} + 2, \\
 C_n &= n! = C_{n-1} \times n.
 \end{aligned}$$

Mais uma coisa: a série de $\text{erf}(x)$ só contém potências ímpares: portanto, $\text{erf}(x)$ é uma função *ímpar*, e vale a relação

$$\text{erf}(-x) = -\text{erf}(x).$$

Com isto, temos uma rotina em Python para calcular $\text{erf}(x)$, chamada `erf_1(x)`, no módulo `erfs`, do arquivo `erfs.py`. As linhas correspondentes a `erf_1(x)` são mostradas na listagem 2.13

Agora, podemos escrever `vererf1.py` (listagem 2.14), gerar um arquivo de saída `vererf1.dat` e plotar com o programa `vererf1.plt` (listagem 2.15). Os resultados continuam idênticos à $\text{erf}(x)$ de Gnuplot.

Listagem 2.13: Cálculo de $\text{erf}(x)$ com uma série de Taylor.

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  from __future__ import division
6  from math import sqrt, pi
7  def erf_1(x):
8      epsilon_erf = 1.0e-6          # mesma precisão
9      eps_erf = 2*epsilon_erf       # garante entrada no while
10     A = x                          # primeiro A
11     B = 1                          # primeiro B
12     C = 1                          # primeiro C
13     n = 0                          # primeiro n
14     termo = A/(B*C)                # primeiro termo da série
15     s = termo                       # primeira soma da série
16     while eps_erf > epsilon_erf:   # loop
17         n += 1                     # incrementa n
18         A *= (-x*x)                 # novo A
19         B += 2                      # novo B
20         C *= n                      # novo C
21         termo = A/(B*C)              # novo termo
22         eps_erf = abs(termo)         # seu valor absoluto
23         s += termo                  # soma na série
24     return (2/sqrt(pi) * s,n)

```

Listagem 2.15: `vererf1.plt` — Plotagem da função erro calculada com série de Taylor *versus* a $\text{erf}(x)$ pré-definida em Gnuplot

```

1 set encoding iso_8859_1
2 # -----
3 # no mundo de linux
4 # -----
5 set terminal postscript eps monochrome 'Times-Roman' 18
6 set output 'vererf1.eps'
7 set format y '%3.1f'
8 set grid
9 set xlabel 'x'
10 set ylabel 'erf(x)'
11 set xrange [0:3]
12 set yrange [0:1]
13 set xtics 0,0.5
14 set ytics 0,0.1
15 plot 'vererf1.dat' using 1:2 title 'erf_1' with points pt 7 ps 0.75,\
16     erf(x) title 'Gnuplot' with lines lt 1 lw 2
17 # -----
18 # agora no mundo de Windows
19 # -----
20 set terminal emf monochrome 'Times_New_Roman' 18
21 set output 'vererf1.emf'
22 replot
23 exit

```

Embora formalmente correta, a rotina `erf_1` fica mais lenta à medida em que $|x|$ cresce. Como já vimos nas figuras 2.3 e 2.4, $\text{erf}(x) \approx 1$ para $|x| \gtrsim 3$. Porém, uma olhada em `vererf1.dat`, nos dá

$\left[\text{erf}_1(3) = 0.999978 \right],$

que ainda está acima da precisão especificada de 0,000001. Precisamos descobrir o valor de x para o qual `erf_1` produz 0,999999. Por tentativa e erro, encontramos

$\left[\text{erf}_1(3.6) = 0.99999952358847277 \right],$

que é igual a 1,0 para 6 casas decimais. Precisamos também saber com quantos termos este cálculo foi feito, e este é o motivo de `erf_1` devolver também n . Para calcular $\text{erf}(3,6)$ com `erf_1`, nós precisamos de $n = 43$ termos. A rotina `erf` na listagem 2.16, também no arquivo `erfs.py`, usa este fato para impedir que o número de termos continue crescendo. Verifique, você mesmo(a), que `erf(100)` retorna 1.0, mas que `erf_1(100)` dá um erro de ponto flutuante.

É preciso enfatizar que `erf` em `erfs.py` *ainda não é uma rotina “profissional”*. O número de termos usado ainda é potencialmente muito grande; conseqüentemente, há muitas operações de ponto flutuante envolvendo `A`, `B` e `C`, e muitos testes dentro do `while`.

É possível obter fórmulas que aproximam $\text{erf}(x)$ com menos termos, uniformemente, no intervalo (digamos) de 0 a 3,6: veja, por exemplo, [Abramowitz e Stegun \(1972\)](#). Mesmo assim, `erf` é uma opção vantajosa em relação à integração numérica.

A lição desta seção é a seguinte: em geral, com esforço analítico adicional, é possível obter uma mistura de métodos analíticos e numéricos que costuma ser amplamente superior ao uso de um método numérico “puro” (por exemplo a regra do trapézio) para a obtenção de resultados semelhantes. Exemplos deste fato vão reaparecer nos capítulos seguintes.

Listagem 2.16: `erfs.py` — Cálculo de $\operatorname{erf}(x)$ com série de Taylor, limitado a no máximo 43 termos

```
25 def erf(x):
26     if x > 3.6:                                     # limita o número de termos a 43
27         return 1.0
28     elif x < -3.6:
29         return -1.0
30     epsilon_erf = 1.0e-6                             # mesma precisão
31     eps_erf = 2*epsilon_erf                         # garante entrada no while
32     A = x                                             # primeiro A
33     B = 1                                             # primeiro B
34     C = 1                                             # primeiro C
35     n = 0                                             # primeiro n
36     termo = A/(B*C)                                  # primeiro termo da série
37     s = termo                                         # primeira soma da série
38     while eps_erf > epsilon_erf:                     # loop
39         n += 1                                       # incrementa n
40         A *= (-x*x)                                 # novo A
41         B += 2                                       # novo B
42         C *= n                                       # novo C
43         termo = A/(B*C)                             # novo termo
44         eps_erf = abs(termo)                         # seu valor absoluto
45         s += termo                                   # soma na série
46     return 2/sqrt(pi) * s
```

Exercícios Propostos

2.9 (Bender e Orszag (1978), seção 6.2, Exemplo 2) Obtenha uma série para

$$F(x) \equiv \int_0^x t^{-1/2} e^{-t} dt;$$

compare o resultado obtido com integração numérica.

3

Solução numérica de equações diferenciais ordinárias

3.1 – Solução numérica de equações diferenciais ordinárias

Solução analítica de uma EDO com Maxima. Considere uma equação diferencial de 1ª ordem simples, forçada eternamente por um seno:

$$\frac{dy}{dx} + \frac{y}{x} = \sin x. \quad (3.1)$$

Na listagem 3.1, nós resolvemos esta equação com Maxima.

Listagem 3.1: `resolve-eqdif` — Solução de uma EDO com Maxima

```
1          y      dy
2 (%i2)      - + -- = sin(x)
3          x      dx
4          dy      y
5 (%o2)      -- + - = sin(x)
6          dx      x
7 (%i3)      ode2(%, y, x)
8          sin(x) - x cos(x) + %c
9 (%o3)      y = -----
10          x
```

Maxima nos informa de que a solução geral é da forma

$$y(x) = \frac{\sin x - x \cos x + c}{x}.$$

É evidente que, em geral, nem a equação diferencial nem sua solução "existem" em $x = 0$. Entretanto, para $c = 0$,

$$y(x) = \frac{\sin x}{x} - \cos x.$$

Agora,

$$\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1,$$

de modo que existe uma solução para a equação partindo de $x = 0$ se nós impusermos a condição inicial $y(0) = 0$. De fato:

$$\lim_{x \rightarrow 0} \left[\frac{\sin x}{x} - \cos x \right] = 1 - 1 = 0.$$

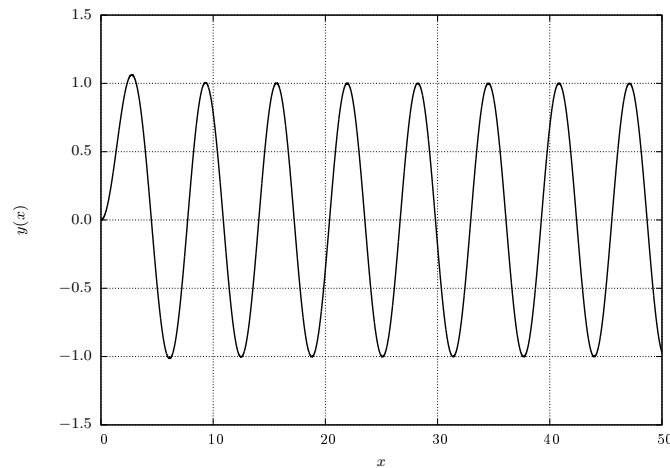


Figura 3.1: Solução da equação (3.1).

O resultado está mostrado na figura 3.1.

Claramente, existe uma parte “transiente” da solução, dada por $\sin x/x$, que “morre” à medida que x cresce, e existe uma parte periódica (mas não permanente!) da solução, dada por $\cos x$, que “domina” $y(x)$ quando x se torna grande. Nós dizemos que $\cos x$ é parte *estacionária* da solução.

3.1.1 – Solução numérica; método de Euler

A coisa mais simples que pode ser pensada para resolver a equação diferencial em questão é transformar a derivada em uma *diferença finita*:

$$\frac{\Delta y}{\Delta x} + \frac{y}{x} = \sin x$$

Isto é um começo, mas não é suficiente. Na verdade, o que desejamos é que o computador gere uma *lista* de x 's (uniformemente espaçados por Δx), e uma *lista* de y 's correspondentes. Obviamente, como os y 's devem aproximar a função, não podemos esperar deles que sejam igualmente espaçados!

Desejamos então:

$$x_0, x_1, \dots, x_n$$

onde

$$x_i = i\Delta x,$$

com os correspondentes

$$y_0, y_1, \dots, y_n.$$

Como Δx será fixo, podemos escrever nossa equação de diferenças finitas da seguinte forma:

$$\frac{y_{i+1} - y_i}{\Delta x} + \frac{y_i}{x_i} = \sin(x_i),$$

onde eu deixei, propositalmente,

$$\dots \frac{x}{y} = \sin(x)$$

ainda sem índices. De fato: qual x_i e qual y_i usar aqui? A coisa mais simples, mas também a mais *instável*, é usar i :

$$\frac{y_{i+1} - y_i}{\Delta x} + \frac{y_i}{x_i} = \sin(x_i).$$

Listagem 3.2: `fracasso.py` — Um programa com o método de Euler que não funciona

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  from __future__ import unicode_literals
4  from __future__ import print_function
5  # -----
6  # cria listas com condições iniciais
7  # -----
8  x = [0.0]
9  y = [0.0]
10 dx = 0.01
11 n = int(50/0.01)
12 from math import sin
13 for i in range(0,n): # de 0 até ... n-1 !!!!
14     xn = (i+1)*dx
15     yn = y[i] + (sin(x[i]) - y[i]/x[i])*dx
16     x.append(xn)
17     y.append(yn)
18 fou = open('fracasso.out','wt')
19 for i in range(n):
20     fou.write('%12.6f %12.6f\n' % (x[i],y[i]))
21 fou.close()

```

Note que é agora possível explicitar y_{i+1} em função de todos os outros valores em i :

$$y_{i+1} = y_i + \left[\sin(x_i) - \frac{y_i}{x_i} \right] \Delta x.$$

Este é um exemplo de um método *explícito*: o novo valor da função em x_{i+1} (y_{i+1}) só depende de valores calculados no valor anterior, x_i . Um olhar um pouco mais cuidadoso será capaz de prever o desastre: na fórmula acima, se partirmos de $x_0 = 0$, teremos uma divisão por zero já no primeiro passo!

Muitos não veriam isto, entretanto, e nosso primeiro programa para tentar resolver a equação diferencial numericamente se chamará `fracasso.py`, e está mostrado na listagem 3.2

O resultado, mostrado na listagem 3.3 é o seguinte fracasso:

Listagem 3.3: Saída de `fracasso.py`

```

Traceback (most recent call last):
  File "./fracasso.py", line 15, in <module>
    yn = y[i] + (sin(x[i]) - y[i]/x[i])*dx
ZeroDivisionError: float division by zero

```

Isto já era previsível: quando $i == 0$ no *loop*, $x[0] == 0$ no denominador, e o programa para com uma divisão por zero. Para conseguir fazer o método numérico funcionar, nós vamos precisar de mais *análise*! De volta à equação diferencial, note que para que exista uma solução na vizinhança de $x = 0$ é necessário que o limite $\lim_{x \rightarrow 0} y(x)/x$ exista; nós devemos ter

$$\lim_{x \rightarrow 0} \frac{y(x)}{x} = L,$$

onde L é uma constante *finita* a determinar. Mas, de nossa condição inicial,

$$\lim_{x \rightarrow 0} y(x) = 0 \Rightarrow L = 0.$$

Listagem 3.4: sucesso.py — Um programa com o método de Euler que funciona

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # sucesso: resolve a equação diferencial
5  # dy/dx + y/x = sen(x)
6  # pelo método de Euler.  Uso:
7  #
8  # ./sucesso.py <dx> <arquivo de saída>
9  # -----
10 from __future__ import print_function
11 from __future__ import division
12 from sys import argv
13 dx = float(argv[1])          # tamanho do passo
14 x = [0.0, dx]               # condições iniciais em x
15 y = [0.0, 0.0]              # condições iniciais em y
16 n = int(50/dx)
17 from math import sin, cos
18 for i in range(1,n):         # solução numérica
19     xn = (i+1)*dx
20     yn = y[i] + (sin(x[i]) - y[i]/x[i])*dx
21     x.append(xn)
22     y.append(yn)
23 erro = 0.0
24 for i in range(1,n+1):       # calcula o erro relativo médio
25     yana = sin(x[i])/x[i] - cos(x[i])
26     erro += abs( (y[i] - yana)/yana )
27 erro /= n ;
28 print ( 'erro relativo médio = ', '%10.5f' % erro )
29 fou = open(argv[2], 'wt')
30 for i in range(0,n+1):       # imprime o arquivo de saída
31     fou.write( '%12.6f %12.6f\n' % (x[i], y[i]) )
32 fou.close()

```

Vamos então reescrever a equação de diferenças *usando* o limite:

$$y_1 = y_0 + \underbrace{\left[\sin(x_0) - \frac{y_0}{x_0} \right]}_{=0, \lim_{x_0 \rightarrow 0}} \Delta x = 0.$$

Na prática, isto significa que nós podemos começar o programa do ponto $x_1 = \Delta x$, $y_1 = 0$! Vamos então reescrever o código, que nós agora vamos chamar, é claro, de `sucesso.py`, que pode ser visto na listagem 3.4

A saída de `sucesso.py` gera o arquivo `sucesso.out`, que nós utilizamos para plotar uma comparação entre a solução analítica e a solução numérica, mostrada na figura 3.2

Na verdade, o sucesso é estrondoso: com $\Delta x = 0,01$, nós conseguimos produzir uma solução numérica que é visualmente indistinguível da solução analítica. Uma das coisas que o programa ‘sucesso.py’ calculou foi o *erro relativo médio*

$$\epsilon \equiv \sum_{i=1}^n \left| \frac{y_i - y(x_i)}{y(x_i)} \right|$$

(onde y_i é a solução numérica, e $y(x_i)$ é a solução exata no mesmo ponto x_i). Para $\Delta x = 0,01$, $\epsilon = 0,02619$, ou seja: menos de 3%.

O preço, entretanto, foi “alto”: nós precisamos de um Δx bem pequeno, e de $50/0,01 = 5000$ pontos para gerar a solução. Será possível gerar uma solução tão boa com, digamos, 100 pontos?

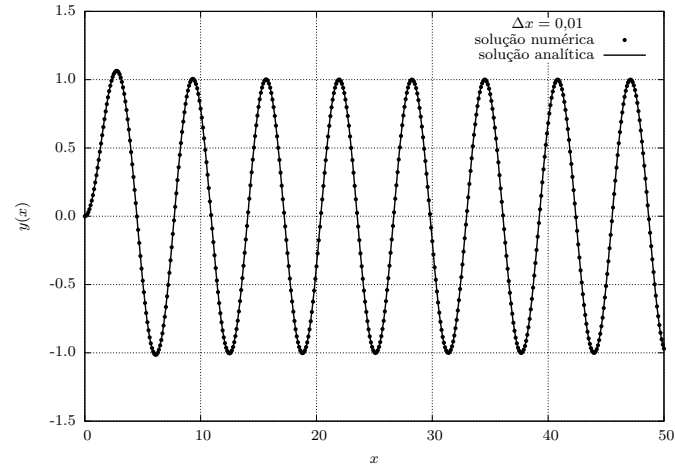


Figura 3.2: Comparação da solução analítica da equação (3.1) com a saída de `sucesso.py`, para $\Delta x = 0,01$.

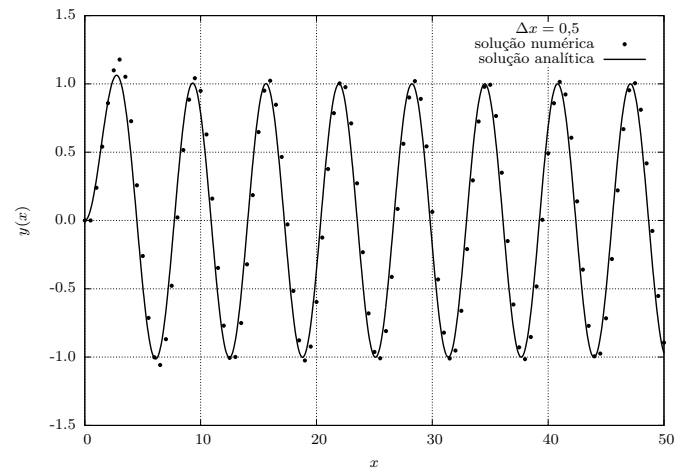


Figura 3.3: Comparação da solução analítica da equação (3.1) com a saída de `sucesso.py`, para $\Delta x = 0,5$.

A figura 3.3 mostra o resultado de rodar `sucesso.py` com $\Delta x = 0,5$, muito maior do que antes.

O erro médio relativo agora pulou para $\epsilon = 1,11774$, nada menos do que 111%, e muito pior do que a figura acima faz parecer à primeira vista!

3.1.2 – Um método implícito, com tratamento analítico

Nosso desafio é desenvolver um método numérico que melhore consideravelmente a solução mesmo com um Δx grosseiro, da ordem de 0,5. Nossa abordagem será propor um método *implícito*:

$$\frac{y_{i+1} - y_i}{\Delta x} + \frac{y_i + y_{i+1}}{x_i + x_{i+1}} = \text{sen} \left(\frac{x_i + x_{i+1}}{2} \right).$$

Note que tanto o termo y/x quando $\text{sen } x$ estão sendo agora avaliados no ponto *médio* entre x_i e x_{i+1} .

Lembrando que $\Delta x = x_{i+1} - x_i$, y_{i+1} :

$$\begin{aligned} \frac{y_{i+1} - y_i}{\Delta x} + \frac{y_i + y_{i+1}}{x_i + x_{i+1}} &= \text{sen} \left(\frac{x_i + x_{i+1}}{2} \right), \\ y_{i+1} \left[\frac{1}{\Delta x} + \frac{1}{x_i + x_{i+1}} \right] + y_i \left[-\frac{1}{\Delta x} + \frac{1}{x_i + x_{i+1}} \right] &= \text{sen} \left(\frac{x_i + x_{i+1}}{2} \right), \\ y_{i+1} \left[\frac{2x_{i+1}}{\Delta x(x_{i+1} + x_i)} \right] - y_i \left[\frac{2x_i}{\Delta x(x_{i+1} + x_i)} \right] &= \text{sen} \left(\frac{x_i + x_{i+1}}{2} \right). \end{aligned}$$

Uma rápida rearrumação produz

$$\begin{aligned} y_{i+1}x_{i+1} - y_ix_i &= \frac{\Delta x(x_{i+1} + x_i)}{2} \text{sen} \left(\frac{x_i + x_{i+1}}{2} \right), \\ y_{i+1} &= y_i \frac{x_i}{x_{i+1}} + \frac{\Delta x(x_{i+1} + x_i)}{2x_{i+1}} \text{sen} \left(\frac{x_i + x_{i+1}}{2} \right). \end{aligned} \quad (3.2)$$

Repare que a condição inicial $y(0) = 0$ não produz nenhuma singularidade em (3.2) para $i = 0 \Rightarrow x_0 = 0$, $y_0 = 0$. O programa que implementa este esquema é o `sucimp.py`, mostrado na listagem 3.5.

O resultado é um sucesso mais estrondoso ainda, e pode ser visto na figura 3.4.

Agora, o erro médio relativo baixou para $\epsilon = 0,02072$, que é ainda menor do que o do método de Euler com $\Delta x = 0,01$, ou seja: com um Δx 50 vezes menor!

3.1.3 – Runge-Kutta

O preço que nós pagamos pelo método extremamente acurado implementado em `sucimp.py` foi trabalhar analiticamente a equação diferencial, até chegar à versão “dedicada” (3.2). Isto é bom! Porém, às vezes não há tempo ou não é possível melhorar o método por meio de um “pré-tratamento” analítico. Nossa discussão agora nos levará a um método excepcionalmente popular, denominado método de Runge-Kutta.

Dada a equação

$$\frac{dy}{dx} = f(x, y)$$

nós podemos tentar estimar a derivada “no meio” do intervalo h (estamos mudando a notação: até agora, usávamos Δx , mas h é uma forma usual de explicitar o passo

Listagem 3.5: sucimp.py — Método de Euler implícito

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # sucimp: resolve a equação diferencial
5  #  $dy/dx + y/x = \sin(x)$ 
6  # usando um método implícito sob medida
7  # -----
8  from __future__ import print_function
9  from __future__ import division
10 dx = 0.5                      # passo em x
11 x = [0.0]                     # x inicial
12 y = [0.0]                     # y inicial
13 n = int(50/dx)                 # número de passos
14 from math import sin, cos
15 for i in range(0,n):           # loop na solução numérica
16     xn = (i+1)*dx
17     xm = x[i] + dx/2.0
18     yn = y[i]*x[i]/xn + (dx*xm/xn)*sin((x[i]+xn)/2)
19     x.append(xn)
20     y.append(yn)
21 erro = 0.0
22 for i in range(1,n+1):         # calcula o erro relativo médio
23     yana = sin(x[i])/x[i] - cos(x[i])
24     erro += abs( (y[i] - yana)/yana )
25 erro /= n ;
26 print ( 'erro relativo médio = ', '%10.5f' % erro )
27 fou = open('sucimp.out','wt')
28 for i in range(0,n+1):         # imprime o arquivo de saída
29     fou.write( '%12.6f%12.6f\n' % (x[i],y[i]) )
30 fou.close()

```

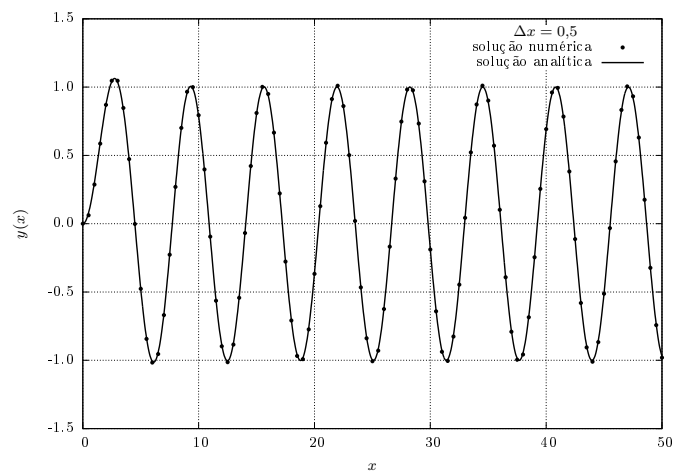


Figura 3.4: Comparação da solução analítica da equação (3.1) com a saída de sucimp.py, para $\Delta x = 0,5$.

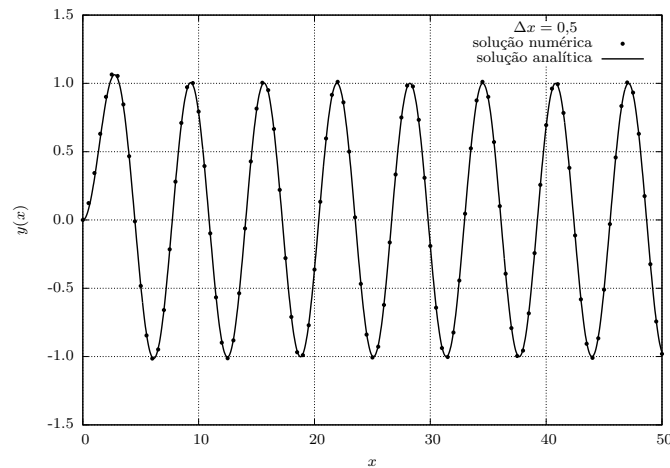


Figura 3.5: Comparação da solução analítica da equação (3.1) com a saída de `euler2.py`, para $\Delta x = 0,5$.

quando se trata do método de Runge-Kutta, ou métodos similares):

$$\left. \frac{dy}{dx} \right|_{x+h/2} = f(x + h/2, y(x + h/2))$$

O problema é que nós não conhecemos y em $x + h/2$! Isto pode ser contornado, entretanto, usando o método de Euler de 1ª ordem para *primeiro* estimar $y(x + h/2)$:

$$y_{n+1/2} \approx y_n + hf(x_n, y_n)/2.$$

Um método mais acurado será portanto

$$\begin{aligned} k_1 &= hf(x_n, y_n), \\ k_2 &= hf(x_n + h/2, y_n + k_1/2), \\ y_{n+1} &= y_n + k_2 \end{aligned}$$

Vamos tentar este método e ver como ele se compara com nossos esforços anteriores. Vamos manter $h = 0,5$ como nos casos anteriores. No entanto, nós ainda sofremos do cálculo da derivada em $x = 0$; por isto, nós vamos mudar o cálculo da derivada, colocando um `if` na função `ff` que a calcula. Note que, do ponto de vista de *eficiência* computacional, isto é péssimo, porque o `if` será verificado em *todos* os passos, quando na verdade ele só é necessário no passo zero. No entanto, o programa resultante, `euler2.py` (listagem 3.6), fica mais simples e fácil de entender, e esta é nossa prioridade aqui.

O resultado é mostrado na figura 3.5.

O resultado é muito bom, com um erro absoluto médio $\epsilon = 0,02529$. Mas nós podemos fazer melhor, com o método de Runge-Kutta de 4ª ordem! Não vamos deduzir as equações, mas elas seguem uma lógica parecida com a do método de

Listagem 3.6: euler2 — Um método explícito de ordem 2

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # euler2: resolve a equação diferencial
5  #  $dy/dx + y/x = \sin(x)$ 
6  # usando um método explícito de ordem 2 (Euler)
7  # -----
8  from __future__ import print_function
9  from __future__ import division
10 h = 0.5                                # passo em x
11 x = [0.0]                              # x inicial
12 y = [0.0]                              # y inicial
13 n = int(50/h)                          # número de passos
14 from math import sin, cos
15 def ff(x,y):
16     if x == 0.0:                        # implementa a condição inicial
17         return 0.0
18     else:
19         return sin(x) - y/x
20 def eul2(x,y,h,ff):
21     k1 = h*ff(x,y)
22     k2 = h*ff(x+h/2,y+k1/2)
23     yn = y + k2
24     return yn
25 for i in range(0,n):                    # loop da solução numérica
26     xn = (i+1)*h
27     yn = eul2(x[i],y[i],h,ff)
28     x.append(xn)
29     y.append(yn)
30 erro = 0.0                             # calcula o erro relativo médio
31 for i in range(1,n+1):
32     yana = sin(x[i])/x[i] - cos(x[i])
33     erro += abs( (y[i] - yana)/yana )
34 erro /= n ;
35 print ( 'erro relativo médio = ', '%10.5f' % erro )
36 fou = open('euler2.out','wt')
37 for i in range(0,n+1):                  # imprime o arquivo de saída
38     fou.write( '%12.6f%12.6f\n' % (x[i],y[i]) )
39 fou.close()

```

Listagem 3.7: rungek4 — Método de Runge-Kutta, ordem 4

```

1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  # -----
4  # rungek4: resolve a equação diferencial
5  # dy/dx + y/x = sen(x)
6  # usando o método de Runge-Kutta de ordem 4
7  # -----
8  from __future__ import print_function
9  h = 0.5          # passo em x
10 x = [0.0]        # x inicial
11 y = [0.0]        # y inicial
12 n = int(50/h)    # número de passos
13 from math import sin, cos
14 def ff(x,y):
15     '''
16     estou integrando dy/dx = sen(x) - y/x
17     '''
18     if x == 0.0:
19         return 0.0
20     else:
21         return sin(x) - y/x
22 def rk4(x,y,h,ff):
23     '''
24     rk4 implementa um passo do método de Runge-Kutta de ordem 4
25     '''
26     k1 = h*ff(x,y)
27     k2 = h*ff(x+h/2,y+k1/2)
28     k3 = h*ff(x+h/2,y+k2/2)
29     k4 = h*ff(x+h,y+k3)
30     yn = y + k1/6.0 + k2/3.0 + k3/3.0 + k4/6.0
31     return yn
32 for i in range(0,n):          # loop da solução numérica
33     xn = (i+1)*h
34     yn = rk4(x[i],y[i],h,ff)
35     x.append(xn)
36     y.append(yn)
37 erro = 0.0                   # calcula o erro relativo médio
38 for i in range(1,n+1):
39     yana = sin(x[i])/x[i] - cos(x[i])
40     erro += abs( (y[i] - yana)/yana )
41 erro /= n ;
42 print ( 'erro relativo médio = ', '%10.5f' % erro )
43 fou = open('rungek4.out','wt')
44 for i in range(0,n+1):      # imprime o arquivo de saída
45     fou.write( '%12.6f%12.6f\n' % (x[i],y[i]) )
46 fou.close()

```

ordem 2:

$$\begin{aligned}
 k_1 &= hf(x_n, y_n), \\
 k_2 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right), \\
 k_3 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right), \\
 k_4 &= hf(x_n + h, y_n + k_3), \\
 y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}.
 \end{aligned}$$

Para o nosso bem conhecido problema, o método é implementado no programa `rungek4`, na listagem 3.7.

O resultado é mostrado na figura 3.6.

Desta vez, o erro absoluto médio foi $\epsilon = 0,00007$: o campeão de todos os métodos tentados até agora, e uma clara evidência da eficácia do método de Runge-Kutta.

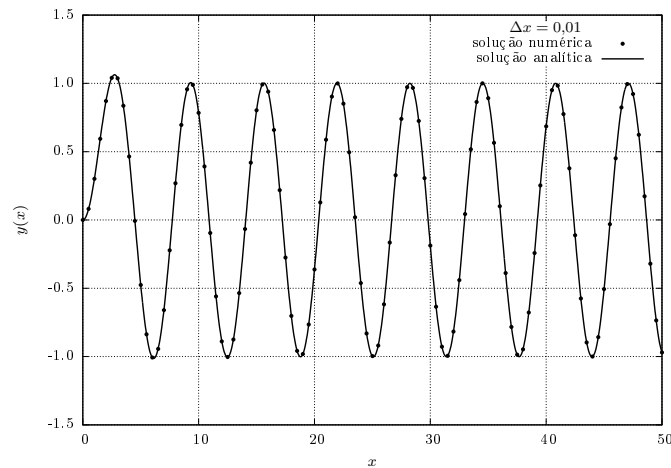


Figura 3.6: Comparação da solução analítica da equação (3.1) com a saída de `rungek4.py`, para $\Delta x = 0,5$.

Exercícios Propostos

3.1 Resolva, usando o método de Euler e o Método de Runge-Kutta de ordem 4:

$$\frac{dy}{dx} + xy = \sin(x), \quad y(0) = 0.$$

3.2 Resolva, usando o método de Euler de ordem 2, e compare com a solução analítica:

$$\frac{dy}{dx} + y = x^2 \exp(-x), \quad y(0) = 1.$$

3.3 Na equação

$$\frac{dy}{dx} + \frac{y}{x} = \sin\left(\frac{2\pi x}{L}\right), \quad y(0) = 0,$$

estude a sensibilidade do h , necessário para produzir $\epsilon = 0,001$, ao valor L .

3.4 Utilizando um método implícito semi-analítico, resolva

$$\frac{dy}{dx} + \frac{y}{x} = \frac{e^x}{x}, \quad y(x) = 0.$$

3.5 Resolva, utilizando Runge-Kutta:

$$\frac{dy}{dx} + y = \sin(x), \quad y(0) = 1.$$

4

Solução numérica de equações diferenciais parciais

4.1 – Advecção pura: a onda cinemática

Considere a equação

$$\frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0, \quad u(x, 0) = g(x). \quad (4.1)$$

A sua solução pode ser obtida pelo método das características, e é

$$u(x, t) = g(x - ct). \quad (4.2)$$

Seja então o problema

$$\frac{\partial u}{\partial t} + 2 \frac{\partial u}{\partial x} = 0, \quad (4.3)$$

$$u(x, 0) = 2x(1 - x). \quad (4.4)$$

A condição inicial, juntamente com $u(x, 1)$, $u(x, 2)$ e $u(x, 3)$ estão mostrados na figura 4.1. Observe que a solução da equação é uma simples onda cinemática.

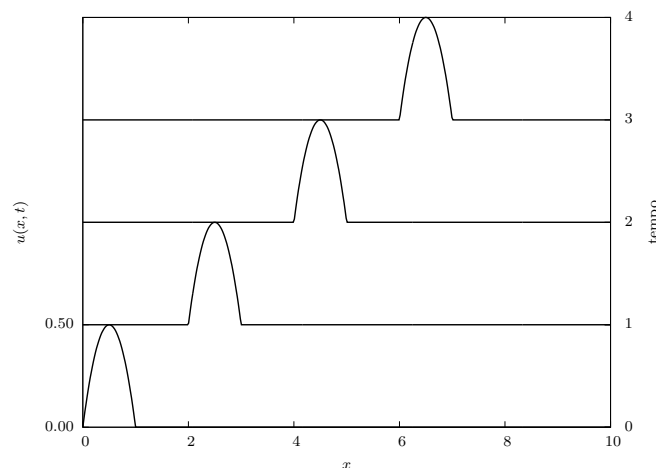


Figura 4.1: Condição inicial da equação 4.3.

Vamos adotar a notação

$$u_i^n \equiv u(x_i, t_n), \quad (4.5)$$

$$x_i = i\Delta x, \quad (4.6)$$

$$t_n = n\Delta t, \quad (4.7)$$

com

$$\Delta x = L/N_x, \quad (4.8)$$

$$\Delta t = T/N_t \quad (4.9)$$

onde L, T são os tamanhos de grade no espaço e no tempo, respectivamente, e N_x, N_t são os números de divisões no espaço e no tempo.

Uma maneira simples de transformar as derivadas parciais em diferenças finitas na equação (4.3) é fazer

$$\left. \frac{\partial u}{\partial t} \right|_{i,n} = \frac{u_i^{n+1} - u_i^n}{\Delta t} + O(\Delta t), \quad (4.10)$$

$$\left. \frac{\partial u}{\partial x} \right|_{i,n} = \frac{u_{i+1}^n - u_{i-1}^n}{\Delta x} + O(\Delta x^2). \quad (4.11)$$

Substituindo na equação (4.3), obtemos o esquema de diferenças finitas *explícito*:

$$\begin{aligned} \frac{u_i^{n+1} - u_i^n}{\Delta t} &= -c \left(\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} \right), \\ u_i^{n+1} &= u_i^n - \frac{c\Delta t}{2\Delta x} (u_{i+1}^n - u_{i-1}^n), \end{aligned} \quad (4.12)$$

(com $c = 2$ no nosso caso). Este é um esquema incondicionalmente *instável*, e vai fracassar. Vamos fazer uma primeira tentativa, já conformados com o fracasso antecipado. Ela vai servir para desenferujar nossas habilidades de programação de métodos de diferenças finitas.

O programa que implementa o esquema instável é o `onda1d-ins.py`, mostrado na listagem 4.1. Por motivos que ficarão mais claros na sequência, nós escolhemos $\Delta x = 0,01$, e $\Delta t = 0,0005$.

O programa gera um arquivo de saída binário, que por sua vez é lindo pelo próximo programa na sequência, `surf1d-ins.py`, mostrado na listagem 4.2. O único trabalho deste programa é selecionar algumas “linhas” da saída de `onda1d-ins.py`; no caso, nós o rodamos com o comando

`[onda1d-ins.py 3 250]`,

o que significa selecionar 3 saídas (além da condição inicial), de 250 em 250 intervalos de tempo Δt . Observe que para isto nós utilizamos uma lista (`v`), cujos elementos são `arrays`.

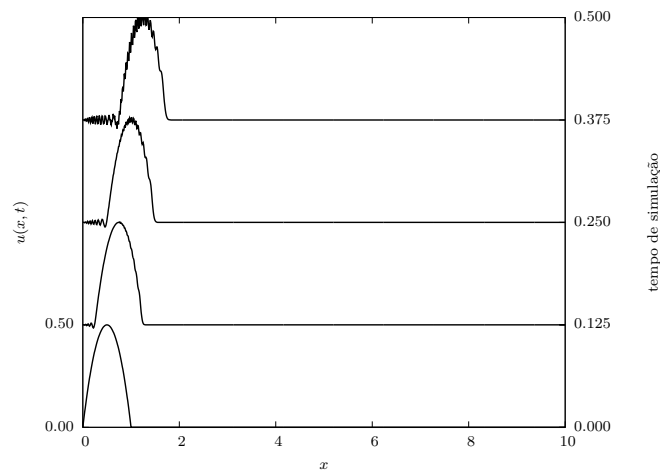
O resultado dos primeiros 750 intervalos de tempo de simulação é mostrado na figura 4.2. Repare como a solução se torna rapidamente instável. Repare também como a solução numérica, em $t = 750\Delta t = 0,375$, ainda está bastante distante dos tempos mostrados na solução analítica da figura 4.1 (que vão até $t = 4$). Claramente, o esquema explícito que nós programamos jamais nos levará a uma solução numérica satisfatória para tempos da ordem de $t = 1$!

Listagem 4.1: onda1d-ins.py — Solução de uma onda 1D com um método explícito instável

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # onda1d-ins resolve uma equação de onda com um método
5  # explícito
6  #
7  # uso: ./onda1d-ins.py
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 fou = open('onda1d-ins.dat','wb')
12 dx = 0.01
13 dt = 0.0005
14 print('#_dx=_%9.4f' % dx)
15 print('#_dt=_%9.4f' % dt)
16 from numpy import zeros
17 nx = int(10.0/dx)          # número de pontos em x
18 nt = int(1.0/dt)          # número de pontos em t
19 print('#_nx=_%9d' % nx)
20 print('#_nt=_%9d' % nt)
21 u = zeros((2,nx+1),float) # apenas 2 posições no tempo
22                             # são necessárias!
23                             # define a condição inicial
24 def CI(x):
25     if 0 <= x <= 1.0:
26         return 2.0*x*(1.0-x)
27     else:
28         return 0.0
29 for i in range(nx+1):      # monta a condição inicial
30     xi = i*dx
31     u[0,i] = CI(xi)
32 u[0].tofile(fou)          # imprime a condição inicial
33 old = False
34 new = True
35 c = 2.0                   # celeridade da onda
36 couhalf = c*dt/(2.0*dx)  # metade do número de Courant
37 for n in range(nt):       # loop no tempo
38     for i in range(1,nx): # loop no espaço
39         u[new,i] = u[old,i] - couhalf*(u[old,i+1] - u[old,i-1])
40     u[new,0] = 0.0
41     u[new,nx] = 0.0
42     u[new].tofile(fou)    # imprime uma linha com os novos dados
43     (old,new) = (new,old) # troca os índices
44 fou.close()

```

Figura 4.2: Solução numérica produzida por onda1d-ins.py, para $t = 250\Delta t$, $500\Delta t$ e $750\Delta t$.

Listagem 4.2: `surf1d-ins.py` — Selecciona alguns intervalos de tempo da solução numérica para plotagem

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # surf1d-ins.py: imprime em <arq> <m>+1 saídas de
5  # onda1d-ins a cada <n> intervalos de tempo
6  #
7  # uso: ./surf1d-ins.py <m> <n>
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 from sys import argv
12 dx = 0.01
13 dt = 0.0005
14 print('#_dx=_%9.4f' % dx)
15 print('#_dy=_%9.4f' % dt)
16 nx = int(10.0/dx)          # número de pontos em x
17 print('#_nx=_%9d' % nx)
18 m = int(argv[1])           # m saídas
19 n = int(argv[2])           # a cada n intervalos de tempo
20 print('#_m=_%9d' % m)
21 print('#_n=_%9d' % n)
22 fin = open('onda1d-ins.dat',
23            'rb')            # abre o arquivo com os dados
24 from numpy import fromfile
25 u = fromfile(fin,float,nx+1) # lê a condição inicial
26 v = [u]                    # inicializa a lista da "transposta"
27 for it in range(m):        # para <m> instantes:
28     for ir in range(n):    # lê <ir> vezes, só guarda a última
29         u = fromfile(fin,float,nx+1)
30         v.append(u)        # guarda a última
31 founam = 'surf1d-ins.dat'
32 print(founam)
33 fou = open(founam,'wt')     # abre o arquivo de saída
34 for i in range(nx+1):
35     fou.write('%10.6f' % (i*dx)) # escreve o "x"
36     fou.write('%10.6f' % v[0][i]) # escreve a cond inicial
37     for k in range(1,m+1):
38         fou.write('%10.6f' % v[k][i]) # escreve o k-ésimo
39     fou.write('\n')
40 fou.close()

```

Por que o esquema utilizado em (4.12) fracassa? Uma forma de obter a resposta é fazer uma *análise de estabilidade de von Neumann*.

A análise de estabilidade de von Neumann consiste primeiramente em observar que, em um computador real, (4.12) jamais será calculada com precisão infinita. O que o computador realmente calcula é um valor *truncado* \tilde{u}_i^n . Por enquanto, nós só vamos fazer esta distinção de notação, entre \tilde{u} e u , aqui, onde ela importa. O *erro de truncamento* é

$$\epsilon_i^n \equiv \tilde{u}_i^n - u_i^n. \quad (4.13)$$

Note que (4.12) se aplica tanto para u quanto para \tilde{u} ; subtraindo as equações resultantes para \tilde{u}_i^{n+1} e u_i^{n+1} , obtém-se a *mesma* equação para a evolução de ϵ_i^n :

$$\epsilon_i^{n+1} = \epsilon_i^n - \frac{\text{Co}}{2}(\epsilon_{i+1}^n - \epsilon_{i-1}^n), \quad (4.14)$$

onde

$$\text{Co} \equiv \frac{c\Delta t}{\Delta x} \quad (4.15)$$

é o *número de Courant*. Isto só foi possível porque (4.12) é uma equação *linear* em u . Mesmo para equações não-lineares, entretanto, sempre será possível fazer pelo menos uma análise *local* de estabilidade.

O próximo passo da análise de estabilidade de von Neumann é escrever uma série de Fourier para ϵ_i^n , na forma

$$\begin{aligned} t_n &= n\Delta t, \\ x_i &= i\Delta x, \\ \epsilon_i^n &= \sum_{l=1}^{N/2} \xi_l e^{at_n} e^{ik_l x_i}, \end{aligned} \quad (4.16)$$

onde e é a base dos logaritmos naturais, $i = \sqrt{-1}$, $N = L/\Delta x$ é o número de pontos da discretização em x , e L é o tamanho do domínio em x . Em (4.16), nós estamos supondo que o erro cresce exponencialmente.

Argumentando novamente com a linearidade, desta vez de (4.14), ela vale para cada *modo* l de (4.16), donde

$$\xi_l e^{a(t_n+\Delta t)} e^{ik_l i\Delta x} = \xi_l e^{at_n} e^{ik_l i\Delta x} - \frac{\text{Co}}{2} (\xi_l e^{at_n} e^{ik_l (i+1)\Delta x} - \xi_l e^{at_n} e^{ik_l (i-1)\Delta x}); \quad (4.17)$$

eliminando o fator comum $\xi_l e^{at_n + ik_l i\Delta x}$,

$$\begin{aligned} e^{a\Delta t} &= 1 - \frac{\text{Co}}{2} (e^{+ik_l \Delta x} - e^{-ik_l \Delta x}) \\ &= 1 - i\text{Co} \sin k_l \Delta x. \end{aligned} \quad (4.18)$$

O lado direito é um número complexo, de maneira que o lado esquerdo também tem que ser! Como conciliá-los? Fazendo $a = \alpha + i\beta$, e substituindo:

$$\begin{aligned} e^{(\alpha-i\beta)\Delta t} &= 1 - i\text{Co} \sin k_l \Delta x; \\ e^{\alpha\Delta t} [\cos(\beta\Delta t) - i \sin(\beta\Delta t)] &= 1 - i\text{Co} \sin k_l \Delta x; \Rightarrow \\ e^{\alpha\Delta t} \cos(\beta\Delta t) &= 1, \end{aligned} \quad (4.19)$$

$$e^{\alpha\Delta t} \sin(\beta\Delta t) = \text{Co} \sin(k_l \Delta x). \quad (4.20)$$

No capítulo sobre séries de Fourier, mostre que isto funciona, e discuta o $N/2$

As duas últimas equações formam um sistema não-linear nas incógnitas α β . O sistema pode ser resolvido:

$$\operatorname{tg}(\beta\Delta t) = \operatorname{Co sen}(k_l\Delta x) \Rightarrow \beta\Delta t = \operatorname{arctg}(\operatorname{Co sen}(k_l\Delta x)).$$

Note que $\beta \neq 0$, donde $e^{\alpha\Delta t} > 1$, e o esquema de diferenças finitas é *incondicionalmente instável*.

O método de Lax Uma alternativa que produz um esquema estável é o método de Lax (estamos seguindo os passos de *Numerical Recipes*):

$$u_i^{n+1} = \frac{1}{2} \left[(u_{i+1}^n + u_{i-1}^n) - \operatorname{Co}(u_{i+1}^n - u_{i-1}^n) \right]. \quad (4.21)$$

Agora que nós já sabemos que esquemas numéricos podem ser instáveis, devemos fazer uma análise de estabilidade *antes* de tentar implementar (4.21) numericamente. Vamos a isto: utilizando novamente (4.16) e substituindo em (4.21), temos

$$\begin{aligned} \xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} &= \frac{1}{2} \left[\left(\xi_l e^{at_n} e^{ik_l(i+1)\Delta x} + \xi_l e^{at_n} e^{ik_l(i-1)\Delta x} \right) \right. \\ &\quad \left. - \operatorname{Co} \left(\xi_l e^{at_n} e^{ik_l(i+1)\Delta x} - \xi_l e^{at_n} e^{ik_l(i-1)\Delta x} \right) \right]; \\ e^{a\Delta t} &= \frac{1}{2} \left[\left(e^{+ik_l\Delta x} + e^{-ik_l\Delta x} \right) - \operatorname{Co} \left(e^{+ik_l\Delta x} - e^{-ik_l\Delta x} \right) \right]; \\ e^{a\Delta t} &= \cos(k_l\Delta x) - i \operatorname{Co sen}(k_l\Delta x) \end{aligned} \quad (4.22)$$

Nós podemos, é claro, fazer $a = \alpha - i\beta$, mas há um caminho mais rápido: o truque é perceber que se o fator de amplificação $e^{a\Delta t}$ for um número complexo com módulo maior que 1, o esquema será instável. Desejamos, portanto, que $|e^{a\Delta t}| \leq 1$, o que só é possível se

$$\operatorname{Co} \leq 1, \quad (4.23)$$

que é o critério de estabilidade de Courant-Friedrichs-Lewy.

A “mágica” de (4.21) é que ela introduz um pouco de *difusão numérica*; de fato, podemos reescrevê-la na forma

$$\begin{aligned} \frac{u_i^{n+1} - u_i^n}{\Delta t} &= -c \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} + \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{2\Delta t} \\ &= -c \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} + \left(\frac{\Delta x^2}{2\Delta t} \right) \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}. \end{aligned} \quad (4.24)$$

Não custa repetir: (4.24) é *idêntica* a (4.21). Porém, comparando-a com (4.12) (nosso esquema instável inicialmente empregado), nós vemos que ela também é equivalente a esta última, *com o termo adicional* $(\Delta x^2/2\Delta t)(u_{i+1}^n - 2u_i^n + u_{i-1}^n)/\Delta x^2$. O que este termo adicional significa? A resposta é *uma derivada numérica de ordem 2*. De fato, considere as expansões em série de Taylor

$$\begin{aligned} u_{i+1} &= u_i + \frac{du}{dx} \Big|_i \Delta x + \frac{1}{2} \frac{d^2u}{dx^2} \Big|_i \Delta x^2 + O(\Delta x^3), \\ u_{i-1} &= u_i - \frac{du}{dx} \Big|_i \Delta x + \frac{1}{2} \frac{d^2u}{dx^2} \Big|_i \Delta x^2 + O(\Delta x^3), \end{aligned}$$

e some:

$$\begin{aligned} u_{i+1} + u_{i-1} &= 2u_i + \left. \frac{d^2 u}{dx^2} \right|_i \Delta x^2 + O(\Delta x^2), \\ \left. \frac{d^2 u}{dx^2} \right|_i &= \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O(\Delta x^2). \end{aligned} \quad (4.25)$$

Portanto, a equação (4.24) — ou seja: o esquema de Lax (4.21) — pode ser interpretada *também* como uma solução aproximada da equação de advecção-difusão

$$\frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = D \frac{\partial^2 u}{\partial x^2},$$

com

$$D = \left(\frac{\Delta x^2}{2\Delta t} \right).$$

Note que D tem dimensões de *difusividade*: $\llbracket D \rrbracket = \mathbb{L}^2 \mathbb{T}^{-1}$. No entanto: não estamos então resolvendo a equação *errada*? De certa forma, sim: estamos introduzindo um pouco de difusão na equação para amortecer as oscilações que aparecerão em decorrência da amplificação dos erros de truncamento.

O quanto isto nos prejudica? Não muito, *desde que o efeito da difusão seja muito menor que o da advecção que estamos tentando simular*. Como a velocidade de advecção (“física”; “real”) que estamos simulando é c , precisamos comparar isto com (por exemplo) a magnitude das velocidades introduzidas pela difusão numérica; devemos portanto verificar se

$$\begin{aligned} \frac{D \frac{\partial^2 u}{\partial x^2}}{c \frac{\partial u}{\partial x}} &\ll 1, \\ \frac{D \frac{u}{\Delta x^2}}{c \frac{u}{\Delta x}} &\ll 1, \\ \frac{D}{\Delta x} &\ll c, \\ \frac{\Delta x^2}{2\Delta t \Delta x} &\ll c, \\ \frac{c \Delta t}{\Delta x} = \text{Co} &\gg \frac{1}{2} \end{aligned}$$

Em outras palavras, nós descobrimos que o critério para que o esquema seja acurado do ponto de vista físico é conflitante com o critério de estabilidade: enquanto que estabilidade demandava $\text{Co} < 1$, o critério de que a solução seja *também* fisicamente acurada demanda que $\text{Co} \gg 1/2$. Na prática, isto significa que, para $c = 2$, ou o esquema é estável com muita difusão numérica, ou ele é instável. Isto praticamente elimina a possibilidade de qualquer uso sério de (4.21).

Mesmo assim, vamos programá-lo! O programa `onda1d-lax.py` está mostrado na listagem 4.3. Ele usa os mesmos valores $\Delta t = 0,0005$ e $\Delta x = 0,01$, ou seja, $\text{Co} = 0,10$.

O programa gera um arquivo de saída binário, que por sua vez é lindo pelo próximo programa na sequência, `surf1d-lax.py`, mostrado na listagem 4.4. O único trabalho deste programa é selecionar algumas “linhas” da saída de `onda1d-lax.py`;

Listagem 4.3: onda1d-lax.py — Solução de uma onda 1D com um método explícito laxtável

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # onda1d-lax resolve uma equação de onda com um método
5  # explícito
6  #
7  # uso: ./onda1d-ins.py
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 fou = open('onda1d-lax.dat','wb')
12 dx = 0.01
13 dt = 0.0005
14 print('#_dx=_%9.4f' % dx)
15 print('#_dy=_%9.4f' % dt)
16 from numpy import zeros
17 nx = int(10.0/dx)          # número de pontos em x
18 nt = int(1.0/dt)          # número de pontos em t
19 print('#_nx=_%9d' % nx)
20 print('#_nt=_%9d' % nt)
21 u = zeros((2,nx+1),float) # apenas 2 posições no tempo
22                             # são necessárias!
23 def CI(x):                 # define a condição inicial
24     if 0 <= x <= 1.0:
25         return 2.0*x*(1.0-x)
26     else:
27         return 0.0
28 for i in range(nx+1):      # monta a condição inicial
29     xi = i*dx
30     u[0,i] = CI(xi)
31 u[0].tofile(fou)          # imprime a condição inicial
32 old = False
33 new = True
34 c = 2.0                   # celeridade da onda
35 cou = c*dt/(dx)           # número de Courant
36 print("Co=_%10.6f" % cou)
37 for n in range(nt):       # loop no tempo
38     for i in range(1,nx): # loop no espaço
39         u[new,i] = 0.5*( (u[old,i+1] + u[old,i-1]) -
40                           cou*(u[old,i+1] - u[old,i-1]) )
41     u[new,0] = 0.0
42     u[new,nx] = 0.0
43     u[new].tofile(fou)     # imprime uma linha com os novos dados
44     (old,new) = (new,old)  # troca os índices
45 fou.close()

```

Listagem 4.4: `surfid-lax.py` — Selecciona alguns intervalos de tempo da solução numérica para plotagem

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # surfid-lax.py: imprime em <arq> <m>+1 saídas de
5  # onda1d-lax a cada <n> intervalos de tempo
6  #
7  # uso: ./surfid-lax.py <m> <n>
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 from sys import argv
12 dx = 0.01
13 dt = 0.0005
14 print('#_dx=_%9.4f' % dx)
15 print('#_dy=_%9.4f' % dt)
16 nx = int(10.0/dx)          # número de pontos em x
17 print('#_nx=_%9d' % nx)
18 m = int(argv[1])           # m saídas
19 n = int(argv[2])           # a cada n intervalos de tempo
20 print('#_m=_%9d' % m)
21 print('#_n=_%9d' % n)
22 fin = open('onda1d-lax.dat',
23           'rb')             # abre o arquivo com os dados
24 from numpy import fromfile
25 u = fromfile(fin,float,nx+1) # lê a condição inicial
26 v = [u]                    # inicializa a lista da "transposta"
27 for it in range(m):
28     for ir in range(n):
29         u = fromfile(fin,float,nx+1)
30         v.append(u)          # guarda a última
31 founam = 'surfid-lax.dat'
32 print(founam)
33 fou = open(founam,'wt')     # abre o arquivo de saída
34 for i in range(nx+1):
35     fou.write('%10.6f' % (i*dx)) # escreve o "x"
36     fou.write('%10.6f' % v[0][i]) # escreve a cond inicial
37     for k in range(1,m+1):
38         fou.write('%10.6f' % v[k][i]) # escreve o k-ésimo
39     fou.write('\n')
40 fou.close()

```

no caso, nós o rodamos com o comando

`[onda1d-lax.py 3 500]`,

o que significa seleccionar 3 saídas (além da condição inicial), de 500 em 500 intervalos de tempo Δt . Com isto, nós conseguimos chegar até o instante 0,75 da simulação.

O resultado dos primeiros 1500 intervalos de tempo de simulação é mostrado na figura 4.3. Observe que agora não há oscilações espúrias: o esquema é estável no tempo. No entanto, a solução está agora “amortecida” pela difusão numérica!

Upwind Um esquema que é conhecido na literatura como indicado por representar melhor o termo advectivo em (4.1) é o esquema de diferenças regressivas; neste esquema, chamado de esquema *upwind* — literalmente, “corrente acima” — na literatura de língua inglesa, a discretização utilizada é

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -c \frac{u_i^n - u_{i-1}^n}{\Delta x},$$

$$u_i^{n+1} = u_i^n - \text{Co} [u_i^n - u_{i-1}^n]. \quad (4.26)$$

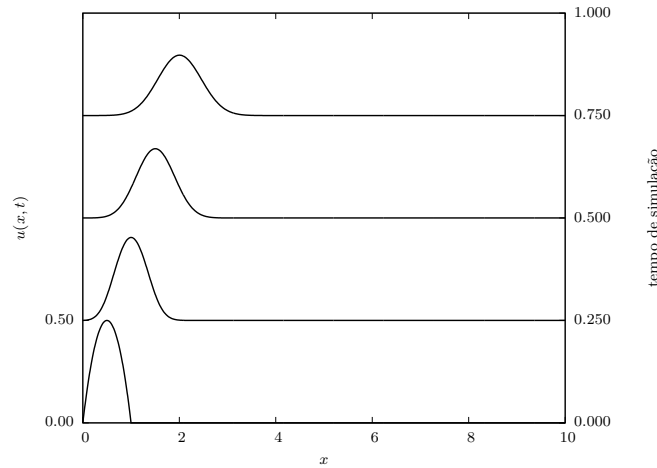


Figura 4.3: Solução numérica produzida por `onda1d-lax.py`, para $t = 500\Delta t$, $1000\Delta t$ e $1500\Delta t$.

Claramente, estamos utilizando um esquema de $O(\Delta x)$ para a derivada espacial. Ele é um esquema menos acurado que os usados anteriormente, mas se ele ao mesmo tempo for condicionalmente estável e não introduzir difusão numérica, o resultado pode ser melhor para tratar a advecção.

Antes de “colocarmos as mãos na massa”, sabemos que devemos analisar analiticamente a estabilidade do esquema. Vamos a isto:

$$\begin{aligned}
 \xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} - \text{Co} [\xi_l e^{at_n} e^{ik_l i \Delta x} - \xi_l e^{at_n} e^{ik_l (i-1) \Delta x}] \\
 e^{a\Delta t} e^{ik_l i \Delta x} &= e^{ik_l i \Delta x} - \text{Co} [e^{ik_l i \Delta x} - e^{ik_l (i-1) \Delta x}] \\
 e^{a\Delta t} &= 1 - \text{Co} [1 - e^{-ik_l \Delta x}] \\
 e^{a\Delta t} &= 1 - \text{Co} + \text{Co} \cos(k_l \Delta x) - i \text{Co} \sin(k_l \Delta x). \tag{4.27}
 \end{aligned}$$

Desejamos que o módulo do fator de amplificação $e^{a\Delta t}$ seja menor que 1. O módulo (ao quadrado) é

$$|e^{a\Delta t}|^2 = (1 - \text{Co} + \text{Co} \cos(k_l \Delta x))^2 + (\text{Co} \sin(k_l \Delta x))^2.$$

Para aliviar a notação, façamos

$$\begin{aligned}
 C_k &\equiv \cos(k_l \Delta x), \\
 S_k &\equiv \sin(k_l \Delta x).
 \end{aligned}$$

Então,

$$\begin{aligned}
 |e^{a\Delta t}|^2 &= (\text{Co} S_k)^2 + (\text{Co} C_k - \text{Co} + 1)^2 \\
 &= \text{Co}^2 S_k^2 + (\text{Co}^2 C_k^2 + \text{Co}^2 + 1) + 2(-\text{Co}^2 C_k + \text{Co} C_k - \text{Co}) \\
 &= \text{Co}^2 (S_k^2 + C_k^2 + 1 - 2C_k) + 2\text{Co}(C_k - 1) + 1 \\
 &= 2\text{Co}^2(1 - C_k) + 2\text{Co}(C_k - 1) + 1.
 \end{aligned}$$

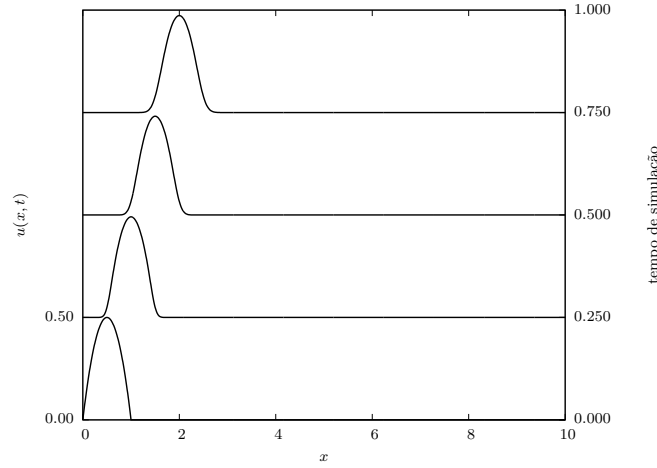


Figura 4.4: Solução numérica produzida pelo esquema *upwind*, para $t = 500\Delta t$, $1000\Delta t$ e $1500\Delta t$.

A condição para que o esquema de diferenças finitas seja estável é, então,

$$\begin{aligned} 2Co^2(1 - C_k) + 2Co(C_k - 1) + 1 &\leq 1, \\ 2Co[Co(1 - C_k) + (C_k - 1)] &\leq 0, \\ (1 - \cos(k_l \Delta x)) [Co - 1] &\leq 0, \\ Co &\leq 1 \blacksquare \end{aligned}$$

Reencontramos, portanto, a condição (4.23), *mas em um outro esquema de diferenças finitas*. A lição não deve ser mal interpretada: longe de supor que (4.23) vale sempre, é a análise de estabilidade que deve refeita para cada novo esquema de diferenças finitas!

O esquema *upwind*, portanto, é condicionalmente estável, e tudo indica que podemos agora implementá-lo computacionalmente, e ver no que ele vai dar. Nós utilizamos os mesmos valores de Δt e de Δx de antes. As mudanças necessárias nos códigos computacionais são óbvias, e são deixadas a cargo do(a) leitor(a).

A figura 4.4 mostra o resultado do esquema *upwind*. Note que ele é *muito melhor* (para *esta* equação diferencial) que o esquema de Lax. No entanto, a figura sugere que algum amortecimento (difusão numérica?) também está ocorrendo, embora em grau muito menor.

Exercícios Propostos

4.1 Escreva o programa `onda1d-upw` e `surfa1d-upw`, que implementam o esquema *upwind*. Reproduza a figura 4.4.

4.2 Calcule a difusividade numérica introduzida pelo esquema *upwind*.

4.2 – Difusão pura

Considere agora a equação da difusão,

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}, \quad (4.28)$$

com condições iniciais e de contorno

$$u(x, 0) = f(x) \quad (4.29)$$

$$u(0, t) = u(L, t) = 0. \quad (4.30)$$

Esta solução foi vista no capítulo ??:

$$u(x, t) = \sum_{n=1}^{\infty} A_n e^{-\frac{n^2 \pi^2 \alpha^2}{L^2} t} \operatorname{sen} \frac{n\pi x}{L}, \quad (4.31)$$

$$A_n = \frac{2}{L} \int_0^L f(x) \operatorname{sen} \frac{n\pi x}{L} dx. \quad (4.32)$$

Em particular, se

$$\begin{aligned} D &= 2, \\ L &= 1, \\ f(x) &= 2x(1-x), \\ A_n &= 2 \int_0^1 2x(1-x) \operatorname{sen}(n\pi x) dx = \frac{8}{\pi^3 n^3} [1 - (-1)^n]. \end{aligned}$$

Todos os A_n 's pares se anulam. Fique então apenas com os ímpares:

$$\begin{aligned} A_{2n+1} &= \frac{16}{\pi^3 (2n+1)^3}, \\ u(x, t) &= \sum_{n=0}^{\infty} \frac{16}{\pi^3 (2n+1)^3} e^{-(2(2n+1)^2 \pi^2) t} \operatorname{sen}((2n+1)\pi x) \end{aligned} \quad (4.33)$$

O programa `difusao1d-ana.py`, mostrado na listagem 4.5, implementa a solução analítica para $\Delta t = 0,0005$ e $\Delta x = 0,001$.

Da mesma maneira que os programas `surf1d*.py`, o programa `divisao1d-ana.py`, mostrado na listagem 4.6, seleciona alguns instantes de tempo da solução analítica para visualização.

A figura 4.5 mostra o resultado da solução numérica para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Este é praticamente o “fim” do processo difusivo, com a solução analítica tendendo rapidamente para zero.

Esquema explícito Talvez o esquema explícito mais óbvio para discretizar (4.28) seja

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = D \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}. \quad (4.34)$$

A derivada parcial em relação ao tempo é de $O(\Delta t)$, enquanto que a derivada segunda parcial em relação ao espaço é, como vimos em (4.25), de $O(\Delta x^2)$. Mas não nos preocupemos muito, ainda, com a acurácia do esquema numérico. Nossa primeira preocupação, como você já sabe, é outra: o esquema (4.34) é *estável*?

Explicitamos u_i^{n+1} em (4.34):

$$u_i^{n+1} = u_i^n + \operatorname{Fo} [u_{i+1}^n - 2u_i^n + u_{i-1}^n], \quad (4.35)$$

onde

$$\operatorname{Fo} = \frac{D\Delta t}{\Delta x^2} \quad (4.36)$$

Listagem 4.5: difusao1d-ana.py — Solução analítica da equação da difusão

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # difusao1d-ana: solução analítica de
5  #
6  #  $du/dt = D \, du^2/dx^2$ 
7  #
8  #  $u(x,0) = 2x(1-x)$ 
9  #  $u(0,t) = 0$ 
10 #  $u(1,t) = 0$ 
11 #
12 # uso: ./difusao1d-ana.py
13 # -----
14 from __future__ import print_function
15 from __future__ import division
16 fou = open('difusao1d-ana.dat','wb')
17 dx = 0.001
18 dt = 0.0005
19 print('#_dx=_%9.4f' % dx)
20 print('#_dy=_%9.4f' % dt)
21 nx = int(1.0/dx)          # número de pontos em x
22 nt = int(1.0/dt)          # número de pontos em t
23 print('#_nx=_%9d' % nx)
24 print('#_nt=_%9d' % nt)
25 from math import pi, sin, exp
26 epsilon = 1.0e-6          # precisão da solução analítica
27 dpiq = 2*pi*pi            #  $2\pi^2$ 
28 dzpic = 16/(pi*pi*pi)     #  $16/\pi^3$ 
29 def ana(x,t):
30     s = 0.0
31     ds = epsilon
32     n = 0
33     while abs(ds) >= epsilon:
34         dnm1 = 2*n + 1      #  $(2n+1)$ 
35         dnm1q = dnm1*dnm1   #  $(2n+1)^2$ 
36         dnm1c = dnm1q*dnm1  #  $(2n+1)^3$ 
37         ds = exp(-dnm1q*dpiq*t)
38         ds *= sin(dnm1*pi*x)
39         ds /= dnm1c
40         s += ds
41         n += 1
42     return s*dzpic
43 from numpy import zeros
44 u = zeros(nx+1,float)      # um array para conter a solução
45 for n in range(nt+1):      # loop no tempo
46     t = n*dt
47     print(t)
48     for i in range(nx+1):  # loop no espaço
49         xi = i*dx
50         u[i] = ana(xi,t)
51     u.tofile(fou)          # imprime uma linha com os novos dados
52 fou.close()

```

Listagem 4.6: `divisao1d-ana.py` — Selecciona alguns instantes de tempo da solução analítica para visualização

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # divisao1d-ana.py: imprime em <arq> <m>+1 saídas de
5  # difusao1d-ana a cada <n> intervalos de tempo
6  #
7  # uso: ./divisao1d-ana.py <m> <n>
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 from sys import argv
12 dx = 0.001
13 dt = 0.0005
14 print('#_dx=_%9.4f' % dx)
15 print('#_dt=_%9.4f' % dt)
16 nx = int(1.0/dx)
17 print('#_nx=_%9d' % nx)      # número de pontos em x
18 m = int(argv[1])
19 n = int(argv[2])
20 print('#_m=_%9d' % m)      # m saídas
21 print('#_n=_%9d' % n)      # a cada n intervalos de tempo
22 fin = open('difusao1d-ana.dat',
23            'rb')           # abre o arquivo com os dados
24 from numpy import fromfile
25 u = fromfile(fin,float,nx+1) # lê a condição inicial
26 v = [u]
27 for it in range(m):
28     for ir in range(n):
29         u = fromfile(fin,float,nx+1)
30         v.append(u)          # guarda a última
31 founam = 'divisao1d-ana.dat'
32 print(founam)
33 fou = open(founam,'wt')     # abre o arquivo de saída
34 for i in range(nx+1):
35     fou.write('%10.6f' % (i*dx)) # escreve o "x"
36     fou.write('%10.6f' % v[0][i]) # escreve a cond inicial
37     for k in range(1,m+1):
38         fou.write('%10.6f' % v[k][i]) # escreve o k-ésimo
39     fou.write('\n')
40 fou.close()

```

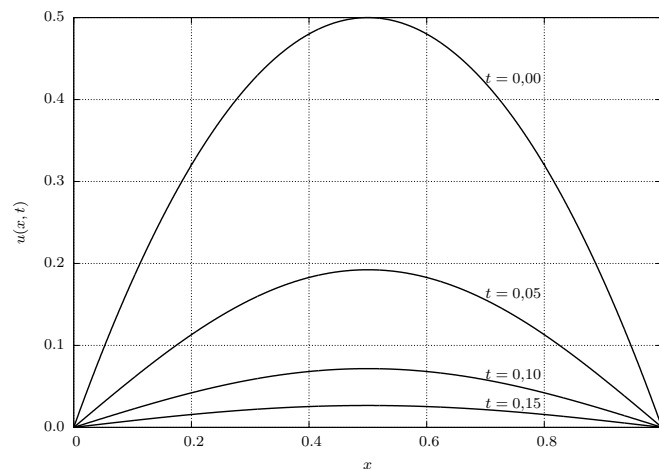


Figura 4.5: Solução analítica da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$.

é o *número de Fourier de grade* (El-Kadi e Ling, 1993). A análise de estabilidade de von Neumann agora produz

$$\begin{aligned}
 \xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} + \\
 &\quad \text{Fo} \left[\xi_l e^{at_n} e^{ik_l (i+1) \Delta x} - 2\xi_l e^{at_n} e^{ik_l i \Delta x} + \xi_l e^{at_n} e^{ik_l (i-1) \Delta x} \right], \\
 e^{a\Delta t} &= 1 + \text{Fo} \left[e^{+ik_l \Delta x} - 2 + e^{-ik_l \Delta x} \right] \\
 &= 1 + 2\text{Fo} [\cos(k_l \Delta x) - 1] \\
 &= 1 - 4\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right)
 \end{aligned} \tag{4.37}$$

A análise de estabilidade requer que $|e^{a\Delta t}| < 1$:

$$|e^{a\Delta t}|^2 = 1 - 8\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right) + 16\text{Fo}^2 \sin^4 \left(\frac{k_l \Delta x}{2} \right) < 1$$

ou

$$\begin{aligned}
 -8\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right) + 16\text{Fo}^2 \sin^4 \left(\frac{k_l \Delta x}{2} \right) &< 0, \\
 8\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right) \left[-1 + 2\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right) \right] &< 0, \\
 \text{Fo} &< \frac{1}{2}.
 \end{aligned} \tag{4.38}$$

Podemos agora calcular o número de Fourier que utilizamos para plotar a solução analítica (verifique nas listagens 4.5 e 4.6):

$$\text{Fo} = \frac{2 \times 0,0005}{(0,001)^2} = 1000.$$

Utilizar os valores $\Delta x = 0,0005$ e $\Delta x = 0,001$ levaria a um esquema instável. Precisamos *diminuir* Δt e/ou *aumentar* Δx . Com $\Delta t = 0,00001$ e $\Delta x = 0,01$,

$$\text{Fo} = \frac{2 \times 0,00001}{(0,01)^2} = 0,2 < 0,5 \quad (\text{OK}).$$

Repare que $\text{Fo} < 1/2$ é um critério de estabilidade muito mais exigente do que $\text{Co} < 1/2$ (para $D = 2$). Nós esperamos que nosso esquema explícito agora rode muito lentamente. Mas vamos implementá-lo. O programa que implementa o esquema é o `difusao1d-exp.py`, mostrado na listagem 4.7.

O programa `divisao1d-exp.py`, mostrado na listagem 4.8, seleciona alguns instantes de tempo da solução analítica para visualização.

O resultado da solução numérica com o método explícito está mostrado na figura 4.6: ele é impressionantemente bom, embora seja computacionalmente muito caro. A escolha judiciosa de Δt e Δx para obedecer ao critério (4.38) foi fundamental para a obtenção de um bom resultado “de primeira”, sem a necessidade dolorosa de ficar tentando diversas combinações até que o esquema se estabilize e produza bons resultados.

Listagem 4.7: `difusao1d-exp.py` — Solução numérica da equação da difusão: método explícito.

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # difusao1d-exp resolve uma equação de difusão com um método
5  # explícito
6  #
7  # uso: ./difusao1d-exp.py
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 fou = open('difusao1d-exp.dat','wb')
12 dx = 0.01
13 dt = 0.00001
14 print('#_dx=_%9.4f' % dx)
15 print('#_dy=_%9.4f' % dt)
16 from numpy import zeros
17 nx = int(round(1.0/dx,0))      # número de pontos em x
18 nt = int(round(1.0/dt,0))      # número de pontos em t
19 print('#_nx=_%9d' % nx)
20 print('#_nt=_%9d' % nt)
21 u = zeros((2,nx+1),float)     # apenas 2 posições no tempo
22                                # são necessárias!
23 def CI(x):                     # define a condição inicial
24     if 0 <= x <= 1.0:
25         return 2.0*x*(1.0-x)
26     else:
27         return 0.0
28 for i in range(nx+1):          # monta a condição inicial
29     xi = i*dx
30     u[0,i] = CI(xi)
31 u[0].tofile(fou)               # imprime a condição inicial
32 old = False
33 new = True
34 D = 2.0                        # celeridade da onda
35 Fon = D*dt/((dx)**2)           # número de Fourier
36 print("Fo=_%10.6f" % Fon)
37 for n in range(nt):           # loop no tempo
38     print(n)
39     for i in range(1,nx):      # loop no espaço
40         u[new,i] = u[old,i] + Fon*(u[old,i+1] - 2*u[old,i] + u[old,i-1])
41         u[new,0] = 0.0         # condição de contorno, x = 0
42         u[new,nx] = 0.0        # condição de contorno, x = 1
43         u[new].tofile(fou)     # imprime uma linha com os novos dados
44         (old,new) = (new,old)  # troca os índices
45 fou.close()

```

Listagem 4.8: `divisao1d-exp.py` — Selecciona alguns instantes de tempo da solução analítica para visualização

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # divisao1d-exp.py: imprime em <arq> <m>+1 saídas de
5  # difusao1d-exp a cada <n> intervalos de tempo
6  #
7  # uso: ./divisao1d-exp.py <m> <n>
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 from sys import argv
12 dx = 0.01
13 dt = 0.00001
14 print('#_dx=_%9.4f' % dx)
15 print('#_dt=_%9.4f' % dt)
16 nx = int(round(1.0/dx,0))      # número de pontos em x
17 nt = int(round(1.0/dt,0))      # número de pontos em t
18 print('#_nx=_%9d' % nx)
19 m = int(argv[1])               # m saídas
20 n = int(argv[2])               # a cada n intervalos de tempo
21 print('#_m=_%9d' % m)
22 print('#_n=_%9d' % n)
23 fin = open('difusao1d-exp.dat',
24           'rb')                # abre o arquivo com os dados
25 from numpy import fromfile
26 u = fromfile(fin,float,nx+1)   # lê a condição inicial
27 v = [u]                       # inicializa a lista da "transposta"
28 for it in range(m):           # para <m> instantes:
29     for ir in range(n):       # lê <ir> vezes, só guarda a última
30         u = fromfile(fin,float,nx+1)
31         v.append(u)           # guarda a última
32 founam = 'divisao1d-exp.dat'
33 fou = open(founam,'wt')        # abre o arquivo de saída
34 for i in range(nx+1):
35     fou.write('%10.6f' % (i*dx)) # escreve o "x"
36     fou.write('%10.6f' % v[0][i]) # escreve a cond inicial
37     for k in range(1,m+1):
38         fou.write('%10.6f' % v[k][i]) # escreve o k-ésimo
39     fou.write('\n')
40 fou.close()

```

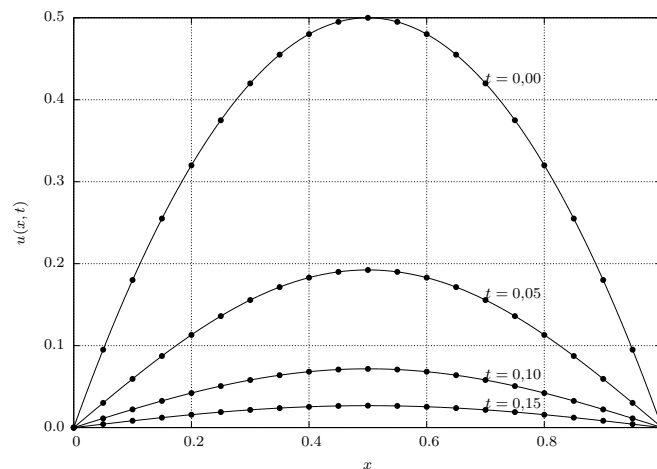


Figura 4.6: Solução numérica com o método explícito (4.35) (círculos) *versus* a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

Esquemas implícitos Embora o esquema explícito que nós utilizamos acima seja acurado, ele é *lento* — se você programou e rodou `difusao1d-exp.py`, deve ter notado *alguma* demora para o programa rodar. Embora nossos computadores estejam ficando a cada dia mais rápidos, isto não é desculpa para utilizar mal nossos recursos computacionais (é claro que, ao utilizarmos uma linguagem interpretada — Python — para programar, nós *já* estamos utilizando muito mal nossos recursos; no entanto, nosso argumento é didático: com uma linguagem mais simples, podemos aprender mais rápido e errar menos. Além disto, todos os ganhos *relativos* que obtivermos se manterão em qualquer outra linguagem)

Vamos portanto fazer uma mudança fundamental nos nossos esquemas de diferenças finitas: vamos calcular a derivada espacial no instante $n + 1$:

$$\begin{aligned} \frac{u_i^{n+1} - u_i^n}{\Delta t} &= D \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}, \\ u_i^{n+1} - u_i^n &= \text{Fo}(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}), \\ -\text{Fo}u_{i-1}^{n+1} + (1 + 2\text{Fo})u_i^{n+1} - \text{Fo}u_{i+1}^{n+1} &= u_i^n \end{aligned} \quad (4.39)$$

Reveja a discretização (4.5)–(4.9): para $i = 1, \dots, N_x - 1$, (4.39) acopla 3 valores das incógnitas u^{n+1} no instante $n + 1$. Quando $i = 0$, e quando $i = N_x$, não podemos utilizar (4.39), porque não existem os índices $i = -1$, e $i = N_x + 1$. Quando $i = 1$ e $i = N_x - 1$, (4.39) precisa ser modificada, para a introdução das *condições de contorno*: como $u_0^n = 0$ e $u_{N_x}^n = 0$ para qualquer n , teremos

$$(1 + 2\text{Fo})u_1^{n+1} - \text{Fo}u_2^{n+1} = u_1^n, \quad (4.40)$$

$$-\text{Fo}u_{N_x-2}^{n+1} + (1 + 2\text{Fo})u_{N_x-1}^{n+1} = u_{N_x-1}^n. \quad (4.41)$$

Em resumo, nossas incógnitas são $u_1^{n+1}, u_2^{n+1}, \dots, u_{N_x-1}^{n+1}$ ($N_x - 1$ incógnitas), e seu cálculo envolve a solução do sistema de equações

$$\begin{bmatrix} 1 + 2\text{Fo} & -\text{Fo} & 0 & \dots & 0 & 0 \\ -\text{Fo} & 1 + 2\text{Fo} & \text{Fo} & 0 & \dots & 0 \\ \vdots & & & & & \vdots \\ 0 & \dots & 0 & -\text{Fo} & 1 + 2\text{Fo} & -\text{Fo} \\ 0 & 0 & \dots & 0 & -\text{Fo} & 1 + 2\text{Fo} \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ \vdots \\ u_{N_x-2}^{n+1} \\ u_{N_x-1}^{n+1} \end{bmatrix} = \begin{bmatrix} u_1^n \\ u_2^n \\ \vdots \\ u_{N_x-2}^n \\ u_{N_x-1}^n \end{bmatrix} \quad (4.42)$$

A análise de estabilidade de von Neumann procede agora da maneira usual:

$$\begin{aligned}
\epsilon_i^{n+1} &= \epsilon_i^n + \text{Fo}(\epsilon_{i+1}^{n+1} - 2\epsilon_i^{n+1} + \epsilon_{i-1}^{n+1}) \\
\xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} \\
&\quad + \text{Fo} \left(\xi_l e^{a(t_n+\Delta t)} e^{ik_l(i+1)\Delta x} - 2\xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} \right. \\
&\quad \left. + \xi_l e^{a(t_n+\Delta t)} e^{ik_l(i-1)\Delta x} \right), \\
e^{a\Delta t} &= 1 + e^{a\Delta t} \text{Fo} \left(e^{ik_l \Delta x} - 2 + e^{-ik_l \Delta x} \right), \\
e^{a\Delta t} &= 1 + e^{a\Delta t} 2\text{Fo} (\cos(k_l \Delta x) - 1), \\
e^{a\Delta t} &= 1 - e^{a\Delta t} 4\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right), \\
e^{a\Delta t} \left[1 + 4\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right) \right] &= 1, \\
|e^{a\Delta t}| &= \frac{1}{1 + 4\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right)} \leq 1 \quad \text{sempre.} \tag{4.43}
\end{aligned}$$

Portanto, o esquema implícito (4.39) é incondicionalmente estável, e temos confiança de que o programa correspondente não se instabilizará.

Existem várias coisas atraentes para um programador em (4.42). Em primeiro lugar, a matriz do sistema é uma matriz banda tridiagonal; sistemas lineares com este tipo de matriz são particularmente simples de resolver, e estão disponíveis na literatura (por exemplo: Press et al., 1992, seção 2.4, subrotina `tridag`). Em segundo lugar, a matriz do sistema é *constante*: ela só precisa ser montada uma vez no programa, o que torna a solução numérica potencialmente muito rápida.

Nós vamos começar, então, construindo um pequeno módulo, convenientemente denominado `alglin.py`, que exporta a função `tridag`, que resolve um sistema tridiagonal, mostrado na listagem 4.9.

Em seguida, o programa `difusao1d-imp.py` resolve o problema com o método implícito. Ele está mostrado na listagem 4.10. A principal novidade está nas linhas 42–46, e depois novamente na linha 56. Em Python e Numpy, é possível especificar sub-listas, e sub-arrays, com um dispositivo denominado *slicing*, que torna a programação mais compacta e clara. Por exemplo, na linha 43, todos os elementos `A[0,1]...A[0,nx-1]` recebem o valor `-Fon`.

Existe um programa `divisao1d-imp.py`, mas ele não precisa ser mostrado aqui, porque as modificações, por exemplo a partir de `divisao1d-exp.py`, são demasiadamente triviais para justificarem o gasto adicional de papel. Para $\Delta t = 0,001$, e $\Delta x = 0,01$, o resultado do método implícito está mostrado na figura 4.7

Nada mal, para uma economia de 100 vezes (em relação ao método explícito) em passos de tempo! (Note entretanto que a solução, em cada passo de tempo, é um pouco mais custosa, por envolver a solução de um sistema de equações acopladas, ainda que tridiagonal.)

Crank Nicholson A derivada espacial em (4.28) é aproximada, no esquema implícito (4.39), por um esquema de $O(\Delta x^2)$. A derivada temporal, por sua vez, é

Listagem 4.9: `alglin.py` — Exporta uma rotina que resolve um sistema tridiagonal, baseado em [Press et al. \(1992\)](#)

```

1  # -*- coding: iso-8859-1 -*-
2  # -----
3  # alglin.py implementa uma solução de um sistema linear com matriz tridiagonal
4  # -----
5  from numpy import zeros
6  def tridag(A,y):
7      m = A.shape[0]
8      n = A.shape[1]
9      assert(m == 3)
10     o = y.shape[0]
11     assert (n == o)
12     x = zeros(n,float)
13     gam = zeros(n,float)
14     if A[1,0] == 0.0 :
15         exit("Erro 1 em tridag")
16     bet = A[1,0]
17     x[0] = y[0]/bet
18     for j in range(1,n):
19         gam[j] = A[2,j-1]/bet
20         bet = A[1,j] - A[0,j]*gam[j]
21         if (bet == 0.0):
22             exit("Erro 2 em tridag")
23         x[j] = (y[j] - A[0,j]*x[j-1])/bet
24     for j in range(n-2,-1,-1):
25         x[j] -= gam[j+1]*x[j+1]
26     return x

```

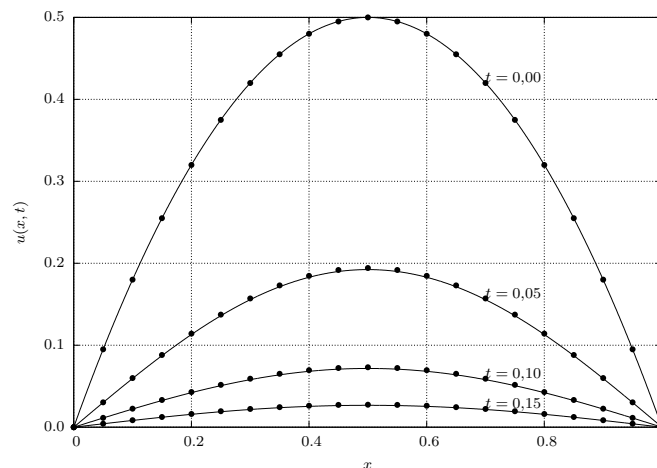


Figura 4.7: Solução numérica com o método implícito (4.39) (círculos) *versus* a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

Listagem 4.10: `difusao1d-imp.py` — Solução numérica da equação da difusão: método implícito.

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # difusao1d-imp resolve uma equação de difusão com um método
5  # implícito
6  #
7  # uso: ./difusao1d-imp.py
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 fou = open('difusao1d-imp.dat','wb')
12 dx = 0.01                # define a discretização em x
13 dt = 0.001               # define a discretização em t
14 print('#_dx_=%9.4f' % dx)
15 print('#_dt_=%9.4f' % dt)
16 nx = int(round(1.0/dx,0)) # número de pontos em x
17 nt = int(round(1.0/dt,0)) # número de pontos em t
18 print('#_nx_=%9d' % nx)
19 print('#_nt_=%9d' % nt)
20 from numpy import zeros
21 u = zeros((2,nx+1),float) # apenas 2 posições no tempo
22                             # são necessárias!
23 def CI(x):                 # define a condição inicial
24     if 0 <= x <= 1.0:
25         return 2.0*x*(1.0-x)
26     else:
27         return 0.0
28 for i in range(nx+1):      # monta a condição inicial
29     xi = i*dx
30     u[0,i] = CI(xi)
31 u[0].tofile(fou)          # imprime a condição inicial
32 old = False
33 new = True
34 D = 2.0                   # difusividade
35 Fon = D*dt/((dx)**2)      # número de Fourier
36 print("Fo_=%10.6f" % Fon)
37 A = zeros((3,nx-1),float) # cria a matriz do sistema
38 # -----
39 # cuidado, "linha" e "coluna" abaixo não significam as reais linhas e colunas
40 # do sistema de equações, mas sim a forma de armazenar uma matriz tridiagonal
41 # -----
42 A[0,0] = 0.0              # zera A[0,0]
43 A[0,1:nx-1] = -Fon        # preenche o fim da 1a linha
44 A[1,0:nx-1] = 1.0 + 2*Fon # preenche a segunda linha
45 A[2,0:nx-2] = -Fon        # preenche o início da 2a linha
46 A[2,nx-2] = 0.0          # zera A[2,nx-2]
47 # -----
48 # importa uma tradução de tridag de Numerical Recipes para Python
49 # -----
50 from alglin import tridag
51 for n in range(nt):       # loop no tempo
52     print(n)
53     # -----
54     # atenção: calcula apenas os pontos internos de u!
55     # -----
56     u[new,1:nx] = tridag(A,u[old,1:nx])
57     u[new,0] = 0.0        # condição de contorno, x = 0
58     u[new,nx] = 0.0       # condição de contorno, x = 1
59     u[new].tofile(fou)    # imprime uma linha com os novos dados
60     (old,new) = (new,old) # troca os índices
61 fou.close()              # fecha o arquivo de saída, e fim.

```

apenas de $O(\Delta t)$. Mas é possível consertar isto! A idéia é substituir (4.39) por

$$\begin{aligned} \frac{u_i^{n+1} - u_i^n}{\Delta t} &= \frac{D}{2} \left[\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} \right], \\ u_i^{n+1} &= u_i^n + \frac{\text{Fo}}{2} [u_{i+1}^n - 2u_i^n + u_{i-1}^n + u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}]. \end{aligned} \quad (4.44)$$

Com esta mudança simples, a derivada espacial agora é uma média das derivadas em n e $n+1$, ou seja: ela está *centrada* em $n+1/2$. Com isto, a derivada temporal do lado esquerdo torna-se, na prática, um esquema de ordem 2 centrado em $n+1/2$!

Como sempre, nosso trabalho agora é verificar a estabilidade do esquema numérico. Para isto, fazemos

$$\epsilon_i^{n+1} - \frac{\text{Fo}}{2} [\epsilon_{i+1}^{n+1} - 2\epsilon_i^{n+1} + \epsilon_{i-1}^{n+1}] = \epsilon_i^n + \frac{\text{Fo}}{2} [\epsilon_{i+1}^n - 2\epsilon_i^n + \epsilon_{i-1}^n],$$

e substituímos um modo de Fourier:

$$\begin{aligned} \xi_l e^{a(t_n + \Delta t)} \left[e^{ik_l i \Delta x} - \frac{\text{Fo}}{2} (e^{ik_l(i+1)\Delta x} - 2e^{ik_l i \Delta x} + e^{ik_l(i-1)\Delta x}) \right] &= \\ \xi_l e^{at_n} \left[e^{ik_l i \Delta x} + \frac{\text{Fo}}{2} (e^{ik_l(i+1)\Delta x} - 2e^{ik_l i \Delta x} + e^{ik_l(i-1)\Delta x}) \right] &= \\ e^{a\Delta t} \left[1 - \frac{\text{Fo}}{2} (e^{ik_l \Delta x} - 2 + e^{-ik_l \Delta x}) \right] &= \left[1 + \frac{\text{Fo}}{2} (e^{ik_l \Delta x} - 2 + e^{-ik_l \Delta x}) \right] \\ e^{a\Delta t} [1 - \text{Fo}(\cos(k_l \Delta x) - 1)] &= [1 + \text{Fo}(\cos(k_l \Delta x) - 1)] \\ e^{a\Delta t} \left[1 + 2\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right) \right] &= \left[1 - 2\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right) \right] \\ e^{a\Delta t} &= \frac{1 - 2\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right)}{1 + 2\text{Fo} \sin^2 \left(\frac{k_l \Delta x}{2} \right)}. \end{aligned}$$

É fácil notar que $|e^{a\Delta t}| < 1$, e o esquema numérico de Crank-Nicholson é incondicionalmente estável. O esquema numérico de Crank-Nicholson é similar a (4.39):

$$-\frac{\text{Fo}}{2} u_{i-1}^{n+1} + (1 + \text{Fo}) u_i^{n+1} - \frac{\text{Fo}}{2} u_{i+1}^{n+1} = \frac{\text{Fo}}{2} u_{i-1}^n + (1 - \text{Fo}) u_i^n + \frac{\text{Fo}}{2} u_{i+1}^n \quad (4.45)$$

Para as condições de contorno de (4.30), as linhas correspondentes a $i = 1$ e $i = N_x - 1$ são

$$(1 + \text{Fo}) u_1^{n+1} - \frac{\text{Fo}}{2} u_2^{n+1} = (1 - 2\text{Fo}) u_1^n + \frac{\text{Fo}}{2} u_2^n, \quad (4.46)$$

$$-\frac{\text{Fo}}{2} u_{N_x-2}^{n+1} + (1 + \text{Fo}) u_{N_x-1}^{n+1} = \frac{\text{Fo}}{2} u_{N_x-2}^n + (1 - \text{Fo}) u_{N_x-1}^n \quad (4.47)$$

As mudanças no código de `difusao-imp.py` são relativamente fáceis de se identificar. O código do programa que implementa o esquema numérico de Crank-Nicholson, `difusao1d-ckn.py`, é mostrado na listagem 4.11.

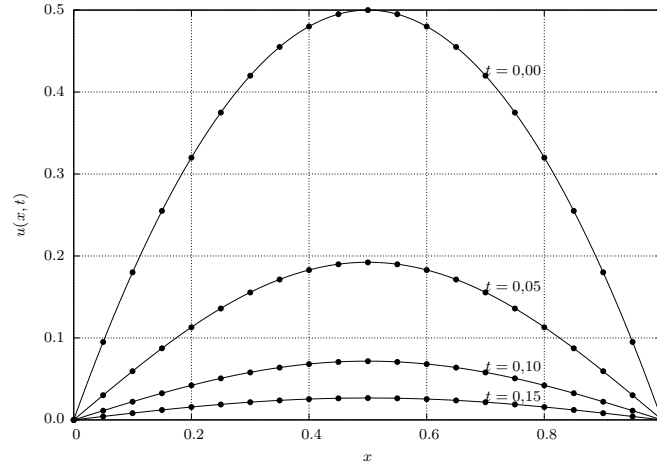


Figura 4.8: Solução numérica com o método de Crank-Nicholson ((4.45)) (círculos) *versus* a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

A grande novidade computacional de `difusao1d-ckn.py` é a linha 56: com os `arrays` proporcionados por Numpy, é possível escrever (4.45) *vetorialmente*: note que não há necessidade de fazer um *loop* em x para calcular cada elemento `b[i]` individualmente. O mesmo tipo de facilidade está disponível em FORTRAN90, FORTRAN95, etc.. Com isto, a implementação computacional dos cálculos gerada por Numpy (ou pelo compilador FORTRAN) também é potencialmente mais eficiente.

O método de Crank-Nicholson possui acurácia $O(\Delta t)^2$, portanto ele deve ser capaz de dar passos ainda mais largos no tempo que o método implícito (4.39); no programa `difusao1d-ckn.py`, nós especificamos um passo de tempo 5 vezes maior do que em `difusao1d-imp.py`.

O resultado é uma solução cerca de 5 vezes mais rápida (embora, novamente, haja mais contas agora para calcular o vetor de carga `b`), e é mostrado na figura 4.8

4.3 – Difusão em 2 Dimensões: ADI, e equações elíticas

Considere a equação da difusão em 2 dimensões,

$$\frac{\partial u}{\partial t} = D \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right). \quad (4.48)$$

Como sempre, nós queremos ser muito concretos, e trabalhar com um problema que possua solução analítica. Considere então a condição inicial

$$u(x, y, 0) = u_0 \exp \left(-\frac{(x^2 + y^2)}{L^2} \right); \quad (4.49)$$

a solução analítica é

$$u(x, y, t) = \frac{u_0}{1 + 4tD/L^2} \exp \left(-\frac{(x^2 + y^2)}{L^2 + 4Dt} \right). \quad (4.50)$$

Listagem 4.11: `difusao1d-ckn.py` — Solução numérica da equação da difusão: esquema de Crank-Nicholson.

```

1  #!/usr/bin/python
2  # -*- coding: iso-8859-1 -*-
3  # -----
4  # difusao1d-ckn resolve uma equação de difusão com o método
5  # de Crank-Nicholson
6  #
7  # uso: ./difusao1d-ckn.py
8  # -----
9  from __future__ import print_function
10 from __future__ import division
11 fou = open('difusao1d-ckn.dat','wb')
12 dx = 0.01          # define a discretização em x
13 dt = 0.005         # define a discretização em t
14 print('#_dx=_%9.4f' % dx)
15 print('#_dt=_%9.4f' % dt)
16 nx = int(round(1.0/dx,0)) # número de pontos em x
17 nt = int(round(1.0/dt,0)) # número de pontos em t
18 print('#_nx=_%9d' % nx)
19 print('#_nt=_%9d' % nt)
20 from numpy import zeros
21 u = zeros((2,nx+1),float) # apenas 2 posições no tempo
22                             # são necessárias!
23 def CI(x):                # define a condição inicial
24     if 0 <= x <= 1.0:
25         return 2.0*x*(1.0-x)
26     else:
27         return 0.0
28 for i in range(nx+1):     # monta a condição inicial
29     xi = i*dx
30     u[0,i] = CI(xi)
31 u[0].tofile(fou)         # imprime a condição inicial
32 old = False
33 new = True
34 D = 2.0                  # difusividade
35 Fon = D*dt/((dx)**2)     # número de Fourier
36 print("Fo=_%10.6f" % Fon)
37 A = zeros((3,nx-1),float) # cria a matriz do sistema
38 # -----
39 # cuidado, "linha" e "coluna" abaixo não significam as reais linhas e colunas
40 # do sistema de equações, mas sim a forma de armazenar uma matriz tridiagonal
41 # -----
42 A[0,0] = 0.0             # zera A[0,0]
43 A[0,1:nx-1] = -Fon/2.0   # preenche o fim da 1a linha
44 A[1,0:nx-1] = 1.0 + Fon  # preenche a segunda linha
45 A[2,0:nx-2] = -Fon/2.0   # preenche o início da 2a linha
46 A[2,nx-2] = 0.0         # zera A[2,nx-2]
47 # -----
48 # importa uma tradução de tridag de Numerical Recipes para Python
49 # -----
50 from alglin import tridag
51 for n in range(nt):      # loop no tempo
52     print(n)
53     # -----
54     # recalcula o vetor de carga vetorialmente
55     # -----
56     b = (Fon/2)*u[old,0:nx-1] + (1 - Fon)*u[old,1:nx] + (Fon/2)*u[old,2:nx+1]
57     # -----
58     # atenção: calcula apenas os pontos internos de u!
59     # -----
60     u[new,1:nx] = tridag(A,b)
61     u[new,0] = 0.0        # condição de contorno, x = 0
62     u[new,nx] = 0.0       # condição de contorno, x = 1
63     u[new].tofile(fou)    # imprime uma linha com os novos dados
64     (old,new) = (new,old) # troca os índices
65 fou.close()              # fecha o arquivo de saída, e fim.

```

Na verdade esta solução se “espalha” por todo o plano xy , mas nós podemos trabalhar com um problema finito em x e y , por exemplo, fazendo $-L \leq x \leq L$, $-L \leq y \leq L$, e impondo condições de contorno que se ajustem *exatamente* à solução analítica:

$$u(-L, y, t) = \frac{u_0}{1 + 4tD/L^2} \exp\left(-\frac{(L^2 + y^2)}{L^2 + 4Dt}\right), \quad (4.51)$$

$$u(L, y, t) = \frac{u_0}{1 + 4tD/L^2} \exp\left(-\frac{(L^2 + y^2)}{L^2 + 4Dt}\right), \quad (4.52)$$

$$u(x, -L, t) = \frac{u_0}{1 + 4tD/L^2} \exp\left(-\frac{(x^2 + L^2)}{L^2 + 4Dt}\right), \quad (4.53)$$

$$u(x, L, t) = \frac{u_0}{1 + 4tD/L^2} \exp\left(-\frac{(x^2 + L^2)}{L^2 + 4Dt}\right). \quad (4.54)$$

Agora, nós vamos fazer $D = 2$ (como antes) e $L = 1$, e resolver o problema numericamente. Nossa escolha recairá sobre um método simples, e de $O(\Delta t)^2$, denominado ADI (*alternating-direction implicit*). Este método nos proporcionará um exemplo de uma técnica denominada *operator splitting* ou *time splitting*, que nós vamos traduzir como “separação de operadores”. Esta técnica consiste em marchar implicitamente em uma dimensão espacial de cada vez, mantendo a outra dimensão “explícita”. Portanto, nós vamos utilizar *dois* esquemas diferentes de diferenças finitas (na prática), para resolver o problema! Ei-los

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} = D \left(\frac{u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1}}{\Delta x^2} + \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2} \right) \quad (4.55)$$

$$\frac{u_{i,j}^{n+2} - u_{i,j}^{n+1}}{\Delta t} = D \left(\frac{u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1}}{\Delta x^2} + \frac{u_{i,j+1}^{n+2} - 2u_{i,j}^{n+2} + u_{i,j-1}^{n+2}}{\Delta y^2} \right) \quad (4.56)$$

Examine cuidadosamente (4.55) e (4.56): na primeira, note que o esquema é implícito em x ; na segunda, a situação se reverte, e o esquema é implícito em y . É claro que nós vamos precisar de duas análises de estabilidade de von Neumann, uma para cada equação.

2011-09-24T17:07:04 Por enquanto, vou supor que os dois esquemas são incondicionalmente estáveis, e mandar ver.

Além disto, por simplicidade vamos fazer $\Delta x = \Delta y = \Delta$, de maneira que só haverá um número de Fourier de grade no problema,

$$\text{Fo} = \frac{D\Delta t}{\Delta^2}, \quad (4.57)$$

e então teremos, para x :

$$\begin{aligned} u_{i,j}^{n+1} - u_{i,j}^n &= \text{Fo} \left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1} + u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n \right), \\ u_{i,j}^{n+1} - \text{Fo} \left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1} \right) &= u_{i,j}^n + \text{Fo} \left(u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n \right), \\ -\text{Fo}u_{i-1,j}^{n+1} + (1 + 2\text{Fo})u_{i,j}^{n+1} - \text{Fo}u_{i+1,j}^{n+1} &= \text{Fo}u_{i,j-1}^n + (1 - 2\text{Fo})u_{i,j}^n + \text{Fo}u_{i,j+1}^n \end{aligned} \quad (4.58)$$

Na dimensão y ,

$$\begin{aligned} u_{i,j}^{n+2} - u_{i,j}^{n+1} &= \text{Fo} \left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1} + u_{i,j-1}^{n+2} - 2u_{i,j}^{n+2} + u_{i,j+1}^{n+2} \right), \\ u_{i,j}^{n+2} - \text{Fo} \left(u_{i,j-1}^{n+2} - 2u_{i,j}^{n+2} + u_{i,j+1}^{n+2} \right) &= u_{i,j}^{n+1} + \text{Fo} \left(u_{i-1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i+1,j}^{n+1} \right), \\ -\text{Fou}_{i,j-1}^{n+2} + (1 + 2\text{Fo})u_{i,j}^{n+2} - \text{Fou}_{i,j+1}^{n+2} &= \text{Fou}_{i-1,j}^{n+1} + (1 - 2\text{Fo})u_{i,j}^{n+1} + \text{Fou}_{i+1,j}^{n+1} \end{aligned} \quad (4.59)$$

Se nós utilizarmos (novamente por simplicidade) o mesmo número de pontos $N + 1$ em x e em y , teremos o seguinte mapeamento para a nossa grade:

$$N = \frac{2L}{\Delta}; \quad (4.60)$$

$$x_i = -L + i\Delta, \quad i = 0, \dots, N, \quad (4.61)$$

$$y_j = -L + j\Delta, \quad j = 0, \dots, N, \quad (4.62)$$

e portanto $-L \leq x_i \leq L$ e $-L \leq y_j \leq L$. Lembrando que os valores de $u_{0,j}$, $u_{N,j}$, $u_{i,0}$ e $u_{i,N}$ estão especificados, há $(N - 1)^2$ incógnitas para serem calculadas. A beleza de (4.58) e (4.59) é que em vez de resolver a cada passo (digamos) $2\Delta t$ um sistema de $(N - 1)^2$ incógnitas, nós agora podemos resolver a cada passo Δt $N - 1$ sistemas de $(N - 1)$ incógnitas, alternadamente para $u_{1,\dots,N-1;j}$ e $u_{i;1,\dots,N-1}$.

É claro que o céu é o limite: poderíamos, por exemplo, em vez de usar um esquema totalmente implícito, usar Crank-Nicholson em cada avanço Δt ; isto nos daria imediatamente um esquema com acurácia de ordem Δt^2 . No entanto, assim como está o método ADI já é suficientemente sofisticado para nosso primeiro encontro com este tipo de problema. Devemos, portanto, programá-lo. Vamos, inicialmente, programar a solução analítica.

A solução analítica do problema para os instantes de tempo $t = 0$, $t = 0,1$, $t = 0,2$ e $t = 0,3$ está mostrada na figura 4.10

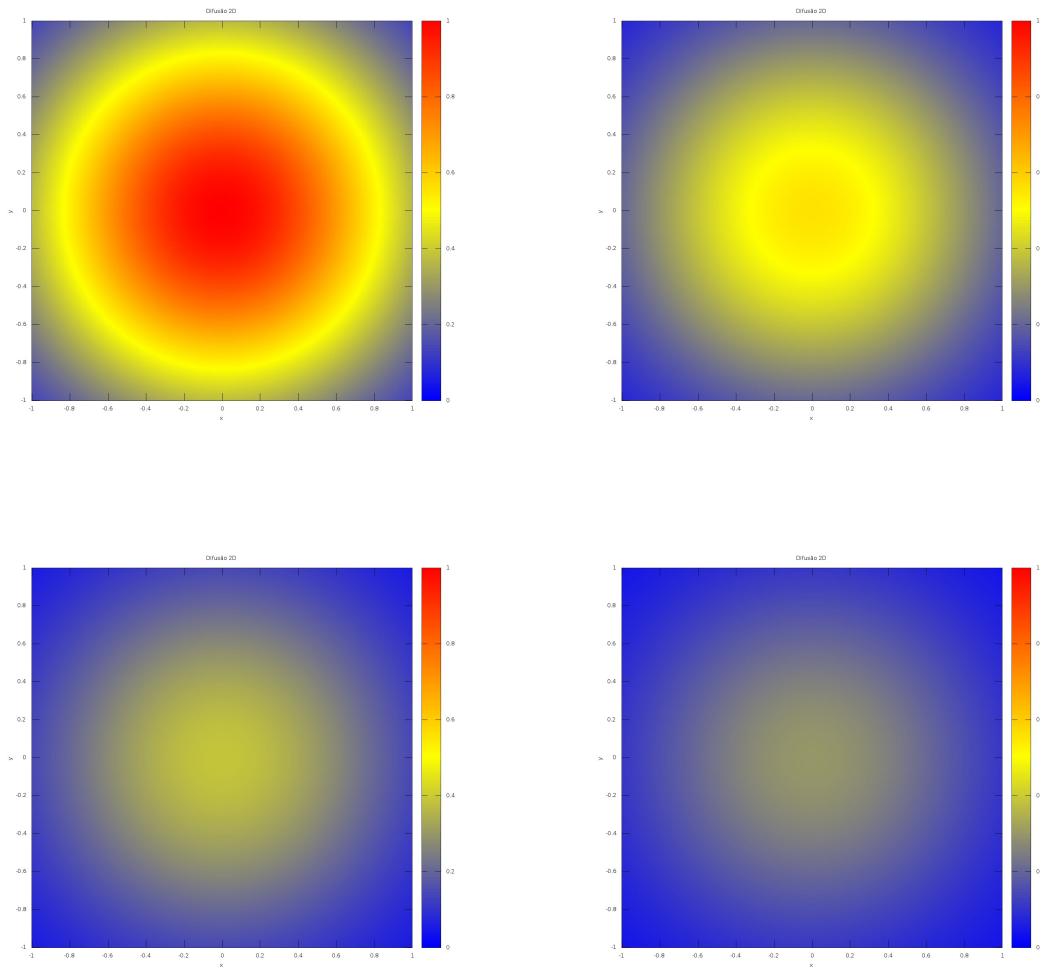


Figura 4.9: Solução analítica da equação da difusão bidimensional, para $t = 0$, $t = 0,1$, $t = 0,2$ e $t = 0,3$

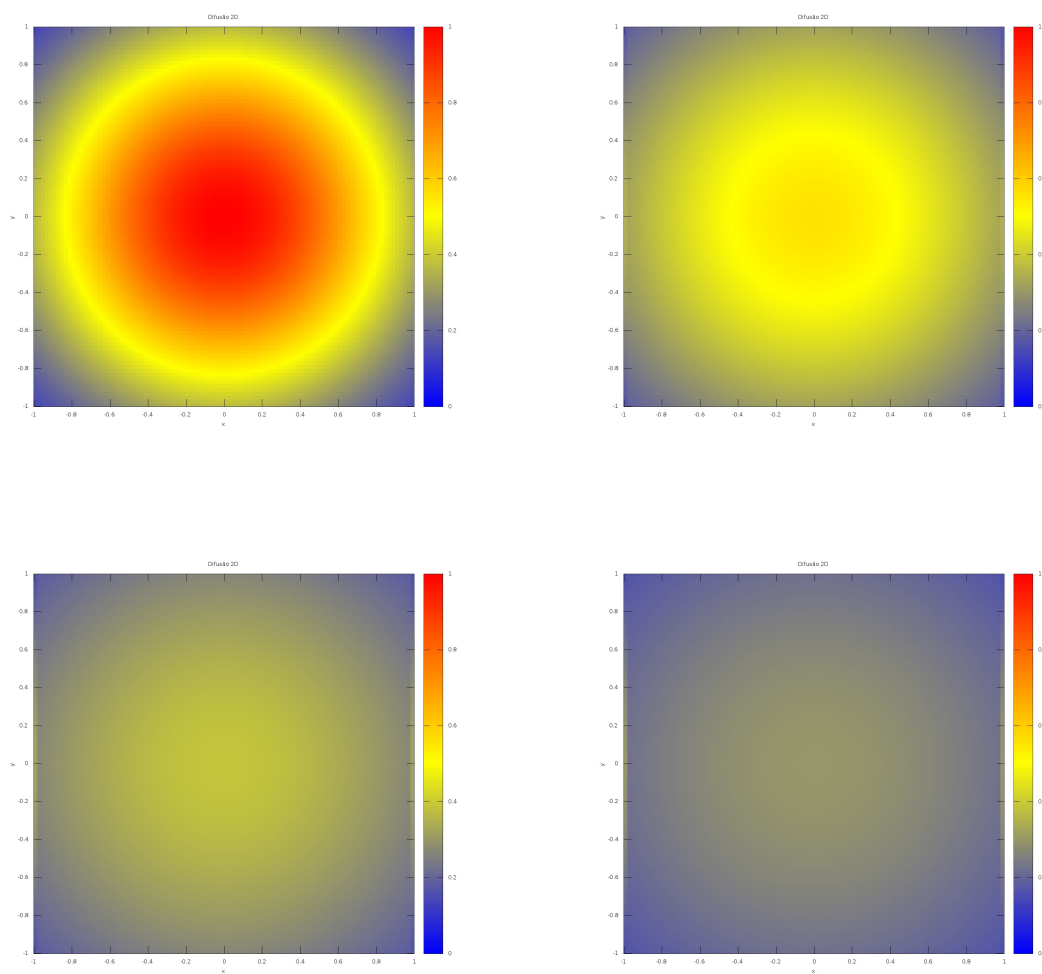


Figura 4.10: Solução numérica da equação da difusão bidimensional com o esquema ADI, para $t = 0$, $t = 0.1$, $t = 0.2$ e $t = 0.3$

A

Dados de vazão média anual e vazão máxima anual, Rio dos Patos, 1931–1999

1931	21.57	272.00
1932	25.65	278.00
1933	4.76	61.60
1934	11.46	178.30
1935	28.10	272.00
1936	14.30	133.40
1937	22.09	380.00
1938	24.09	272.00
1939	22.29	251.00
1940	7.48	56.10
1941	27.49	171.60
1942	19.11	169.40
1943	15.62	135.00
1944	16.25	146.40
1945	16.57	299.00
1946	26.75	206.20
1947	21.88	243.00
1948	20.68	223.00
1949	8.36	68.40
1950	21.62	165.00
1951	24.72	266.00
1952	14.59	192.10
1953	15.31	131.80
1954	27.33	281.00
1955	28.23	311.50
1956	15.64	156.20
1957	41.70	399.50
1958	20.04	152.10
1959	14.50	127.00
1960	22.61	176.00
1961	30.82	257.00
1962	15.22	133.40
1963	22.49	248.00
1964	24.23	211.00
1965	36.80	208.60
1966	21.60	152.00
1967	13.25	92.75
1968	9.05	125.00
1969	22.94	135.60
1970	25.95	202.00
1971	32.82	188.00
1972	34.13	198.00
1973	30.33	252.50
1974	17.81	119.00
1975	26.77	172.00
1976	32.50	174.00
1977	13.63	75.40
1978	13.26	146.80
1979	26.97	222.00

1980	26.92	182.00
1981	14.73	134.00
1982	31.68	275.00
1983	57.60	528.00
1984	27.13	190.00
1985	12.55	245.00
1986	16.74	146.80
1987	26.64	333.00
1988	15.22	255.00
1989	31.20	226.00
1990	43.48	275.00
1991	11.92	131.00
1992	35.24	660.00
1993	34.30	333.00
1994	20.74	128.00
1995	31.78	472.00
1996	35.44	196.00
1997	41.02	247.50
1998	51.55	451.00
1999	24.49	486.00

Referências Bibliográficas

- Abramowitz, M. e Stegun, I. A., editores (1972). *Handbook of mathematical functions*. Dover Publications, Inc., New York.
- Bender, C. M. e Orszag, S. A. (1978). *Advanced Mathematical Methods for Scientists and Engineers*. McGraw-Hill.
- El-Kadi, A. I. e Ling, G. (1993). The Courant and Peclet Number Criteria for the Numerical Solution of the Richards Equation. *Water Resour Res*, 29:3485–3494.
- Oliphant, T. E. (2006). *Guide to Numpy*. Trelgol Publishing.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., e Flannery, B. P. (1992). *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK. 1020 pp.