

**EAMB7024 MÉTODOS NUMÉRICOS EM ENGENHARIA AMBIENTAL  
— NOTAS DE AULA**

Nelson Luís da Costa Dias

30 de agosto de 2024

Versão: 2024-08-30T17:57:38

©Nelson Luís da Costa Dias, 2024. Todos os direitos deste documento estão reservados. Este documento não está em domínio público. Cópias para uso acadêmico podem ser feitas e usadas livremente, e podem ser obtidas em <http://nldias.github.io>. Este documento é distribuído sem nenhuma garantia, de qualquer espécie, contra eventuais erros aqui contidos.

# Sumário

---

<b>1 Ferramentas computacionais</b>	<b>6</b>
1.1 Antes de começar a trabalhar, . . . . .	6
1.2 Chapel . . . . .	7
<b>2 Métodos numéricos para o Cálculo de uma variável</b>	<b>15</b>
2.1 Aproximações de diferenças finitas e suas ordens . . . . .	15
2.2 Solução numérica de equações diferenciais ordinárias . . . . .	16
2.3 Solução numérica; o método de Euler de ordem1 . . . . .	17
2.4 Um esquema de diferenças centradas, com tratamento analítico . . . . .	21
2.5 A forma padrão $dy/dx = f(x, y)$ . . . . .	24
2.6 Interlúdio sobre arrays . . . . .	30
2.7 Uma medida relativa de tempo de processamento . . . . .	32
2.8 O método de Runge-Kutta multidimensional . . . . .	34
2.9 Problemas de valor de contorno em 1D . . . . .	43
2.10 Trabalhos computacionais . . . . .	53
<b>3 Solução numérica de equações diferenciais parciais</b>	<b>68</b>
3.1 Advecção pura: a onda cinemática . . . . .	68
3.2 Difusão pura . . . . .	89
3.3 Difusão em 2 Dimensões: ADI, e equações elíticas . . . . .	108
3.4 Trabalhos Computacionais . . . . .	118
3.5 Successive over-relaxation . . . . .	131
3.6 Really efficient Successive Over-Relaxation . . . . .	135

# **Lista de Tabelas**

---

1.1	Vazões Máximas Anuais ( $\text{m}^3 \text{s}^{-1}$ ) no Rio dos Patos, PR, 1931–1999 . . . . .	9
2.1	Curva de remanso em canal trapezoidal – solução de Ven Te Chow . . . . .	61
2.2	Parâmetros do modelo de ? . . . . .	65
3.1	First values of $m$ , $n$ , their sum, and $(2m + 1)$ and $(2n + 1)$ in (3.77)–(3.78). . . . .	109
3.2	Grid size $N_n$ , number of iterations to convergence $n_c$ , estimated $\bar{u}$ , MAD and relative runtime $t_r$ for the serial and parallel versions of the solution of Laplace's equation with SOR. . . . .	135
3.3	Ratio of serial to parallel runtimes. . . . .	135
3.4	Grid size $N_n$ , number of iterations to convergence $n_c$ , estimated $\bar{u}$ , MAD and relative runtime $t_r$ for the serial (laplace-sor) and accelerated serial (laplace-asor) versions of the solution of Laplace's equation with SOR. . . . .	139
3.5	Grid size $N_n$ , number of iterations to convergence $n_c$ , estimated $\bar{u}$ , MAD and relative runtime $t_r$ for the serial (laplace-sor) and accelerated serial (laplace-asor) versions of the solution of Laplace's equation with SOR. . . . .	141

# **Lista de Figuras**

---

1.1	FDA empírica da vazão máxima anual no Rio dos Patos. . . . .	12
2.1	Solução da equação (2.7) para $y(0) = 0$ . . . . .	17
2.2	Comparação da solução analítica da equação (2.7) com a saída de <code>sucesso.chpl</code> , para $\Delta x = 0,01$ . . . . .	20
2.3	Comparação da solução analítica da equação (2.7) com a saída de <code>sucesso.chpl</code> , para $\Delta x = 0,5$ . . . . .	21
2.4	Comparação da solução analítica da equação (2.7) com a saída de <code>succent.chpl</code> , para $\Delta x = 0,5$ . . . . .	23
2.5	Os métodos de Euler de ordens 1 e 2. . . . .	25
2.6	Comparação da solução analítica da equação (2.7) com a saída de <code>euler2.chpl</code> , para $\Delta x = 0,5$ . . . . .	29
2.7	Comparação da solução analítica da equação (2.7) com a saída de <code>rungek4.chpl</code> , para $\Delta x = 0,5$ . . . . .	29
2.8	Solução numérica pelo Método de Runge-Kutta de um sistema de 2 equações diferenciais ordinárias . . . . .	37
2.9	Solução numérica de (2.17)–(2.19). . . . .	44
2.10	Erro da solução numérica de (2.17)–(2.19) em função do número de pontos $N$ da grade. . . . .	47
2.11	Solução numérica de (2.26)–(2.28) com uma discretização de ordem 1 para a condição de contorno direita. . . . .	49
2.12	Erro da solução numérica de (2.26)–(2.28) em função do número de pontos $N$ da grade. O coeficiente angular da reta é $\sim -1$ . . . . .	50
2.13	Solução numérica de (2.26)–(2.28) com uma discretização de ordem 2 para a condição de contorno direita. . . . .	50
2.14	Erro da solução numérica de (2.26)–(2.28) em função do número de pontos $N$ da grade. O coeficiente angular da reta é $\sim -2$ . . . . .	52
2.15	Um pêndulo (possivelmente) não-linear. . . . .	53
2.16	O período de um pêndulo não linear em função da amplitude inicial $\Theta_0$ . A linha tracejada é o valor teórico $f(0) = 2\pi$ para oscilações de pequena amplitude. . . . .	56
2.17	Características geométricas de um canal. . . . .	60
2.18	Área molhada em função da distância de jusante do exemplo 10.1 de ?, calculados por Chow (círculos) versus resultados do método de Runge-Kutta (linha contínua). . . . .	62
2.19	Perda de carga em função da distância de jusante do exemplo 10.1 de ?, calculados por Chow (círculos) versus resultados do método de Runge-Kutta (linha contínua). . . . .	63
2.20	Velocidade em função da distância de jusante do exemplo 10.1 de ?, calculados por Chow (círculos) versus resultados do método de Runge-Kutta (linha contínua). . . . .	63
2.21	Cota em função da distância de jusante do exemplo 10.1 de ?, calculados por Chow (círculos) versus resultados do método de Runge-Kutta (linha contínua). . . . .	64
2.22	A traça ( <i>Spruce budworm</i> ) <i>Choristoneura orae</i> . Fonte: Wikipedia. ©entomart ( <a href="http://www.entomart.be">http://www.entomart.be</a> ). . . . .	65

2.23 Simulação de eclosão de uma praga de traças, utilizando o modelo de ? . . . . .	66
2.24 O oscilador não-linear de van der Pol ( $\mu = 5$ ). . . . .	67
3.1 Solução analítica das equações 3.3–3.4. . . . .	69
3.2 Solução numérica produzida por <code>onda1d-ins.chpl</code> , para $t = 250\Delta t$ , $500\Delta t$ e $750\Delta t$ . . . . .	72
3.3 Solução numérica produzida por <code>onda1d_lax.chpl</code> , para $t = 500\Delta t$ , $1000\Delta t$ e $1500\Delta t$ . . . . .	77
3.4 Solução numérica produzida pelo esquema <i>upwind</i> , para $t = 250$ , $500$ e $750$ . . . . .	78
3.5 Quick's interpolation scheme. . . . .	79
3.6 Solução numérica produzida pelo esquema QUICK, para $t = 250$ , $500$ e $750$ . . . . .	83
3.7 As funções $\sin^2(\theta)$ e $\cos^2(\theta)$ . . . . .	87
3.8 A função definida em 3.44 (para $Fo = 1$ ) . . . . .	87
3.9 Exemplo 3.1: interseção (em cinza escuro) de 32 regiões (em cinza-claro; incrementos de $\pi/32$ ) na equação (3.44) A parábola em preto representa o caso degenerado $\theta = 0$ . . . . .	88
3.10 Solução analítica da equação de difusão para $t = 0$ , $t = 0,05$ , $t = 0,10$ e $t = 0,15$ . . . . .	92
3.11 Solução numérica com o método explícito (3.54) (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. . . . .	96
3.12 Solução numérica com o método implícito (3.58) (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. . . . .	99
3.13 Solução numérica com o método de Crank-Nicholson ((3.64)) (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. . . . .	101
3.14 Comparação entre as soluções analítica (linhas) e numérica com um esquema implícito (pontos) da equação da difusão-advecção com termo de decaimento, para $t = 0,333$ , $t = 0,666$ e $t = 0,999$ . . . . .	106
3.15 Comparison between the analytical (mesh) and numerical (points; ADI) solutions of the two-dimensional diffusion equation. . . . .	117
3.16 Solução correta do problema para $t = 0, 0,5, 1,0, 1,5, 2,0$ . As linhas cheias são a solução analítica, e os pontos a solução numérica. . . . .	121
3.17 Parte real de $\phi(\zeta, \tau)$ , $\tau = 5, 25, 50, 100$ , e solução analítica. . . . .	123
3.18 Parte imaginária de $\phi(\zeta, \tau)$ , $\tau = 5, 25, 50, 100$ , e solução analítica. . . . .	124
3.19 Comparação do esquema numérico com a solução analítica de ?. De cima para baixo, $\tau = 0,01$ , $\tau = 0,1$ , e $\tau = 1,0$ . . . . .	127
3.20 Comparação da solução numérica completa de (3.134)–(3.135) com a solução linearizada (3.143) para $\Phi_0 = 0$ . De cima para baixo, $\tau = 0,01$ , $\tau = 0,1$ , e $\tau = 1,0$ . . . . .	128
3.21 Comparação da solução numérica completa de (3.134)–(3.135) com a solução linearizada (3.143) para $\Phi_0 = 0,25$ . De cima para baixo, $\tau = 0,01$ , $\tau = 0,1$ , e $\tau = 1,0$ . . . . .	129
3.22 Comparação da solução numérica completa de (3.134)–(3.135) com a solução linearizada (3.143) para $\Phi_0 = 0,50$ . De cima para baixo, $\tau = 0,01$ , $\tau = 0,1$ , e $\tau = 1,0$ . . . . .	129
3.23 Comparação da solução numérica completa de (3.134)–(3.135) com a solução linearizada (3.143) para $\Phi_0 = 0,75$ . . . . .	130
3.24 Running times $t_r$ as a function of $n = N_n^2$ for the SOR method, serial and parallel implementations. The lines are parabolas of the type $t_r = an^2$ . . . . .	137
3.25 Alternating grids in 2D for the SOR methods. Note that when we update the black points using (3.171), only the white points are used, and vice-versa. In this way, we can update the whole grid in 2 half-sweeps, with the blacks depending only the whites (and vice-versa), so that race conditions never arise. . . . .	137

# **Lista de Listagens**

---

1.1	<code>binint.chpl</code> – Exemplo de <i>strings</i> , e inteiros.	7
1.2	<code>floats.chpl</code> – Exemplo de uso de <code>real</code> e <code>complex</code> .	9
1.3	<code>patos-medmax.dat</code> – Vazões média e máxima anuais, Rio dos Patos	10
1.4	<code>fqiemp.chpl</code> – Cálculo de uma FDA empírica	11
1.5	<code>fqiemp.dat</code> – FDA empírica da vazão máxima anual no Rio dos Patos.	11
1.6	<code>bintext.chpl</code> – Exemplo de arquivo texto e arquivo binário	13
1.7	<code>writearr.chpl</code> – Escreve um arquivo binário contendo 3 “linhas”, cada uma das quais com um <i>array</i> de 10 <i>reals</i> .	13
1.8	<code>readarr.chpl</code> – Lê um arquivo binário contendo 3 “linhas”, cada uma das quais com um <i>array</i> de 10 <i>floats</i> .	14
2.1	<code>fracasso.chpl</code> – Um programa com o método de Euler que não funciona	18
2.2	<code>fracasso.out</code> – Saída do programa <code>fracasso</code> .	18
2.3	<code>sucesso.chpl</code> – Um programa com o método de Euler que funciona	20
2.4	<code>succent.chpl</code> – Método de Euler implícito	22
2.5	<code>euler2</code> – Um método explícito de ordem 2	26
2.6	<code>rungek4</code> – Método de Runge-Kutta, ordem 4	27
2.7	<code>learnarray</code> – Alguns aspectos de arrays em Chapel	30
2.8	<code>fada</code> – Implementa um vetor ( <code>vec</code> ) que contorna a limitação de arrays genéricos em argumentos de rotinas.	32
2.9	<code>unitTime</code> – Uma medida aproximada de tempo de processamento independente da máquina.	33
2.10	<code>runkutr</code> – Implementa o método de Runge-Kutta multidimensional.	34
2.11	<code>rktestr</code> – Resolve um sistema de EDOs com o método de Runge-Kutta multidimensional.	35
2.12	<code>oncin.chpl</code> – Propagação de cheia com o método de Runge-Kutta.	40
2.13	<code>tridiag.chpl</code> – Exporta uma rotina que resolve um sistema tridiagonal.	45
2.14	<code>edo-l1.chpl</code> – Solução do problema de valor de contorno (2.17)–(2.19).	46
2.15	<code>edo-l2.chpl</code> – Solução do problema de valor de contorno (2.26)–(2.28).	48
2.16	<code>edo-l2b.chpl</code> – Solução do problema de valor de contorno (2.26)–(2.28) com um esquema de ordem 2 para a condição de contorno direita.	51
3.1	<code>onda1d_ins.chpl</code> – Solução de uma onda cinemática 1D com um método explícito instável	70
3.2	<code>surf1d_ins.chpl</code> – Seleciona alguns intervalos de tempo da solução numérica para plotagem	71
3.3	<code>onda1d_lax.chpl</code> – Solução de uma onda cinemática 1D com o método de Lax	75
3.4	<code>surf1d_lax.chpl</code> – Seleciona alguns intervalos de tempo da solução numérica para plotagem	76
3.5	<code>quick_grid.chpl</code> – A stable grid for QUICK’s solution of (3.1)	80
3.6	<code>onda1d_quick.chpl</code> – Solution of a 1D kinematic wave using QUICK	81
3.7	<code>surf1d_quick.chpl</code> – Seleciona alguns intervalos de tempo da solução numérica com QUICK para plotagem	82
3.8	<code>difusao1d-ana.chpl</code> – Solução analítica da equação da difusão	90

3.9	<code>divisao1d-ana.py</code> – Seleciona alguns instantes de tempo da solução analítica para visualização . . . . .	91
3.10	<code>difusao1d-exp.py</code> – Solução numérica da equação da difusão: método explícito. . . . .	94
3.11	<code>divisao1d-exp.chpl</code> – Seleciona alguns instantes de tempo da solução analítica para visualização . . . . .	95
3.12	<code>difusao1d-imp.chpl</code> – Solução numérica da equação da difusão: método implícito. . . . .	98
3.13	<code>difusao1d-ckn.chpl</code> – Solução numérica da equação da difusão: esquema de Crank-Nicholson. . . . .	100
	<code>sana.max</code> . . . . .	104
3.14	Implementação de um esquema numérico implícito para a equação da difusão-advecção.	105
3.15	<code>difgrid2d.chpl</code> – Grid constants for solutions of the two-dimensional diffusion equation. . . . .	109
3.16	<code>difana2d.chpl</code> – Analytical solution of the two-dimensional diffusion equation. . . . .	110
3.17	<code>difadi2d.chpl</code> – Numerical (ADI) solution of the two-dimensional diffusion equation.	112
3.18	<code>difadi2d-p</code> – Parallel implementation of the ADI method: sweep in the $x$ direction. . . . .	114
3.19	<code>difview2d.chpl</code> – Read a chosen timestep from a binary file and print it in a text file. . . . .	116
3.20	<code>laplace-sor.chpl</code> – Solution of Laplace's equation with successive over relaxation. . . . .	132
3.21	<code>laplace-sor-p</code> – Parallel solution of Laplace's equation with successive over relaxation.	133
3.22	<code>laplace-asor</code> – Solution of Laplace's equation in 2D over staggered grids using Chebyshev acceleration. . . . .	136
3.23	<code>laplace-asor-p</code> – Parallel solution of Laplace's equation with accelerated successive over relaxation. . . . .	139

# 1

## Ferramentas computacionais

---

Neste curso todos os exemplos serão feitos em Chapel. Chapel é uma *linguagem de programação paralela*, bem recente. Visite a página da linguagem em <https://chapel-lang.org/>. Ela pode ser instalada com facilidade em Linux. Em Windows provavelmente a melhor rota é instalar dentro de WSL: veja <https://learn.microsoft.com/en-us/windows/wsl/install>.

No entanto, você pode fazer os trabalhos do curso na linguagem que for mais conveniente para você, desde que eu consiga rodá-la no meu computador. As linguagens que você pode escolher são

Linguagem	Sistemas Operacionais	Onde encontrar
Chapel	Linux, MacOs, Windows(?)	<a href="https://chapel-lang.org/">https://chapel-lang.org/</a>
Fortran	Linux, MacOs, Windows	<a href="https://gcc.gnu.org/wiki/GFortran">https://gcc.gnu.org/wiki/GFortran</a>
C	Linux, MacOs, Windows	(Variável: parte de <a href="https://gcc.gnu.org/">https://gcc.gnu.org/</a> )
Pascal	Linux, MacOs, Windows	<a href="https://www.freepascal.org/">https://www.freepascal.org/</a>
MatLab <b>(substitua por Octave)</b>	Linux, MacOs, Windows	<a href="https://octave.org/">https://octave.org/</a>
Basic	Linux, MacOs, Windows	<a href="https://freebasic.net/">https://freebasic.net/</a>

### 1.1 – Antes de começar a trabalhar,

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.
- 4) Certificar-se de que você tem Chapel instalado.

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 *notepad++* (<https://notepad-plus-plus.org>). Em Linux, editores de texto simples são o *gedit* (<http://projects.gnome.org/gedit/>) — que também funciona muito bem em Windows, e o *kate*. Programadores mais experientes costumam preferir o *vim*,

ou o *Emacs*. Esses dois últimos possuem versões para os 3 sistemas operacionais mais comuns hoje em dia: Windows, Linux e Mac OS X.

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

- 1) Criar o arquivo com o programa em Chapel, usando o editor de texto, e salvá-lo.
- 2) Ir para a linha de comando.
- 3) Executar o programa digitando o seu nome (*e não clicando!*).
- 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 essas 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!

## 1.2 – Chapel

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

### 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 *string* em Chapel. O tipo dos números inteiros chama-se *int* em Chapel.

A listagem 1.1 mostra o conteúdo do arquivo *binint.chpl* com alguns exemplos simples do uso de inteiros e *strings*. A legenda de cada listagem se inicia sempre com o nome do arquivo correspondente. A listagem é um retrato fiel do arquivo, com duas exceções: as *palavras reservadas* de Chapel estão sublinhadas na listagem (mas não no arquivo), e os espaços em branco *dentro das strings* estão enfatizados pelo símbolo `.`

Listagem 1.1: *binint.chpl* – Exemplo de *strings*, e inteiros.

---

```

1 var a = "açafrão";           // a é uma string
2 writeln(a);                 // imprime a na tela
3 writeln(a.size);            // imprime o número de caracteres de a
4 var i = 2**32 - 1;          // i é o maior int s/sinal que cabe em 32 bits
5 var b = i:string;
6 writeln(b);                 // imprime b na tela
7 writeln("--- a:");
8 for c in a do {             // p/ cada c de a, imprime seu valor unicode
9   writeln("ord(", c, ") = ", ord(c));
10 }
11 writeln("--- b:");
12 for c in b do {             // p/ cada c de b, imprime seu valor unicode
13   writeln("ord(", c, ") = ", ord(c));
14 }
15 writeln(chr(227));          // imprime o caractere unicode no 227
16 // -----
17 // --> ord: the codepoint of a 1-char string
18 // -----
19 proc ord(c: string): int(32) {
20   assert(c.size == 1);
21   return c.toCodepoint();
22 }
```

---

---

```

23 // -----
24 // --> chr: the 1-char string corresponding to a codepoint
25 // -----
26 proc chr(i: int(32)): string {
27     return codepointToString(i);
28 }
```

---

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.chpl` está explicada nos próprios comentários.

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

```
açafrão
7
4294967295
--- a:
ord( a ) = 97
ord( ç ) = 231
ord( a ) = 97
ord( f ) = 102
ord( r ) = 114
ord( á ) = 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
á
```

---

`writeln` 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*<sup>1</sup> 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 á.

Se você estiver vendo uma saída diferente da mostrada acima, com caracteres estranhos, não se assuste (demais): o seu arquivo `binint.chpl` e o terminal dentro do qual você está executando este programa provavelmente estão utilizando codificações diferentes.

## 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  $\mathbb{Q}$ .

Chapel 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 (`real`) e “complexos” (`complex`) não podia ser mais simples, na listagem 1.2.

---

<sup>1</sup>veja <https://pt.wikipedia.org/wiki/Unicode>

Listagem 1.2: floats.chpl – Exemplo de uso de real e complex.

---

```

1 use Math only pi, e, cos, sin, sqrt;           // pi, e, seno, raiz q
2 writeln("pi = ",pi);                         // imprime o valor de pi
3 writeln("e = ",e);                           // imprime o valor de e
4 writeln("sen(pi/2) = ",sin(pi/2));          // imprime sen(pi/2)
5 writeln("sen(sqrt(-1)) = ",sin(0.0+1.0i)); // imprime sen(sqrt(-1))

```

---

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

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		

---

Eis a saída de floats.chpl:

---

```

pi = 3.14159
e = 2.71828
sen(pi/2) = 1.0
sen(sqrt(-1)) = 0.0 + 1.1752i

```

---

## 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

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

---

tabela 1.1, se  $q_i$  é a vazão máxima do  $i$ -ésimo ano, teríamos que a probabilidade de ocorrência de  $q_i$  é

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

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

$$q_1 \leq q_2 \leq \dots \leq q_n.$$

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

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

$F(q_i)$  em (1.3) é chamada de distribuição acumulada *empírica* de probabilidade. Em Hidrologia, muitas vezes (1.3) é denominada *curva de permanência*. O resultado  $i/n$  é denominado uma *posição de plotagem*. Por diversos motivos, existem muitas outras posições de plotagem possíveis para a FDA empírica. Uma muito popular é  $i/(n + 1)$ . 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 ??). Esse 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.chpl, mostrado na listagem 1.4, calcula a curva de permanência, ou FDA empírica, para as vazões máximas anuais do Rio dos Patos. Essa é, simplesmente, uma tabela de duas colunas: a vazão observada (em ordem crescente), e o valor de  $i/n$ .

As primeiras 5 linhas do arquivo de saída fqiemp.dat são mostradas na listagem 1.5; o seu gráfico, plotado a partir de fqiemp.dat, é mostrado na figura 1.1.

Listagem 1.4: fqiemp.chpl – Cálculo de uma FDA empírica

---

```

1 use IO only openReader, openWriter;
2 use Sort;
3 use dgrow;
4 // -----
5 // abre o arquivo de entrada
6 // -----
7 const fin = openReader("patos-medmax.dat", locking=false);
8 var dom = {1..10};
9 var qmax: [dom] real;           // um array com 10 elementos
10 var linha: string;
11 var n = 0;
12 while fin.readLine(linha) do {   // loop nas linhas do arquivo
13     var campo = linha.split();    // separa os campos
14     writeln(campo);             // para ver campo a campo na tela
15     n += 1;                     // incrementa n
16     dgrow(n, dom);              // ajusta o domínio
17     qmax[n] = campo[2]: real;   // a vazão é o terceiro campo
18 }
19 fin.close();                   // fecha o arquivo de entrada
20 dom = {1..n};                  // reajusta o domínio para o tamanho final
21 // -----
22 // abre o arquivo de saída
23 // -----
24 const fou = openWriter("fqiemp.dat", locking=false);
25 sort(qmax);                   // ordena as vazões
26 for i in dom do {             // loop nas vazões
27     var qi = qmax[i];          // vazão
28     var Fi = (i:real)/n;        // posição de plotagem
29     // imprime uma linha
30     fou.writef("%8.2dr %8.6dr\n",qi,Fi);
31 }
32 fou.close();                  // fim de papo

```

---

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

---

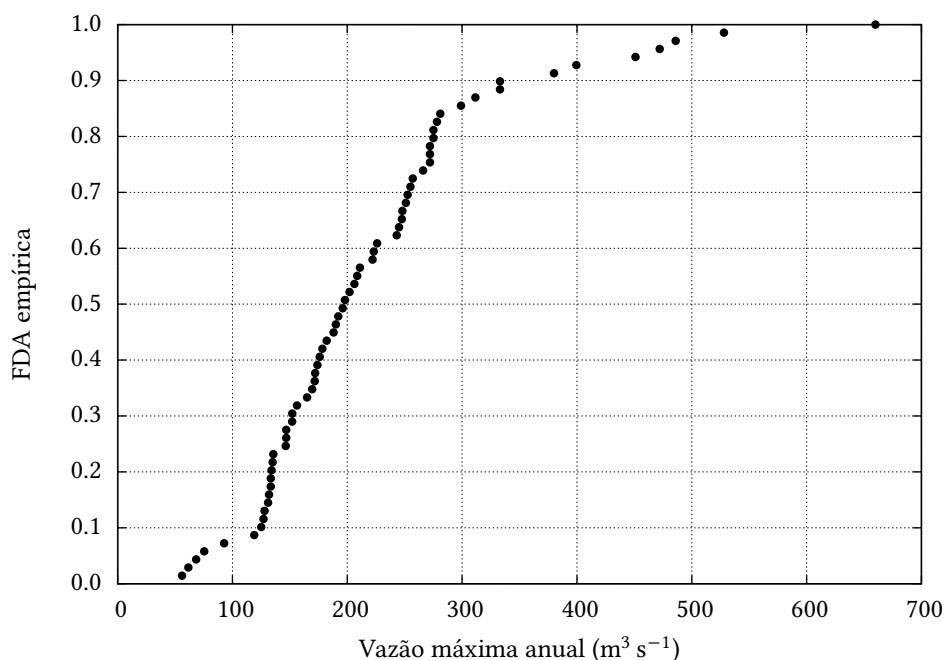


Figura 1.1: FDA empírica da vazão máxima anual no Rio dos Patos.

## 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..

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

Listagem 1.6: `bintext.chpl` – Exemplo de arquivo texto e arquivo binário

---

```

1 use IO only openWriter,binarySerializer;
2 var i: int(32) = 673451;
3 const fot = openWriter("i.txt",locking=false);
4 const ffot = fot.getFile();
5 fot.writef("%6i",i);
6 fot.close();
7 writef("fot size = %i\n", ffot.size);
8 const fob = openWriter("i.bin",serializer=new binarySerializer(),locking=false);
9 const ffob = fob.getFile();
10 fob.write(i);
11 fob.close();
12 writef("fob size = %i\n", ffob.size);

```

---

Eis aqui a saída de `bintext`:

---

```

fot size = 6
fob size = 4

```

---

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 escrita de arquivos binários é muito mais *rápida*, porque não há necessidade de traduzir a informação de, e para, *strings*. Na prática, arquivos binários estão invariavelmente associados ao uso de *arrays*, pelo menos em Engenharia.

Primeiro, um programa para *escrever* um array. A listagem 1.7 mostra o programa `writearr.chpl`.

Listagem 1.7: `writearr.chpl` – Escreve um arquivo binário contendo 3 “linhas”, cada uma das quais com um *array* de 10 reals.

---

```

1 use IO only openWriter,binarySerializer;
2 use Random only fillRandom;
3 // -----
4 // abre um arquivo binário para escrita
5 // -----
6 const fob = openWriter("a.bin",serializer=new binarySerializer(),locking=false);
7 var a: [1..10] real;
8 for k in 1..3 do           // loop em 3 "linhas"
9   fillRandom(a);          // gera um array com 10 números aleatórios
10  fob.write(a);            // escreve uma "linha"
11 }
12 fob.close();

```

---

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.chpl`, ele escreve 10 *reals* no arquivo. Cada *real* 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. Isso é feito com o programa `readarr.py`, mostrado na listagem 1.8.

Listagem 1.8: `readarr.chpl` – Lê um arquivo binário contendo 3 “linhas”, cada uma das quais com um *array* de 10 *floats*.

---

```

1 use IO only openReader,binaryDeserializer;
2 use Random only fillRandom;
3 // -----
4 // abre o arquivo binário
5 // -----
6 const fib = openReader("a.bin",deserializer=new binaryDeserializer(),
7   locking=false);
8 var a: [1..10] real;
9 for k in 1..3 do {           // loop em 3 "linhas"
10   fib.read(a);             // lê um array com 10 floats
11   for e in a do {          // imprime com formato na tela
12     writef("%5.4dr",e);
13   }
14   writef("\n");
15 }
16 // -----
17 // fecha o arquivo binário
18 // -----
19 fib.close();

```

---

Eis a sua saída:

---

```

0.3375 0.3076 0.8824 0.9950 0.9612 0.5473 0.9010 0.1064 0.0738 0.6007
0.2119 0.4100 0.7633 0.2188 0.8546 0.2163 0.7849 0.6421 0.9421 0.0840
0.3684 0.5918 0.2107 0.1978 0.1820 0.4022 0.9412 0.1794 0.5461 0.1043

```

---

O que vemos são os números aleatórios das 3 instâncias de `a` escritas pelo programa `writearr.chpl`. Atenção: cada vez que você rodar `writearr.chpl`, a sua saída será diferente (pois os números gerados são aleatórios<sup>2</sup>). Portanto, não se preocupe em reproduzir *exatamente* os valores acima, pois eles sempre estarão mudando a cada rodada de `writearr.chpl`.

---

<sup>2</sup>Na verdade, *pseudo-aleatórios*; mas isso é outra história.

# 2

## Métodos numéricos para o Cálculo de uma variável

---

### 2.1 – Aproximações de diferenças finitas e suas ordens

---

**Definição 2.1** (“big O”) Dada  $g(x) > 0$ , nós dizemos que

$$f(x) = \mathcal{O}(g(x)) \text{ quando } x \rightarrow a$$

se existem números positivos  $\delta$  e  $M$  tais que para  $\forall x, 0 < |x - a| < \delta$ ,

$$|f(x)| \leq Mg(x)$$

ou

$$\lim_{x \rightarrow a} \frac{|f(x)|}{g(x)} < \infty.$$

---

**Definição 2.2** (“little o”) Dada  $g(x) > 0$ , nós dizemos que

$$f(x) = o(g(x)) \text{ quando } x \rightarrow a$$

se

$$\lim_{x \rightarrow a} \frac{|f(x)|}{g(x)} = 0.$$

---

A notação da definição 2.1 pode ser usada para expansões em séries de Taylor. Se

$$u(x_0 + \Delta x) = u(x_0) + \frac{du(x_0)}{dx} \Delta x + \frac{1}{2!} \frac{d^2 u(x_0)}{dx^2} \Delta x^2 + \dots,$$

com  $\Delta x > 0$ , então podemos rearrumar a expressão acima para

$$\frac{du(x_0)}{dx} = \frac{u(x_0 + \Delta x) - u(x_0)}{\Delta x} + \mathcal{O}(\Delta x). \quad (2.1)$$

Dizemos que (2.1) é uma aproximação de diferenças finitas progressiva de  $du/dx$  com “erro” de  $\Delta x$ , ou da “ordem de”  $\Delta x$ . Trocando-se  $\Delta x$  por  $-\Delta x$  em (2.1) obtém-se uma aproximação regressiva:

$$\frac{du(x_0)}{dx} = \frac{u(x_0) - u(x_0 - \Delta x)}{\Delta x} + \mathcal{O}(\Delta x). \quad (2.2)$$

Aproximações de maior ordem sempre podem ser obtidas. Por exemplo, se fizermos duas expansões em torno de  $x_0$ ,

$$u(x_0 + \Delta x) = u(x_0) + \frac{du(x_0)}{dx} \Delta x + \frac{1}{2!} \frac{d^2u(x_0)}{dx^2} \Delta x^2 + \frac{1}{3!} \frac{d^3u(x_0)}{dx^3} \Delta x^3 + \dots, \quad (2.3)$$

$$u(x_0 - \Delta x) = u(x_0) - \frac{du(x_0)}{dx} \Delta x + \frac{1}{2!} \frac{d^2u(x_0)}{dx^2} \Delta x^2 - \frac{1}{3!} \frac{d^3u(x_0)}{dx^3} \Delta x^3 + \dots, \quad (2.4)$$

e subtrairmos a 2ª da 1ª,

$$u(x_0 + \Delta x) - u(x_0 - \Delta x) = 2 \frac{du(x_0)}{dx} \Delta x + \frac{2}{3!} \frac{d^3u(x_0)}{dx^3} \Delta x^3 + \dots,$$

ou

$$\frac{du(x_0)}{dx} = \frac{u(x_0 + \Delta x) - u(x_0 - \Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2). \quad (2.5)$$

Nós dizemos que (2.5) é uma aproximação de diferenças finitas centradas de erro, ou de ordem,  $\Delta x^2$ .

Por outro lado, a soma de (2.3) e (2.4) produz

$$u(x_0 + \Delta x) + u(x_0 - \Delta x) = 2u(x_0) + \frac{2}{2!} \frac{d^2u(x_0)}{dx^2} \Delta x^2 + \frac{2}{4!} \frac{d^4u(x_0)}{dx^4} \Delta x^4 + \dots,$$

ou

$$\frac{d^2u(x_0)}{dx^2} = \frac{u(x_0 + \Delta x) - 2u(x_0) + u(x_0 - \Delta x)}{\Delta x^2} + \mathcal{O}(\Delta x)^2 \quad (2.6)$$

(a aproximação centrada de ordem 2 para a segunda derivada), que é uma das aproximações mais utilizadas em métodos numéricos.

## 2.2 – Solução numérica de equações diferenciais ordinárias

Considere uma equação diferencial de 1ª ordem simples, forçada eternamente por um seno:

$$\frac{dy}{dx} + \frac{y}{x} = \operatorname{sen}(x). \quad (2.7)$$

A solução geral é da forma

$$y(x) = \frac{\operatorname{sen}(x) - x \cos(x) + c}{x}. \quad (2.8)$$

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

$$y(x) = \frac{\operatorname{sen}(x)}{x} - \cos(x). \quad (2.9)$$

Agora,

$$\lim_{x \rightarrow 0} \frac{\operatorname{sen}(x)}{x} = 1,$$

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

$$\lim_{x \rightarrow 0} \left[ \frac{\operatorname{sen}(x)}{x} - \cos(x) \right] = 1 - 1 = 0. \quad (2.10)$$

A solução partindo de  $y(0) = 0$  está mostrada na figura 2.1. Claramente, existe uma parte “transiente” da solução, dada por  $\operatorname{sen}(x)/x$ , 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.

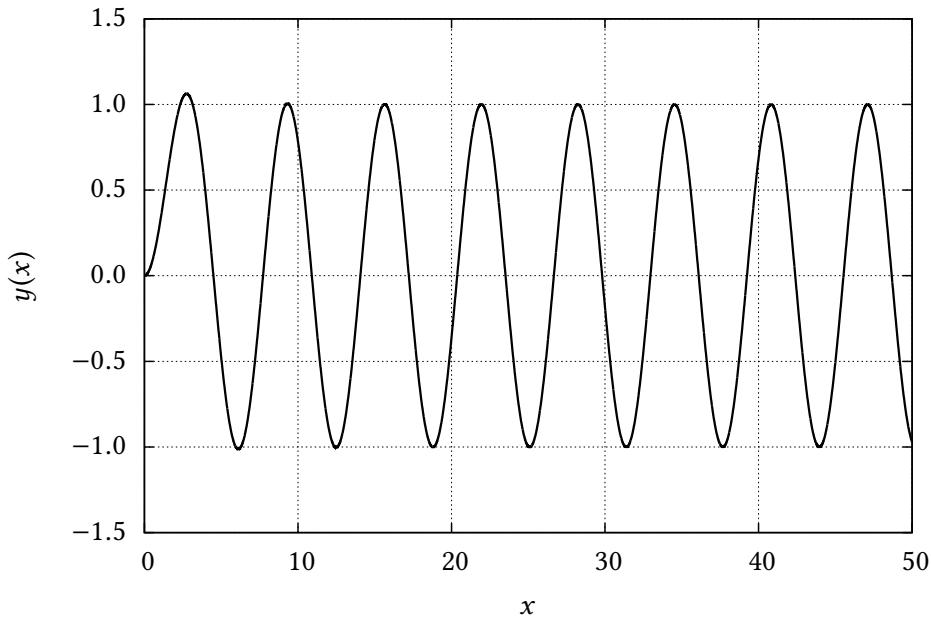


Figura 2.1: Solução da equação (2.7) para  $y(0) = 0$ .

### 2.3 – Solução numérica; o método de Euler de ordem 1

Here Euler 1 says the world will be free.

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

$$\frac{\Delta y}{\Delta x} + \frac{y}{x} = \sin(x).$$

Isso 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  $\Delta x$ ), 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

$$x_n = n\Delta x,$$

com os correspondentes

$$y_0, y_1, \dots, y_N.$$

Como  $\Delta x$  será fixo, podemos escrever nossa equação de diferenças finitas da seguinte forma:

$$\frac{y_{n+1} - y_n}{\Delta x} + \frac{y}{x} = \sin(x),$$

onde eu deixei, propositadamente,

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

*ainda* sem índices. De fato: qual  $x_n$  e qual  $y_n$  usar aqui? A coisa mais simples, mas também a mais *instável*, é usar  $n$ :

$$\frac{y_{n+1} - y_n}{\Delta x} + \frac{y_n}{x_n} = \sin(x_n).$$

---

Listagem 2.1: `fracasso.chpl` – Um programa com o método de Euler que não funciona

---

```

1 // -----
2 // fracasso: um programa que fracassa
3 //
4 use Math only round, sin;
5 use IO only openWriter;
6 var dx = 0.01;
7 var NN = round(50/0.01):int;
8 writeln("NN = ",NN);
9 var x,y: [0..NN] real;
10 x[0] = 0.0;
11 y[0] = 0.0;
12 for n in 0..NN-1 do {           // de 0 até ... n-1 !!!!
13     var xnew = (n+1)*dx;
14     var ynew = y[n] + (sin(x[n]) - y[n]/x[n])*dx;
15     x[n+1] = xnew ;
16     y[n+1] = ynew;
17 }
18 const fou = openWriter("fracasso.out",locking=false);
19 for n in 0..NN do {
20     fou.writef("%12.6dr %12.6dr\n",x[n],y[n]);
21 }
22 fou.close();

```

---

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

$$y_{n+1} = y_n + \left[ \sin(x_n) - \frac{y_n}{x_n} \right] \Delta x. \quad (2.11)$$

Esse é um exemplo de um esquema *progressivo* de diferenças finitas: o novo valor da função em  $x_{n+1}$  ( $y_{n+1}$ ) só depende de valores calculados no valor anterior,  $x_n$ . 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 isso, entretanto, e nosso primeiro programa para tentar resolver a equação diferencial numericamente se chamará `fracasso.chpl`, e está mostrado na listagem 2.1. O resultado é o seguinte fracasso: o programa gera o arquivo `fracasso.out`, cujas primeiras linhas são

---

Listagem 2.2: `fracasso.out` – Saída do programa `fracasso`.

---

0.000000	0.000000
0.010000	nan
0.020000	nan
0.030000	nan
0.040000	nan

---

Isso já era previsível: quando  $n == 0$  no loop,  $x[0] == 0$  no denominador, e o programa produz uma divisão por zero, cujo resultado é *nan* (*not a number*). Para conseguir fazer o método numérico funcionar, nós vamos precisar de mais *análise*!

De volta à equação diferencial, na verdade é possível conhecer uma boa parte do comportamento próximo da origem de  $y(x)$  (a solução de (2.7) para a condição inicial  $y(0) = 0$ ) sem resolvê-la! Para tanto, expanda tanto  $y(x)$  quanto  $\sin(x)$  em série de Taylor em torno de  $x = 0$ :

$$\begin{aligned} y &= a + bx + cx^2 + \dots, \\ \frac{dy}{dx} &= b + 2cx + \dots, \\ \sin(x) &= x + \dots \end{aligned}$$

Substituindo na equação diferencial,

$$b + 2cx + \frac{a}{x} + b + cx + \dots = x + \dots$$

Note que a série de Taylor de  $\sin(x)$  foi truncada corretamente, porque o maior expoente de  $x$  em ambos os lados da equação acima é 1. Simplificando,

$$\frac{a}{x} + 2b + 3cx + \dots = x + \dots$$

A igualdade acima só é possível se  $a = 0$  e  $b = 0$ ; neste caso,  $c = 1/3$ . Esse é o valor correto! De fato, a expansão em série de Taylor da solução analítica em torno de  $x = 0$  dá

$$\sin(x)/x - \cos(x) = \frac{x^2}{3} - \frac{x^4}{30} + \frac{x^6}{840} - \dots$$

Esse resultado nos informa que, próximo da origem,  $y(x)$  “se comporta como”  $x^2/3$ . Nós vamos utilizar a notação

$$x \rightarrow 0 \Rightarrow y \sim x^2/3 \quad (2.12)$$

para indicar esse fato.

Na verdade, (2.12), obtida sem recurso à solução analítica, é suficiente para tratar numericamente o problema da singularidade na origem. Note que

$$\lim_{x \rightarrow 0} \frac{y(x)}{x} = \lim_{x \rightarrow 0} \frac{1}{3} \frac{x^2}{x} = x/3 = 0;$$

Vamos então reescrever a equação de diferenças (2.11) 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, isso 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.chpl`, que pode ser visto na listagem 2.3.

A saída de `sucesso.chpl` gera o arquivo `sucesso-0.01.out`, que nós utilizamos para plotar uma comparação entre a solução analítica e a solução numérica, mostrada na figura 2.2.

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.chpl` calculou foi o *erro absoluto relativo médio*

$$\epsilon \equiv \sum_{n=1}^N \left| \frac{y_n - y(x_n)}{y(x_n)} \right|$$

(onde  $y_n$  é a solução numérica, e  $y(x_n)$  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  $\Delta x$  bem pequeno, e de  $50/0,01 = 5000$  pontos para gerar a solução. Será possível gerar uma solução tão boa com, digamos, 100 pontos?

A figura 2.3 mostra o resultado de rodar `sucesso.chpl` 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 2.3 faz parecer à primeira vista!

Listagem 2.3: `sucesso.chpl` – Um programa com o método de Euler que funciona

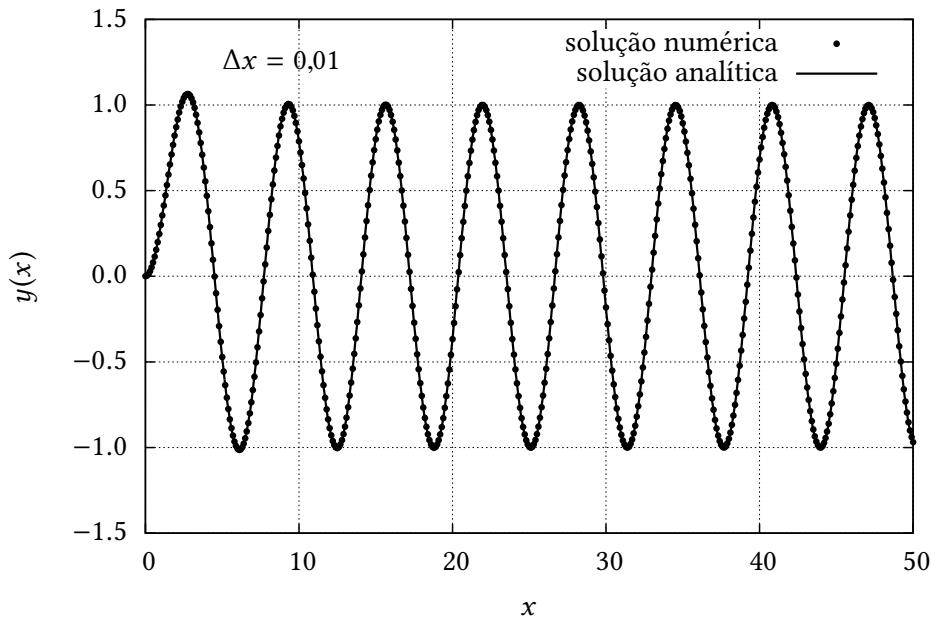
---

```

1 // -----
2 // sucessos: um programa que funciona
3 // -----
4 use Math only cos, round, sin;
5 use IO only openWriter;
6 config const dx = 0.01;
7 const sdx = dx:string;
8 var NN = round(50/0.01):int;
9 writeln("NN = ",NN);
10 var x,y: [0..NN] real;
11 x[0] = 0.0; y[0] = 0.0;
12 x[1] = dx; y[1] = 0.0;
13 for n in 1..NN-1 do {
14     var xnew = (n+1)*dx;
15     var ynew = y[n] + (sin(x[n]) - y[n]/x[n])*dx;
16     x[n+1] = xnew ;
17     y[n+1] = ynew;
18 }
19 const fou = openWriter("sucesso-"+sdx+".out",locking=false);
20 for n in 0..NN do {
21     fou.writef("%12.6dr %12.6dr\n",x[n],y[n]);
22 }
23 fou.close();
24 var erro = 0.0;
25 for n in 1..NN do {                                // calcula o erro relativo médio
26     var yana = sin(x[n])/x[n]-cos(x[n]);
27     erro += abs((y[n]-yana)/yana);
28 }
29 erro /= NN ;
30 writeln("erro relativo médio = %10.5dr \n\n",erro);

```

---

Figura 2.2: Comparação da solução analítica da equação (2.7) com a saída de `sucesso.chpl`, para  $\Delta x = 0,01$ .

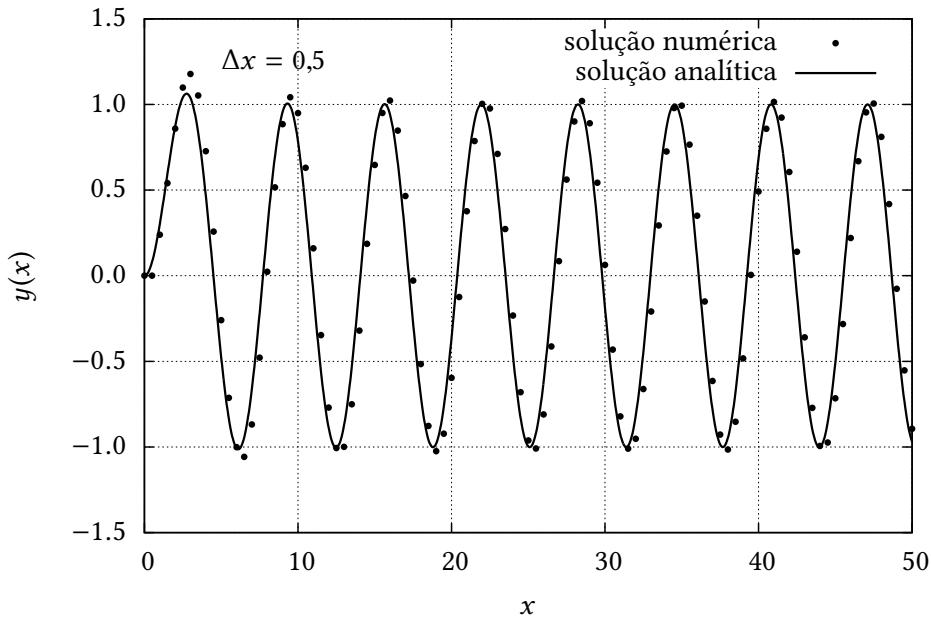


Figura 2.3: Comparação da solução analítica da equação (2.7) com a saída de sucesso .chpl, para  $\Delta x = 0,5$ .

## 2.4 – Um esquema de diferenças centradas, com tratamento analítico

Nosso desafio é desenvolver um método numérico que melhore consideravelmente a solução mesmo com um  $\Delta x$  grosso, da ordem de 0,5. Nossa abordagem será propor um esquema *centrado*:

$$\frac{y_{n+1} - y_n}{\Delta x} + \frac{y_n + y_{n+1}}{x_n + x_{n+1}} = \operatorname{sen}\left(\frac{x_n + x_{n+1}}{2}\right).$$

Note que tanto o termo  $y/x$  quanto  $\operatorname{sen}(x)$  estão sendo agora avaliados no ponto *médio* entre  $x_n$  e  $x_{n+1}$ .

Lembrando que  $\Delta x = x_{n+1} - x_n$ ,

$$\begin{aligned} \frac{y_{n+1} - y_n}{\Delta x} + \frac{y_n + y_{n+1}}{x_n + x_{n+1}} &= \operatorname{sen}\left(\frac{x_n + x_{n+1}}{2}\right), \\ y_{n+1} \left[ \frac{1}{\Delta x} + \frac{1}{x_n + x_{n+1}} \right] + y_n \left[ -\frac{1}{\Delta x} + \frac{1}{x_n + x_{n+1}} \right] &= \operatorname{sen}\left(\frac{x_n + x_{n+1}}{2}\right), \\ y_{n+1} \left[ \frac{2x_{n+1}}{\Delta x(x_{n+1} + x_n)} \right] - y_n \left[ \frac{2x_n}{\Delta x(x_{n+1} + x_n)} \right] &= \operatorname{sen}\left(\frac{x_n + x_{n+1}}{2}\right). \end{aligned}$$

Uma rápida rearrumação produz

$$\begin{aligned} y_{n+1}x_{n+1} - y_nx_n &= \frac{\Delta x(x_{n+1} + x_n)}{2} \operatorname{sen}\left(\frac{x_n + x_{n+1}}{2}\right), \\ y_{n+1} &= y_n \frac{x_n}{x_{n+1}} + \frac{\Delta x(x_{n+1} + x_n)}{2x_{n+1}} \operatorname{sen}\left(\frac{x_n + x_{n+1}}{2}\right). \end{aligned} \tag{2.13}$$

Repare que a condição inicial  $y(0) = 0$  não produz nenhuma singularidade em (2.13) para  $i = 0 \Rightarrow x_0 = 0$ ,  $y_0 = 0$ , pois os denominadores em (2.13) não contêm  $x_i$ . O programa que implementa esse esquema é o `succent.chpl`, mostrado na listagem 2.4.

O resultado é um sucesso mais estrondoso ainda, e pode ser visto na figura 2.4.

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  $\Delta x$  50 vezes menor!

Listagem 2.4: succent.chpl – Método de Euler implícito

---

```

1 // -----
2 // succent: resolve a equação diferencial
3 // dy/dx + y/x = sen(x)
4 // usando um esquema centrado, "sob medida"
5 // -----
6 use Math only sin, cos;
7 use IO only openWriter;
8 const dx = 0.5;           // passo em x
9 const NN = round(50/dx):int; // número de passos
10 var
11     x,                      // variáveis independentes
12     y:                      // variáveis dependentes
13     [0..NN] real;
14 x[0] = 0.0;               // x inicial
15 y[0] = 0.0;               // y inicial
16 for n in 0..NN-1 do {      // loop na solução numérica
17     var xn1 = (n+1)*dx;
18     var xm = x[n] + dx/2.0;
19     var yn1 = y[n]*x[n]/xn1 + (dx*xm/xn1)*sin((x[n]+xn1)/2);
20     x[n+1] = xn1;
21     y[n+1] = yn1;
22 }
23 var erro = 0.0;
24 for n in 1 .. NN do {      // calcula o erro relativo médio
25     var yana = sin(x[n])/x[n] - cos(x[n]);
26     erro += abs( (y[n] - yana)/yana );
27 }
28 erro /= NN ;
29 writef("erro relativo médio = %10.5dr", erro);
30 writeln();
31 const fou = openWriter("succent.out", locking=false);
32 for n in 0..NN do {        // imprime o arquivo de saída
33     fou.writef("%12.6dr %12.6dr\n", x[n], y[n]);
34 }
35 fou.close();

```

---

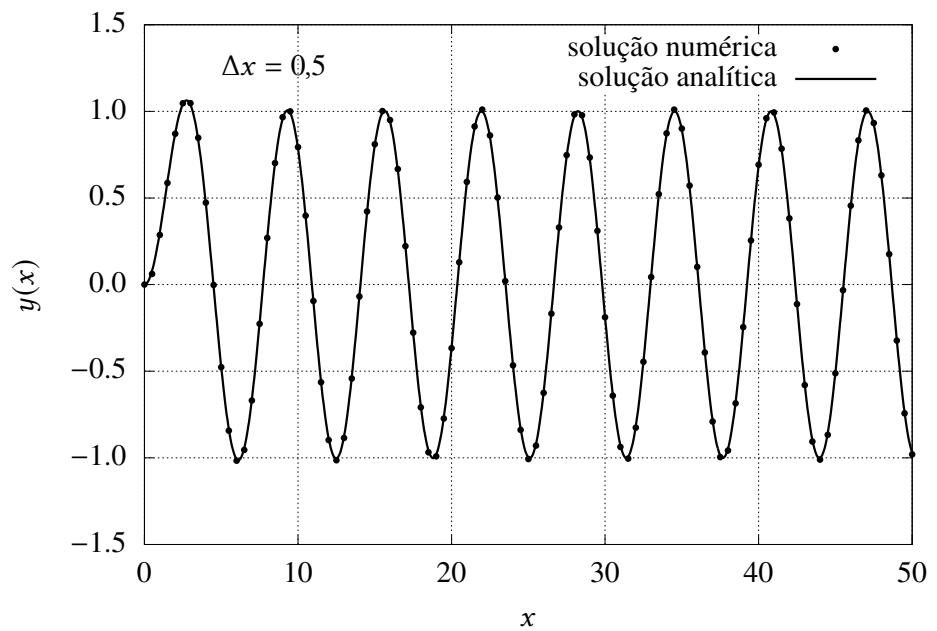


Figura 2.4: Comparação da solução analítica da equação (2.7) com a saída de `succent.chpl`, para  $\Delta x = 0,5$ .

## 2.5 – A forma padrão $dy/dx = f(x, y)$ : os métodos de Euler de ordem 1 e 2, e de Runge-Kutta de ordem 4

O preço que nós pagamos pelo método extremamente acurado implementado em `succent.chpl` foi trabalhar analiticamente a equação diferencial, até chegar à versão “dedicada” (2.13). Isso 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.

Considere portanto a equação

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

Se voltarmos ao método de Euler de ordem 1 apresentado na seção 2.3, podemos reescrevê-lo na forma

$$\begin{aligned}\frac{\Delta y}{\Delta x} &= f(x, y), \\ \frac{y_{n+1} - y_n}{\Delta x} &= f(x_n, y_n).\end{aligned}$$

Mudando um pouco a notação para uma forma mais usual, fazemos  $\Delta x = h$  e explicitamos  $y_{n+1}$ :

$$y_{n+1} = y_n + \underbrace{hf(x_n, y_n)}_{k_1};$$

finalmente, reescrevemos o método de Euler de ordem 1 usando uma notação padrão que será estendida para os métodos de Euler de ordem 2, e de Runge-Kutta de ordem 4:

$$\begin{aligned}k_1 &= hf(x_n, y_n), \\ y_{n+1} &= y_n + k_1.\end{aligned}\tag{2.14}$$

O método de Euler de ordem está ilustrado na figura 2.5 (que também ilustra o método de Euler de ordem 2, que vem a seguir). Na figura, o método consiste em dar um passo de tamanho  $h$ , a partir do ponto  $P$ , cuja abscissa é  $x_n$ , utilizando a inclinação da função neste ponto,  $dy(x_n)/dx$ . Por simplicidade, estamos colocando  $P$  *exatamente* sobre a curva  $y(x)$ , mas isso só é verdade em uma solução numérica quando  $P$  é a condição inicial. A partir daí, nós vamos “errando” a cada passo, e não estamos mais sobre a solução verdadeira  $y(x)$ . Na figura 2.5 nós não explicitamos esse fato, para não sobrecarregá-la. No método de Euler de ordem 1, o valor estimado da função  $y(x)$  em  $x_{n+1}$  é dado pela ordenada do ponto  $C$ .

Claramente, utilizar a derivada “no início” do intervalo, em  $P$ , introduz um erro relativamente grande (note que estamos tentando chegar ao ponto  $A$ , cuja ordenada é o valor exato de  $y(x_{n+1})$ ). Nós podemos tentar estimar a derivada “no meio” do intervalo  $h$ , que é claramente uma alternativa melhor:

$$\left. \frac{dy}{dx} \right|_{x_n+h/2} = f(x_n + h/2, y(x_n + h/2)).\tag{2.15}$$

O problema é que nós não conhecemos  $y$  em  $x + h/2$ ! Se soubéssemos, conheceríamos exatamente a ordenada do ponto  $M$ : utilizando a inclinação da função neste ponto (que é a inclinação da reta tangente no ponto  $M$ ), nós partíramos de  $P$  e chegariam, com um passo  $h$ , em  $A'$ : o segmento  $PA'$  é paralelo à reta tangente por  $M$ . Note que nada garante que a inclinação de  $y(x)$  em  $M$  produz *exatamente* o incremento necessário para atingir o ponto  $A$ ; portanto, *mesmo que conhecêssemos M, ainda assim em geral atingiríamos um ponto A' diferente de A*. Como não conhecemos  $M$ , entretanto, precisamos utilizar o método de Euler de 1ª ordem para *primeiro* estimar  $y(x_n + h/2)$ :

$$y_{n+1/2} \approx y_n + hf(x_n, y_n)/2.\tag{2.16}$$

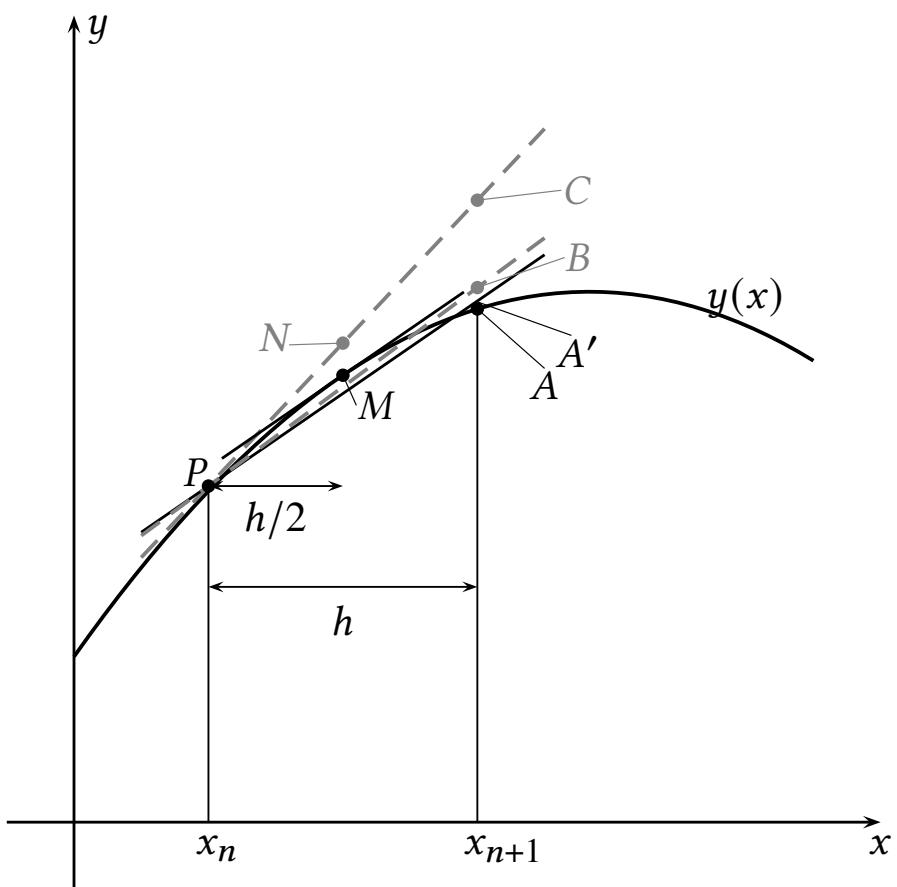


Figura 2.5: Os métodos de Euler de ordens 1 e 2.

Isso corresponde a estimar  $y_{n+1/2}$  com a ordenada do ponto  $N$ . Usando a inclinação estimada utilizando (2.16) em (2.15), obtemos a inclinação do segmento de reta cinza tracejado que parte de  $P$  e vai até  $B$ . Nossa estimativa de  $y_{n+1}$  agora é a ordenada de  $B$ . Podemos resumir todo esse procedimento com

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

Ele se denomina método de Euler de 2<sup>a</sup> ordem. Vamos tentar esse método e ver como ele se compara com nossos esforços anteriores. Vamos manter  $h = 0,5$  como antes. No entanto, nós ainda sofremos do cálculo da derivada em  $x = 0$ ; por isso, 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*, isso 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.chpl` (listagem 2.5), fica mais simples e fácil de entender, e essa é nossa prioridade aqui. O resultado é mostrado na figura 2.6.

Listagem 2.5: `euler2` – Um método explícito de ordem 2

```

1 // -----
2 // euler2: resolve a equação diferencial
3 // dy/dx + y/x = sen(x)
4 // usando um método explícito de ordem 2 (Euler)
5 // -----
6 const h = 0.5;           // passo em x
7 const n = round(50/h):int; // número de passos
8 use Math only sin, cos;
9 use IO only openWriter;
10 var
11     x,
12     y:
13     [0..n] real;
14 x[0] = 0.0;              // x inicial
15 y[0] = 0.0;              // y inicial
16 // -----
17 // ff define a equação diferencial
18 // -----
19 proc ff(
20     const in x: real,
21     const in y: real
22 ): real {
23     if x == 0.0 then {      // implementa a condição inicial
24         return 0.0;
25     }
26     else {
27         return sin(x) - y/x;
28     }
29 }
30 // -----
31 // --> eul2: integra a EDO definida pela função af
32 // -----
33 proc eul2(
34     const in x: real,
35     const in y: real,
36     const in h: real,
37     const ref af: proc(
38         const in ax: real,
39         const in ay: real
40     ): real
41 ): real {
42     var k1 = h*af(x,y);
43     var k2 = h*af(x+h/2,y+k1/2);

```

```

44     var yn = y + k2;
45     return yn;
46 }
47 for i in 0..n-1 do {           // loop da solução numérica
48     var xn1 = (i+1)*h;
49     var yn1 = eul2(x[i],y[i],h,ff);
50     x[i+1] = xn1;
51     y[i+1] = yn1;
52     writeln(xn1, " ",yn1);
53 }
54 var erro = 0.0;                // calcula o erro relativo médio
55 for i in 1..n do {
56     var yana = sin(x[i])/x[i] - cos(x[i]);
57     erro += abs( (y[i] - yana)/yana );
58 }
59 erro /= n ;
60 writef(" erro relativo médio = %10.5dr", erro);
61 writeln();
62 const fou = openWriter("euler2.out", locking=false);
63 for i in 0..n do {           // imprime o arquivo de saída
64     fou.writef("%12.6dr %12.6dr\n",x[i],y[i]);
65 }
66 fou.close();

```

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 4<sup>a</sup> ordem! Não vamos deduzir as equações, mas elas seguem uma lógica parecida com a do método de ordem 2:

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

Para o nosso bem conhecido problema, o método é implementado no programa `rungek4`, na listagem 2.6, e o resultado é mostrado na figura 2.7.

Listagem 2.6: `rungek4` – Método de Runge-Kutta, ordem 4

```

1 // -----
2 // rungek4: resolve a equação diferencial
3 // dy/dx + y/x = sen(x)
4 // usando o método de Runge-Kutta de ordem 4
5 // -----
6 use Math only sin, cos;
7 use IO only openWriter;
8 const h = 0.5;                  // passo em x
9 const n = round(50/h):int;      // número de passos
10 var
11     x,                         // variável independente
12     y:                          // variável dependente
13     [0..n] real;
14     x[0] = 0.0;                 // x inicial
15     y[0] = 0.0;                 // y inicial
16 // -----
17 // função que define a EDO dy/dx = sen(x) - y/x
18 // -----
19 proc ff(

```

```

20 const in x: real,
21 const in y: real
22 ): real {
23 if x == 0.0 then {
24     return 0.0 ;
25 }
26 else {
27     return sin(x) - y/x ;
28 }
29 }
30 // -----
31 // rk4 implementa um passo do método de Runge-Kutta de ordem 4
32 //
33 proc rk4(
34     const in x: real,
35     const in y: real,
36     const in h: real,
37     const ref af: proc(
38         const in ax: real,
39         const in ay: real
40         ): real
41     ): real {
42     var k1 = h*ff(x,y);
43     var k2 = h*ff(x+h/2,y+k1/2);
44     var k3 = h*ff(x+h/2,y+k2/2);
45     var k4 = h*ff(x+h,y+k3);
46     var yn = y + k1/6.0 + k2/3.0 + k3/3.0 + k4/6.0;
47     return yn;
48 }
49 for i in 0..n-1 do {           // loop da solução numérica
50     var xn1 = (i+1)*h;
51     var yn1 = rk4(x[i],y[i],h,ff);
52     x[i+1] = xn1;
53     y[i+1] = yn1;
54 }
55 var erro = 0.0;                // calcula o erro relativo médio
56 for i in 1..n do {
57     var yana = sin(x[i])/x[i] - cos(x[i]);
58     erro += abs( (y[i] - yana)/yana );
59 }
60 erro /= n ;
61 writef("erro relativo médio = %10.5dr",erro);
62 writeln();
63 const fou = openWriter("rungek4.out",locking=false);
64 for i in 0..n do {           // imprime o arquivo de saída
65     fou.writef("%12.6dr %12.6dr\n",x[i],y[i]);
66 }
67 fou.close();

```

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.

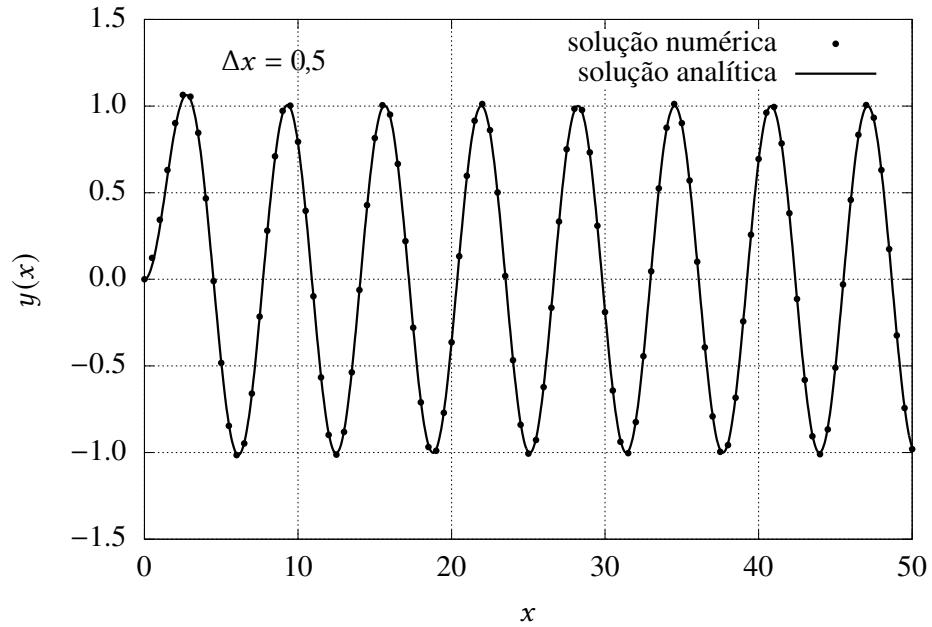


Figura 2.6: Comparaçāo da solução analítica da equaçāo (2.7) com a saída de euler2.chpl, para  $\Delta x = 0,5$ .

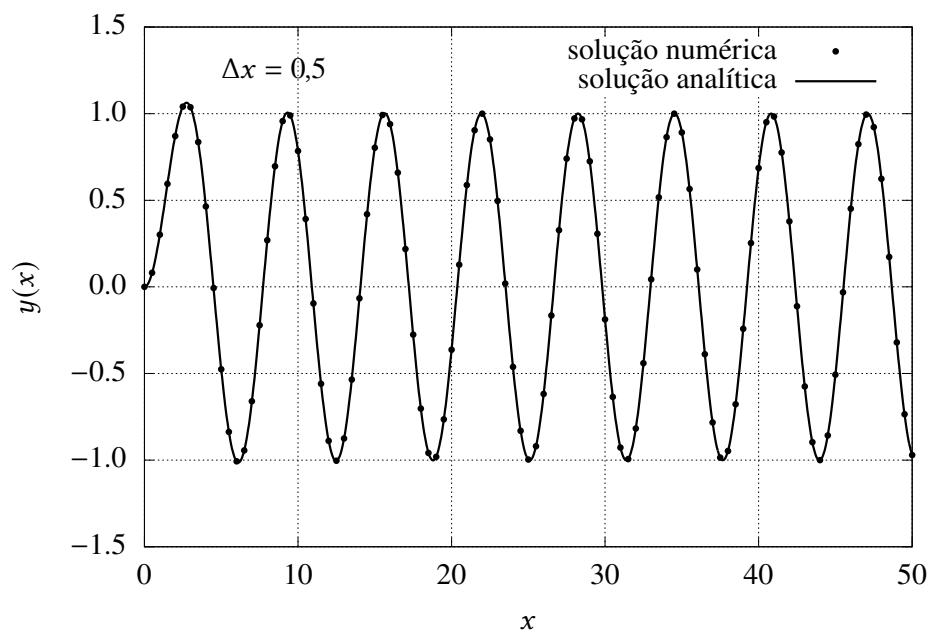


Figura 2.7: Comparaçāo da solução analítica da equaçāo (2.7) com a saída de rungek4.chpl, para  $\Delta x = 0,5$ .

## 2.6 – Interlúdio sobre arrays

### Interlúdio

Aqui, basta discutir alguns aspectos de arrays em Chapel, usando `learnarray.chpl`:

Listagem 2.7: `learnarray` – Alguns aspectos de arrays em Chapel

```

1 // -----
2 // learnarray: como declarar e usar arrays em chapel (só o começo)
3 // -----
4 use Random only fillRandom;
5 const n = 100_000;
6 var dom = {1..n};           // em Chapel, domínios podem ser variáveis
7 var a: [dom] real;         // e arrays são declarados "sobre" domínios
8 var b: [dom] real;
9 fillRandom(a);             // gera um array com 100_000 números aleatórios
10 var (am,av) = stat2(a);
11 writeln(am, " ", av);
12 dom = {0..<n>;           // domínios podem mudar dinamicamente
13 b = 1.0;
14 var (bm,bv) = stat2(b);
15 writeln("b size = ", b.size);
16 writeln(bm, " ", bv);
17 dom = {1..6};              // domínios podem mudar dinamicamente
18 a = [1.0,2.0,3.0,4.0,5.0,6.0];
19 (am,av) = stat2(a);
20 writeln(am, " ", av);
21 // -----
22 // stat2: calcula estatísticas de ordem 1 e 2 de um array
23 // -----
24 proc stat2(
25     const ref x: [] real      // the data
26     ): (real,real)            // mean, variance
27 where x.rank == 1 {
28     // -----
29     // evite arrays vazios
30     // -----
31     var n = x.size;
32     if n == 0 then {
33         halt("-->stat2: array vazio");
34     }
35     // -----
36     // calcula a média
37     // -----
38     var xdom = x.indices;
39     writeln("xdom = ", xdom);
40     var soma = 0.0;
41     for i in xdom do {
42         soma += x[i];
43     }
44     var xm = soma/n;
45     // -----
46     // calcula a variância
47     // -----
48     soma = 0.0;
49     for i in xdom do {
50         soma += (x[i]-xm)**2;
51     }
52     var xv = soma/n;
53     return (xm,xv);
54 }
```

Note que `stat2` é uma rotina (proc) genérica, por causa do argumento `x`: `[] real`. Uma rotina desse tipo não pode ser argumento de outra rotina, tal como feito em `rk4`, que chama outra rotina `af`.

Listagem 2.8: fada — Implementa um vetor (vec) que contorna a limitação de arrays genéricos em argumentos de rotinas.

---

```

1 // =====
2 // ==> fada: attached domain arrays vec (1D) and mat (2D)
3 // =====
4 record vec {
5     var dom: domain(1);           // the 1d domain
6     var arr: [dom] real;         // the array
7     var vfirst = dom.first;      // the first index after reind
8     var vlast = dom.last;        // the last index after reind
9     var vdelta = 0;              // the array shift
10    proc size: int {            // the size of a vec
11        return dom.size;
12    }
13    proc ref reindex(
14        const in dv: range(int)
15        ) {
16        assert ( dv.size == dom.size );
17        vfirst = dv.first;
18        vlast = dv.last;
19        vdelta = vfirst - dom.first;
20    }
21
22    proc ref this(in k: int) ref { // access arr[k]
23        return arr[k-vdelta];
24    }

```

---

Para contornar essa limitação, nós precisamos implementar um tipo especial chamado vec, no módulo `fada.chpl`, cujas primeiras linhas são mostradas da listagem 2.8.

## 2.7 – Uma medida relativa de tempo de processamento

O módulo Time de Chapel permite o cálculo do tempo de processamento de um programa. O tempo de processamento de um mesmo programa pode variar de máquina para máquina, dependendo do tipo de processador, velocidade de acesso à memória, etc.. Por isso, nós vamos procurar calcular (muito aproximadamente!) o tempo *relativo* entre programas e algoritmos diferentes. Faremos isso com um módulo, `unitTime.chpl`, que proporciona uma única rotina `utime`. Ela demora aproximadamente 1 s para rodar, e proporciona uma unidade de medida para programas rodando em máquinas diferentes. O módulo `unitTime` é mostrado na listagem 2.9.

Listagem 2.9: `unitTime` – Uma medida aproximada de tempo de processamento independente da máquina.

---

```
1 // =====
2 // ==> unitTime: a unit for cpu time
3 // =====
4 use Time only stopwatch;
5 use Random only fillRandom;
6 use nsmatrix only dot_mm;
7 const n = 250;
8 proc utime(): real {
9     var
10    a,
11    b,
12    c: [1..n,1..n] real;
13    var runtime: stopwatch;
14    runtime.start();
15    for k in 1..100 do {
16        fillRandom(a);
17        fillRandom(b);
18        dot_mm(a,b,c);
19    }
20    runtime.stop();
21    return runtime.elapsed();
22 }
```

---

## 2.8 – O método de Runge-Kutta multidimensional

antes de começar

Vamos, na sequência, generalizar o método de Runge-Kutta para que ele seja capaz de resolver sistemas de equações diferenciais ordinárias do tipo

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

Note que  $\mathbf{y}$  e  $f$  são **vetores**, enquanto que  $x$  permanece sendo um escalar. Neste livro, **vetores** são escritos em **negrito**:  $\mathbf{v}$  significa um vetor, e isso é *diferente* de um escalar  $v$ ! Não é possível fazer negritos com lápis e canetas: a forma usual em engenharia de denotar um vetor “no papel” é utilizar uma das seguintes notações:  $\vec{v}$  ou  $\underline{v}$ .

A base para a solução de sistemas de equações diferenciais ordinárias com o método de Runge-Kutta é muito simples: basta reconhecer que as equações também “funcionam” vetorialmente! De fato, podemos escrever

$$\begin{aligned} \mathbf{k}_1 &= hf(x_n, \mathbf{y}_n), \\ \mathbf{k}_2 &= hf\left(x_n + \frac{h}{2}, \mathbf{y}_n + \frac{1}{2}\mathbf{k}_1\right), \\ \mathbf{k}_3 &= hf\left(x_n + \frac{h}{2}, \mathbf{y}_n + \frac{1}{2}\mathbf{k}_2\right), \\ \mathbf{k}_4 &= hf(x_n + h, \mathbf{y}_n + \mathbf{k}_3), \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{6}\mathbf{k}_1 + \frac{1}{3}\mathbf{k}_2 + \frac{1}{3}\mathbf{k}_3 + \frac{1}{6}\mathbf{k}_4. \end{aligned}$$

O módulo em Chapel que implementa o método de Runge-Kutta multidimensional é `runkutr.chpl`:

Listagem 2.10: `runkutr` – Implementa o método de Runge-Kutta multidimensional.

```

1 // -----
2 // ==> runkutr: A module for Runge-Kutta integration of systems of ODEs; works
3 // only with vecs called by reference
4 // -----
5 //
6 // --> rk4r: one step of Runge-Kutta integration (all arrays called by reference
7 // as vecs)
8 //
9 use fada;                                // access arr[k]
10 proc rk4r(
11     const in x: real,                      // the position
12     ref y: vec,                            // y is passed by reference
13     const in h: real,                      // the step
14     f: proc(                                // the function that defines the ODE
15         const in ax: real,                  // the position
16         ref ay: vec,                      // the array at current position
17         ref dadx: vec                   // the derivative
18     ),                                     // f is a classical procedure; returns dadx
19     ref ynew: vec                         // ynew is passed and returned by reference
20 ) {
21 //
22 // y and ynew must have the same domain
23 //
24 const D = y.dom;
25 var dydx = new vec(D);                  // we need a local array
26 var k = new vec(D);                    // and another
27 //
28 // 0th part
29 // -----
```

```

30 ynew = y;                                // 0th part of y + k1/6 + k2/3 + k3/3 + k4/6
31 // -
32 // 1st part
33 // -
34 f(x,y,dydx);                            // dydx = f(x,y)
35 k = (h/2.0)*dydx;                      // this is k1/2
36 ynew += k/3.0;                          // 1st part of y + k1/6 + k2/3 + k3/3 + k4/6
37 // -
38 // 2nd part
39 // -
40 k = k + y;                            // y + k1/2
41 f(x+h/2.0,k,dydx);                    // dydx = f(x+h/2,y + k1/2)
42 k = (h/2.0)*dydx;                      // this is k2/2
43 ynew += k*(2.0/3.0);                  // 2nd part of y + k1/6 + k2/3 + k3/3 + k4/6
44 // -
45 // 3rd part
46 // -
47 k = k + y;                            // y + k2/2
48 f(x+h/2,k,dydx);                      // dydx = f(x+h/2,y+k2/2)
49 k = h*dydx;                           // this is k3
50 ynew += k/3.0;                          // 3rd part of y + k1/6 + k2/3 + k3/3 + k4/6
51 // -
52 // 4th part
53 // -
54 k = k + y;                            // y + k3
55 f(x+h,k,dydx);                      // dydx = f(x+h,y+k3)
56 k = h*dydx;                           // this is k4
57 ynew += k/6.0;                          // 4th part of y + k1/6 + k2/3 + k3/3 + k4/6
58 }

```

Vamos agora resolver um caso para o qual possuímos solução analítica. Dado o sistema

$$\begin{aligned}\frac{du_1}{dx} &= u_2, \\ \frac{du_2}{dx} &= u_1,\end{aligned}$$

a sua solução é

$$\begin{aligned} u_1(x) &= k_1 e^{-x} + k_2 e^x, \\ u_2(x) &= -k_1 e^{-x} + k_2 e^x. \end{aligned}$$

O programa que resolve este (pequeno) problema chama-se `rktestr.chpl`:

Listagem 2.11: **rktestr** – Resolve um sistema de EDOs com o método de Runge-Kutta multidimensional.

```

16 const in x: real,
17 ref y: vec,                                // intent is ref: vec fields cannot be const
18 ref ynew: vec                               // intent is ref
19 ) {
20     ynew[1] = y[2];                         // assume both domains are == {1..2}!
21     ynew[2] = y[1];
22 }
23 x[0] = 0.0;                                // initial...
24 y[0] = new vec({1..2}, [1.0, 0.0]);        // ...condition
25 for n in 0..nt-1 do {                      // advance in time
26     x[n+1] = (n+1)*h;
27     rk4r(x[n],y[n],h,ff,y[n+1]);          //
28 }
29 var error1 = 0.0;                           // calculate the mean relative error
30 var error2 = 0.0;
31 var ya: [0..nt] [1..2] real;                // analytical solution (no need for vecs)
32 ya[0] = [1.0, 0.0];
33 use Math only sinh, cosh;
34 for n in 1..nt do {
35     ya[n][1] = cosh(x[n]);
36     ya[n][2] = sinh(x[n]);
37     error1 += abs( (y[n][1] - ya[n][1])/ya[n][1] );
38     error2 += abs( (y[n][2] - ya[n][2])/ya[n][2] );
39 }
40 error1 /= nt;
41 error2 /= nt;
42 writeln("mean relative error = %10.6dr %10.6dr",error1,error2);
43 writeln();
44 use IO only openWriter;
45 const fou = openWriter("rktestr.out", locking=false); // open output file
46 for n in 0..nt do {                         // print results
47     fou.writef("%12.6dr %12.6dr %12.6dr %12.6dr %12.6dr\n",
48             x[n],y[n][1],y[n][2],ya[n][1],ya[n][2]);
49 }
50 fou.close();                                // close output file

```

Com um  $h = 0.1$ , os erros relativos médios de  $u_1$  e  $u_2$  são extremamente pequenos:  $\epsilon = 0.000004$  em ambos os casos. Graficamente, temos a resultado mostrado na figura 2.8. Note que  $\cosh(x) \approx \sinh(x)$  para  $x \gtrsim 2.5$ .

## A onda cinemática

Começamos com a equação de continuidade,

$$\frac{\partial h}{\partial t} + \frac{\partial(vh)}{\partial x} = 0,$$

e simplificamos bastante a equação de Momentum:

$$\cancel{\frac{\partial v}{\partial t}}^0 + v \cancel{\frac{\partial v}{\partial x}}^0 + g \cancel{\frac{\partial h}{\partial x}}^0 = g(S_0 - S_f).$$

Quando  $S_0 = S_f$ , a linha de energia é paralela à linha d'água; o movimento é (localmente) uniforme; podemos integrar o perfil  $v(z)/v_*$ , etc., e obter (por exemplo) a equação de Manning. Em suma, se a equação de momentum admite essa simplificação, então  $v = v(h)$  (existe uma “curva-chave”!) e

$$v = \frac{1}{n} R^{2/3} S_0^{1/2} \approx \frac{1}{n} h^{2/3} S_0^{1/2}.$$

Um ponto importante que é sempre bom relembrar: estamos simplificando a equação de momentum, mas mantendo a equação da continuidade em sua forma completa! Isso acontece frequentemente em

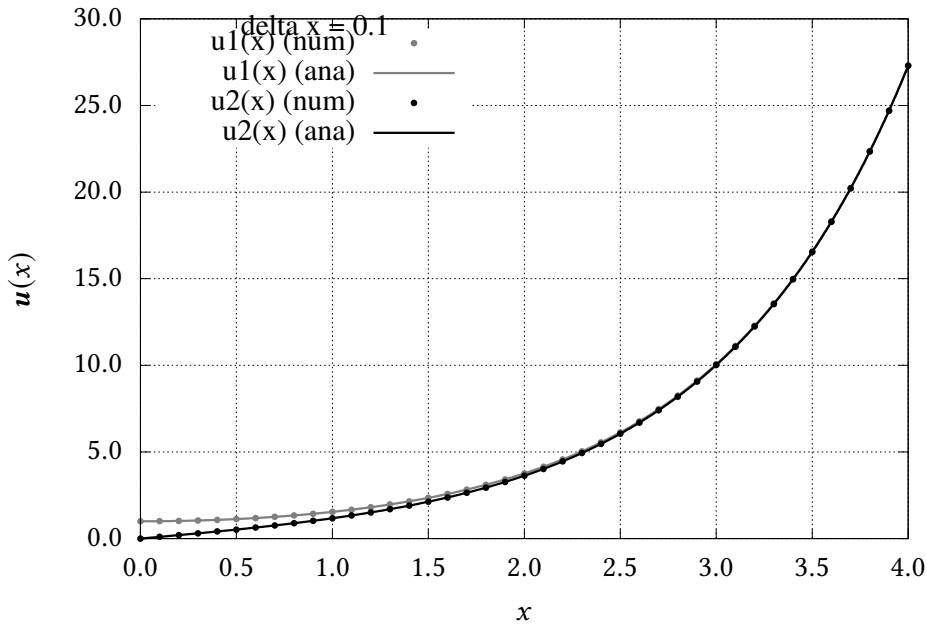


Figura 2.8: Solução numérica pelo Método de Runge-Kutta de um sistema de 2 equações diferenciais ordinárias

Mecânica dos Fluidos, e pode ser rigorosamente justificado se analisarmos as ordens de magnitude *relativas* dos termos em cada equação — mas não vamos fazer isso aqui.

Levando  $v$  na equação da continuidade,

$$\begin{aligned} \frac{\partial h}{\partial t} + \frac{\partial}{\partial x} \left[ \frac{1}{n} h^{5/3} S_0^{1/2} \right] &= 0, \\ \frac{\partial h}{\partial t} + \underbrace{\frac{5}{3} \frac{h^{2/3} S_0^{1/2}}{n}}_{c_h} \frac{\partial h}{\partial x} &= 0. \end{aligned}$$

Note que a celeridade da onda  $h(t)$  é

$$\begin{aligned} c_h &= \frac{5}{3} \frac{h^{2/3} S_0^{1/2}}{n} \\ &= \frac{5}{3} v(h)! \end{aligned}$$

Portanto, a onda de cheia move-se *mais rapidamente* do que a água do rio.

Algumas vezes, é mais prático trabalhar com as variáveis  $Q$  e  $A$  do que com  $v$  e  $h$ . Multiplique a equação da continuidade por  $b$ :

$$\begin{aligned} \frac{\partial(hb)}{\partial t} + \frac{\partial(vhb)}{\partial x} &= 0; \\ \frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} &= 0. \end{aligned}$$

A equação de Manning pode ser reescrita em termos de  $Q = Q(A)$ :

$$\begin{aligned} v &= \frac{1}{n} h^{2/3} S_0^{1/2}, \\ vhb &= Q = \frac{1}{n} b h^{5/3} S_0^{1/2} \end{aligned}$$

$$\begin{aligned} Q &= \frac{1}{n} \frac{b^{2/3}}{b^{2/3}} b^{3/3} h^{5/3} S_0^{1/2} \\ Q &= \frac{1}{nb^{2/3}} [bh]^{5/3} S_0^{1/2} \\ Q &= \frac{1}{nb^{2/3}} A^{5/3} S_0^{1/2} \end{aligned}$$

Escrevemos agora a equação da onda cinemática como

$$\begin{aligned} \frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} &= 0, \\ \frac{dA}{dQ} \frac{\partial Q}{\partial t} + \frac{\partial Q}{\partial x} &= 0, \\ \frac{\partial Q}{\partial t} + \frac{dQ}{dA} \frac{\partial Q}{\partial x} &= 0. \end{aligned}$$

Esta última é uma equação de onda na variável que talvez seja de maior interesse prático,  $Q = Q(x, t)$ . A celeridade da onda cinemática de vazão é

$$\begin{aligned} c_Q &= \frac{dQ}{dA} = \frac{5}{3} \frac{1}{nb^{2/3}} A^{2/3} S_0^{1/2} \\ &= \frac{5}{3} \frac{1}{n} \left( \frac{A}{b} \right)^{2/3} S_0^{1/2} \\ &= \underbrace{\frac{5}{3} \frac{1}{n} h^{2/3} S_0^{1/2}}_{v(h)} \\ &= \frac{5}{3} v(h). \end{aligned}$$

Não surpreendentemente,

$$c_Q = c_h.$$

Concluímos que  $Q$  é uma onda que se propaga solidariamente com  $h$ , o que é, em retrospecto, óbvio. Nossa problema agora é resolver a onda cinemática para  $Q$  numericamente. Existem muitas possibilidades (em princípio, existem infinitas possibilidades).

Vou tentar encontrar uma solução *relativamente simples*. Em primeiro lugar, noto que para resolver a onda cinemática tendo uma única variável dependente  $Q(x, t)$  eu ainda vou precisar de  $A$ ; logo, preciso inverter a equação de Manning:

$$\begin{aligned} Q &= \frac{1}{nb^{2/3}} A^{5/3} S_0^{1/2} \\ Qnb^{2/3} &= A^{5/3} S_0^{1/2} \\ Qnb^{2/3} S_0^{-1/2} &= A^{5/3} \\ A &= \left[ Qnb^{2/3} S_0^{-1/2} \right]^{3/5} \\ c_Q &= \frac{5}{3} \frac{1}{nb^{2/3}} S_0^{1/2} \left\{ \left[ Qnb^{2/3} S_0^{-1/2} \right]^{3/5} \right\}^{2/3} \\ c_Q &= \frac{5}{3} \frac{1}{nb^{2/3}} S_0^{1/2} \left[ Qnb^{2/3} S_0^{-1/2} \right]^{2/5} \\ &= \frac{5}{3} n^{-3/5} b^{-2/5} S_0^{3/10} Q^{2/5}. \end{aligned}$$

Supondo que eu esteja certo, e nunca se sabe, o meu objetivo agora é utilizar o método (explícito, mas acurado) de Runge-Kutta, porque o “pacote” já está semi-pronto. Como sempre, a ideia é discretizar  $Q$ :

$$Q_i^k = Q(i\Delta x, k\Delta t), \quad i = 0, \dots, N_x, \quad k = 0, \dots, N_t.$$

Suponha que eu discretize *primeiro* em  $x$ ; então, deixo de ter uma incógnita contínua em  $x$  e passo a ter  $n + 1$  incógnitas  $Q_i$ . Por enquanto, vamos supor que  $Q_i = Q_i(t)$ :

$$\begin{aligned} \Delta x &= L/N_x, \\ Q_i &= Q(i\Delta x, t) = Q_i(t). \end{aligned}$$

Vamos representar a totalidade dos  $Q_i$ s por um vetor  $\mathbf{Q}$ . A equação da onda se torna

$$\begin{aligned} \frac{dQ_i}{dt} + c_{Q_i} \frac{Q_i - Q_{i-1}}{\Delta x} &= 0, \quad i = 1, \dots, N_x, \\ c_{Q_i} &= \frac{5}{3} n^{-3/5} b^{-2/5} S_0^{3/10} Q_i^{2/5} \\ \frac{dQ_i}{dt} &= - \underbrace{\frac{5}{3} n^{-3/5} b^{-2/5} S_0^{3/10} Q_i^{2/5}}_{K_F} \frac{Q_i - Q_{i-1}}{\Delta x}, \quad i = 1, \dots, N_x \end{aligned}$$

Vetorialmente, agora nós temos

$$\begin{aligned} \frac{d\mathbf{Q}}{dt} &= \mathbf{F}(t, \mathbf{Q}), \\ F_i(\mathbf{Q}, t) &= -K_F Q_i^{2/5} \frac{Q_i - Q_{i-1}}{\Delta x}, \quad i = 1, \dots, N_x. \end{aligned}$$

Note entretanto que as equações acima só podem ser calculadas para  $F_1, \dots, F_{N_x}$ ; entretanto,  $Q_i$  começa em  $i = 0$ ! Para que o método de Runge-Kutta possa ser aplicado, nós precisamos de uma equação para

$$\frac{dQ_0}{dt} = F_0(t, \mathbf{Q}).$$

Note que a onda cinemática pressupõe que nós conhecemos a condição de contorno de jusante,  $Q_0(t)$ . Portanto, essa derivada é conhecida. Discretizemos agora no tempo:

$$\begin{aligned} Q_i^k &= Q(i\Delta x, k\Delta t), \\ F_0^k &= F_0(k\Delta t). \end{aligned}$$

Então

$$\frac{Q_0^k - Q_0^{k-1}}{\Delta t} = F_0^k.$$

Portanto, a cada passo de tempo  $k$ ,  $F_0$  é facilmente calculado, a partir de  $k = 1$ . Isso nos dá a equação para  $F_0$ , que faltava, e efetivamente é a forma de impor a condição de contorno no método de Runge-Kutta. O método em si é simplesmente a sequência de operações vetoriais

$$\begin{aligned} \frac{d\mathbf{Q}}{dt} &= \mathbf{F}(t, \mathbf{Q}), \\ \mathbf{k}_1 &= \Delta t \mathbf{F}(t_k, \mathbf{Q}^k), \\ \mathbf{k}_2 &= \Delta t \mathbf{F}\left(t_k + \frac{\Delta t}{2}, \mathbf{Q}^k + \frac{1}{2}\mathbf{k}_1\right), \\ \mathbf{k}_3 &= \Delta t \mathbf{F}\left(t_k + \frac{\Delta t}{2}, \mathbf{Q}^k + \frac{1}{2}\mathbf{k}_2\right), \\ \mathbf{k}_4 &= \Delta t \mathbf{F}(t_k + \Delta t, \mathbf{Q}^k + \mathbf{k}_3), \end{aligned}$$

$$Q^{k+1} = Q^k + \frac{1}{6}\mathbf{k}_1 + \frac{1}{3}\mathbf{k}_2 + \frac{1}{3}\mathbf{k}_3 + \frac{1}{6}\mathbf{k}_4.$$

Em princípio, nós agora temos tudo o que é necessário para a implementação computacional da onda cinemática.

Um ponto importante, e doloroso, é a estabilidade restrita dos métodos explícitos. Embora não seja proveitoso fazer uma análise de estabilidade detalhada do método numérico que vamos tentar aplicar, é sempre útil olhar para o número de Courant,

$$\text{Co} = \frac{c_Q \Delta t}{\Delta x}$$

e verificar se (em geral)  $\text{Co} \leq 1$ . Como  $c_Q$  depende de  $Q^{2/5}$ , parece suficiente calcular

$$\text{Co}_{\max} = \frac{5}{3} n^{-3/5} b^{-2/5} S_0^{3/10} Q_{\max}^{2/5} \frac{\Delta t}{\Delta x}.$$

Todas essas ideias agora ficam implementadas no programa `oncin.chpl`:

Listagem 2.12: `oncin.chpl` – Propagação de cheia com o método de Runge-Kutta.

```

1 // -----
2 // oncin.chpl: resolve uma onda cinemática com o método de Runge-Kutta
3 //
4 // Nelson Luís Dias
5 //
6 use fada;
7 use runkutr;
8 //
9 // rugosidade e geometria do canal
10 //
11 const n = 0.035;           // coeficiente de Manning
12 const S0 = 0.01;           // declividade
13 var b = 200.0;             // largura, ft
14 b *= 0.3048;              // b em m
15 //
16 // KF pode ser uma variável global
17 //
18 const KF = 5.0*n**(-0.6)*b**(-0.4)*S0**(0.3)/3.0;
19 //
20 // agora eu escolho os parâmetros de minha discretização
21 //
22 var L = 15000.0;           // comprimento do canal, ft
23 L *= 0.3048;               // comprimento do canal, m
24 var Nx = 1500;              // discretização em x
25 var dx = L/Nx;             // deltax, metros
26 var TT = 150*60.0;          // tempo de simulação: 150 min = 150 * 60 s
27 var Nt = 15000;             // discretização em t
28 var dt = TT/Nt;             // deltat, s
29 writeln("L, b, T = ", L, " ", b, " ", TT);
30 writeln(dx, ",dt);        // verifica
31 //
32 // O programa propriamente dito começa aqui
33 //
34 writeln("Tabela de vazões (m³/s):");
35 for it in 0..121*60 by 12*60 do {
36   writef("%6i %7.2dr\n", it/60, Qt(it));
37 }
38 //
39 // Importante! Verifica o número de Courant
40 //
41 var Qtime: [0..Nt] real = 0.0;
42 for k in 0..Nt do {

```

```

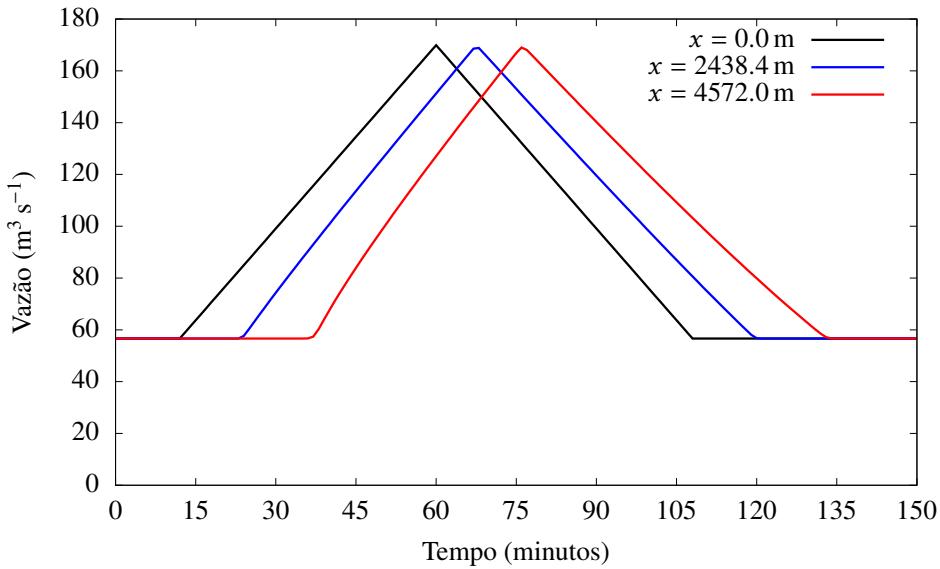
43  var t = k*dt ;
44  Qtime[k] = Qt(t);
45 }
46 var Qmax = amax(Qtime);
47 var cQmax = KF*Qmax**(.4);
48 var Cou = cQmax*dt/dx ;
49 writeln("Cou = ", Cou);
50 assert (Cou < 1.0);
51 // -----
52 // só quero guardar (acadak,acadai) dos dados Q calculados
53 // -----
54 writeln(Nx," ",Nt);
55 const acadak = 100;
56 const acadai = 100;
57 const IL = Nx/acadai;
58 const IT = Nt/acadak;
59 writeln("IT, IL = ",IT,IL);
60 var Qs: [0..IT,0..IL] real;
61 // -----
62 // aloco o vetor QQ
63 // -----
64 var QQ: [0..1] vec;
65 // -----
66 // abaixo, todos os valores iniciais de vazão *em x* são iguais a Qt(0.0)
67 // -----
68 QQ[0] = new vec({0..Nx});           // condição inicial
69 QQ[0].arr = Qt(0.0);              // preenche todos os valores de QQ[0] com
70                      // Qt(0.0)
71 // -----
72 // guarda valores selecionados da condição inicial em Qs[0]
73 // -----
74 for isx in 0..IL do {
75   Qs[0,isx] = QQ[0][isx*acadai];
76 }
77 // -----
78 // prepara o início da simulação
79 // -----
80 var iold = 0;                     // posição das alturas antigas
81 var inew = 1;                     // posição das novas alturas
82 // -----
83 // finalmente, o loop para simulação
84 // -----
85 for k in 1..Nt do {               // loop no tempo
86   var t = k*dt;                  // tempo em segundos
87   writeln(k," ",Nt);            // imprime o tempo
88   rk4r(t,QQ[iold],dt,FF,QQ[inew]); // avança o vetor QQ
89 }
90 // a cada acadak:
91 // -----
92 if k % acadak == 0 then {
93   var ks = k /acadak;
94   for isx in 0..IL do {
95     Qs[ks,isx] = QQ[inew][isx*acadai];
96   }
97 }
98 inew <=> iold ;
99 }
100 // -----
101 // agora imprime, primeiramente, as vazões nas acadai seções
102 // -----
103 use IO only openWriter;
104 const fout = openWriter("oncin-t.out",locking=false);
105 for ks in 0..IT do {

```

```

106  var t = ks*acadak*dt;
107  fout.writef("%8.2dr",t);
108  for isx in 0..IL do {
109      fout.writef("%8.2dr",Qs[ks,isx]);
110  }
111  fout.writef("\n");
112 }
113 fout.close();           // fim de papo!
114 proc FF(
115     const in t: real,
116     ref Q: vec,
117     ref Qnew: vec
118 ) {
119     Qnew[0] = (Qt(t) - Q[0])/dt;    // calcula a derivada no tempo
120 // -----
121 // a forma a seguir é *muito* mais eficiente do que um loop clássico
122 // -----
123     Qnew.arr[1..Nx] =
124         -KF*(Q.arr[1..Nx])**0.4*(Q.arr[1..Nx] - Q.arr[0..Nx-1])/dx ;
125 }
126 // -----
127 // a vazão (cfs!!!--> m3/s) em função do tempo na seção zero
128 // -----
129 proc Qt(const in t: real):real {
130     assert ( t >= 0 );
131     var Q0 = 2000.0;
132     var QM = 6000.0;
133     var delQ = 4000.0/(48.0*60) ;
134     var Qr: real;
135     if t <= (12.0*60) then {
136         Qr = Q0;
137     }
138     else if t <= (60.0*60) then {
139         Qr = Q0 + delQ*(t - 12*60);
140     }
141     else if t <= 108.0*60 then {
142         Qr = QM - delQ*(t - 60*60);
143     }
144     else {
145         Qr = Q0;
146     }
147     return Qr*(0.3048)**3;      // retorno tudo em m3/s
148 }
149 // -----
150 // --> amax: the maximum of a real array
151 // -----
152 proc amax(
153     ref a: [] real
154 ): real {
155     var b = min(real);
156     for x in a do {
157         if x > b then {
158             b = x;
159         }
160     }
161     return b;
162 }
```

A saída pode ser vista aqui, graficamente:



Propagação de uma onda cinemática, método de Runge-Kutta (Exemplo 9.6.1 de ?).

## 2.9 – Problemas de valor de contorno em 1D

Considere o problema de valor de contorno (?), Eq. 4.13)

$$\frac{d^2T}{dx^2} = 0, \quad (2.17)$$

$$T(0) = T_0, \quad (2.18)$$

$$T(L) = T_L, \quad (2.19)$$

para  $T_0 = 100$ ,  $T_L = 500$ ,  $L = 0.5$ . A solução analítica é

$$T_a(x) = 100 + 800x, \quad 0 \leq x \leq 0.5. \quad (2.20)$$

Faça agora

$$\begin{aligned} \Delta x &= \frac{L}{N}, \\ x_i &= i\Delta x, \quad i = 0, \dots, N, \\ T_i &= T(x_i). \end{aligned}$$

Use agora (2.6) em (2.17):

$$\frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} = 0,$$

ou, simplesmente,

$$-T_{i-1} + 2T_i - T_{i+1} = 0. \quad (2.21)$$

Mas  $i$  corre de 0 até  $N$ ; portanto, o menor  $i$  possível acima é 1, e o maior é  $N-1$ . Essas duas extremidades produzem as *condições de contorno*

$$2T_1 - T_2 = T_0, \quad (2.22)$$

$$-T_{N-1} + 2T_{N-1} = T_N. \quad (2.23)$$

Em conjunto, (2.21)–(2.23) produzem o sistema de equações lineares

$$\begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ \vdots & & & & & \vdots \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_{N-2} \\ T_{N-1} \end{bmatrix} = \begin{bmatrix} T_0 \\ 0 \\ \vdots \\ 0 \\ T_L \end{bmatrix}. \quad (2.24)$$

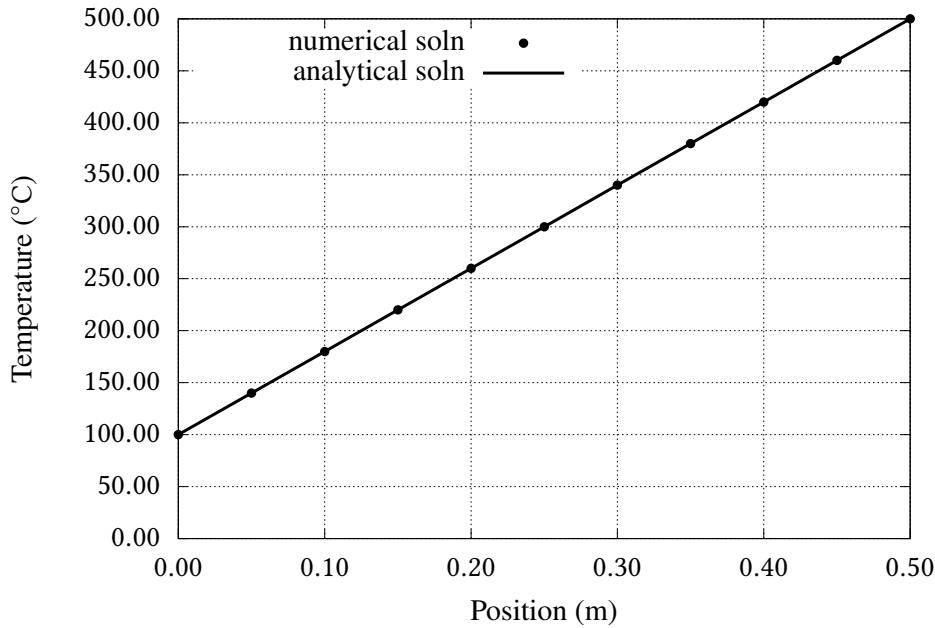


Figura 2.9: Solução numérica de (2.17)–(2.19).

Existem várias coisas atraentes para um programador em (2.24). Em primeiro lugar, a matriz do sistema é uma matriz banda tridiagonal; sistemas lineares com esse tipo de matriz são particularmente simples de resolver, e estão disponíveis na literatura (por exemplo: ?, seção 2.4, subrotina `tridiag`). 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 `tridiag.chpl`, que exporta a função `tridiag`, que resolve um sistema tridiagonal, mostrado na listagem 2.13.

Em `tridiag`, a diagonal inferior é armazenada em `a`, sendo que o elemento `a[1]` não é utilizado; a diagonal principal é armazenada em `b`, e a diagonal superior é armazenada em `c`, sendo que o último elemento dessa linha, `c[n]`, não é utilizado.

Agora vamos escrever um programa que resolve (2.17)–(2.19), na listagem 2.14.

Para  $N = 10$ , a solução é mostrada na figura 2.9

Nós definimos o erro médio absoluto relativo da solução por

$$\epsilon_m \equiv \frac{1}{N-1} \sum_{i=1}^{N-1} \left| \frac{T_i - T_a(i\Delta x)}{T_L - T_0} \right|. \quad (2.25)$$

Em princípio, nós esperamos que o erro da solução numérica *diminua* com o refinamento da grade; no entanto, neste caso isso não acontece, como pode ser visto na figura 2.10. Na figura, o erro aumenta de acordo com  $\epsilon_m \sim N^{1.516}$ . A melhor explicação parece ser que com o aumento do número de pontos, o erro de arredondamento aumenta.

Considere agora o problema de valor de contorno

$$\frac{d^2y}{dx^2} - k^2 y = 0, \quad (2.26)$$

$$y(0) = y_0, \quad (2.27)$$

$$\frac{dy(L)}{dx} = 0. \quad (2.28)$$

A solução é

$$y(x) = A \cosh(kx) + B \sinh(kx),$$

Listagem 2.13: `tridiag.chpl` – Exporta uma rotina que resolve um sistema tridiagonal.

---

```

1 // -----
2 // tridiag implementa uma soluo de um sistema linear com matriz tridiagonal
3 // -----
4 proc tridiag(
5     const ref aa: [] real,
6     const ref ab: [] real,
7     const ref ac: [] real,
8     const ref ay: [] real,
9     ref ax: [] real
10    ) where ( (aa.rank == 1) &&
11                (ab.rank == 1) &&
12                (ac.rank) == 1 &&
13                (ay.rank == 1) && (ax.rank == 1) ) {
14 // -----
15 // reindexing is needed
16 // -----
17 var n = aa.size;
18 assert ((n == ab.size) &&
19          (n == ac.size) &&
20          (n == ay.size) &&
21          (n == ax.size));
22 const ref a = aa.reindex(1..n);
23 const ref b = ab.reindex(1..n);
24 const ref c = ac.reindex(1..n);
25 const ref y = ay.reindex(1..n);
26 ref x = ax.reindex(1..n);
27 var gam: [2..n] real = 0.0 ;
28 if b[1] == 0.0 then {
29     halt("tridiag error 1");
30 }
31 var bet = b[1];
32 x[1] = y[1]/bet;
33 for j in 2..n do {
34     gam[j] = c[j-1]/bet;
35     bet = b[j] - a[j]*gam[j];
36     if bet == 0 then {
37         halt("tridiag error 2");
38     }
39     x[j] = (y[j] - a[j]*x[j-1])/bet;
40 }
41 for j in 1..n-1 by -1 do {
42     x[j] -= gam[j+1]*x[j+1];
43 }
44 }
```

---

Listagem 2.14: edo-l1.chpl – Solução do problema de valor de contorno (2.17)–(2.19).

---

```

1 // -----
2 // edoo-l1: solves d2T/dx2 = 0
3 // -----
4 config const N = 2;                      // start with 11 points
5 config const L = 0.5;                     // and a length of 0.5 m
6 config const T0 = 100.0;                   // 100 degrees C
7 config const TL = 500.0;                   // 500 degrees C
8 const dx = L/N;                         // delta x
9 var x: [0..N] real = [i in 0..N] i*dx;   // this is an expression forall
10 var T: [0..N] real = 0.0;                 // the unknowns
11 var e: [0..N] real = 0.0;                 // the errors
12 var a,b,c,y : [1..N-1] real;            // the tridiag matrix and forcing y
13 a = -1.0;
14 b = +2.0;
15 c = -1.0;
16 T[0] = 100.0;
17 T[N] = 500.0;
18 if N > 2 then {                         // the usual forcing for all N > 2
19     y[1] = T[0];
20     y[N-1] = T[N];
21 }
22 else {                                    // if N == 2, the y vector is different!
23     y[1] = T[0]+T[2];
24 }
25 use tridiag;
26 tridiag(a,b,c,y,T[1..N-1]);
27 const DeltaT = TL - T0;
28 use IO only openWriter;
29 use IO.FormattedIO;
30 var founam = "edo-l1-%05i.out".format(N);
31 const fou = openWriter(founam,locking=false);
32 for i in 0 .. N do {
33     var Ta = anasol(x[i]);
34     e[i] = abs((T[i] - Ta)/DeltaT);
35     fou.writef("%8.4dr %8.4dr %8.4dr %12.4er\n",x[i],T[i],Ta,e[i]);
36 }
37 fou.close();
38 const em = (+ reduce e[1..N-1])/(N-1);
39 writef("%05i %12.4er\n",N,em);
40 // -----
41 // the analytical solution
42 // -----
43 proc anasol(
44     const in x: real
45     ): real {
46     return T0 + (TL - T0)*x/L;
47 }
```

---

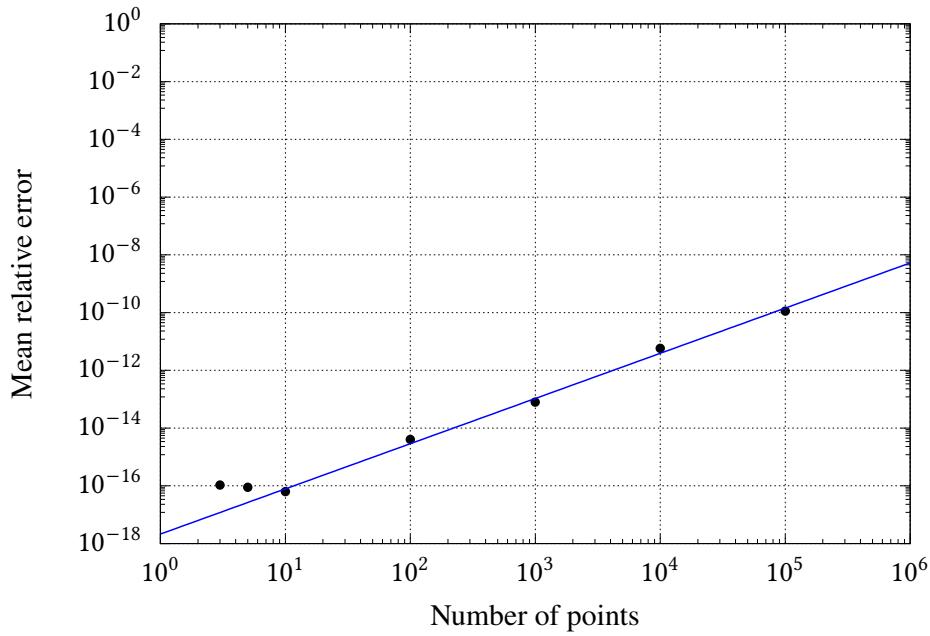


Figura 2.10: Erro da solução numérica de (2.17)–(2.19) em função do número de pontos  $N$  da grade.

$$y'(x) = k [A \operatorname{senh}(kx) + B \cosh(kx)].$$

Então,

$$\begin{aligned} y_0 &= A, \\ 0 &= k [y_0 \operatorname{senh}(kL) + B \cosh(kL)], \\ B \cosh(kL) &= -y_0 \operatorname{senh}(kL), \\ B &= -y_0 \operatorname{tgh}(kL). \end{aligned}$$

Consequentemente, a solução analítica é

$$y_a(x) = y_0 [\cosh(kx) - \operatorname{tgh}(kL) \operatorname{senh}(kx)]. \quad (2.29)$$

A discretização de (2.26) é a seguinte

$$\begin{aligned} \frac{y_{i-1} - 2y_i + y_{i+1}}{\Delta x^2} - k^2 y_i &= 0, \\ y_{i-1} - 2y_i - (k\Delta x)^2 y_i + y_{i+1} &= 0, \\ -y_{i-1} + (2 + \kappa^2) y_i - y_{i+1} &= 0, \end{aligned} \quad (2.30)$$

onde  $\kappa = k\Delta x$  é um número de onda adimensional de grade. As condições de contorno são como se segue. Para  $i = 1$ ,

$$(2 + \kappa^2)y_1 - y_2 = y_0. \quad (2.31)$$

Para  $i = N - 1$ , nós implementamos a condição de derivada nula via

$$\frac{dy(L)}{dx} = \frac{y_N - y_{N-1}}{\Delta x} + \mathcal{O}(\Delta x) = 0, \quad (2.32)$$

$$\begin{aligned} y_N &= y_{N-1}, \\ -y_{N-2}(2 + \kappa^2)y_{N-1} - y_N &= 0, \\ -y_{N-2}(2 + \kappa^2)y_{N-1} - y_{N-1} &= 0, \\ -y_{N-2} + (1 + \kappa^2)y_{N-1} &= 0. \end{aligned} \quad (2.33)$$

Listagem 2.15: edo-l2.chpl – Solução do problema de valor de contorno (2.26)–(2.28).

---

```

1 // -----
2 // edo-l2: solves d2y/dx2 - k^2 y = 0
3 // -----
4 config const N = 3;                      // start with 4 points (two interior)
5 config const k = 4;
6 config const L = 1.0;                     // and a length of 1.0
7 config const y0 = 1.0;                    // left boundary condition
8 const dx = L/N;                         // delta x
9 var x: [0..N] real = [i in 0..N] i*dx; // this is an expression forall
10 var y: [0..N] real = 0.0;               // the unknowns
11 var e: [0..N] real = 0.0;               // the errors
12 var a,b,c,d : [1..N-1] real;          // the tridiag matrix and forcing d
13 assert( N >= 3 );                   // don't bother with N < 3
14 const kappa = k*dx;                  // the grid dimensionless wavenumber
15 const B2 = (2.0 + kappa**2);          // the main diagonal
16 const B1 = (1.0 + kappa**2);          // the last diagonal
17 a = -1.0;
18 b[1..N-2] = B2;                     // sets the diagonal values
19 b[N-1] = B1;
20 c = -1.0;
21 d[1] = y0;                         // the left boundary condition
22 use tridiag;
23 tridiag(a,b,c,d,y[1..N-1]);
24 y[0] = 1.0;                         // BC enforced after tridiag
25 y[N] = y[N-1];                      // BC enforced after tridiag
26 use IO only openWriter;
27 use IO.FormattedIO;
28 var founam = "edo-l2-%06i.out".format(N);
29 const fou = openWriter(founam,locking=false);
30 for i in 0 .. N do {
31     var ya = anasol(x[i]);
32     e[i] = abs(y[i] - ya);
33     fou.writef("%8.4dr %8.4dr %8.4dr %12.4er\n",x[i],y[i],ya,e[i]);
34 }
35 fou.close();
36 const em = (+ reduce e[1..N-1])/(N-1);
37 writef("%06i %12.4er\n",N,em);
38 // -----
39 // the analytical solution
40 // -----
41 use Math only cosh, sinh, tanh;
42 proc anasol(
43     const in x: real
44 ): real {
45     return y0*(cosh(k*x) - tanh(k*L)*sinh(k*x));
46 }
```

---

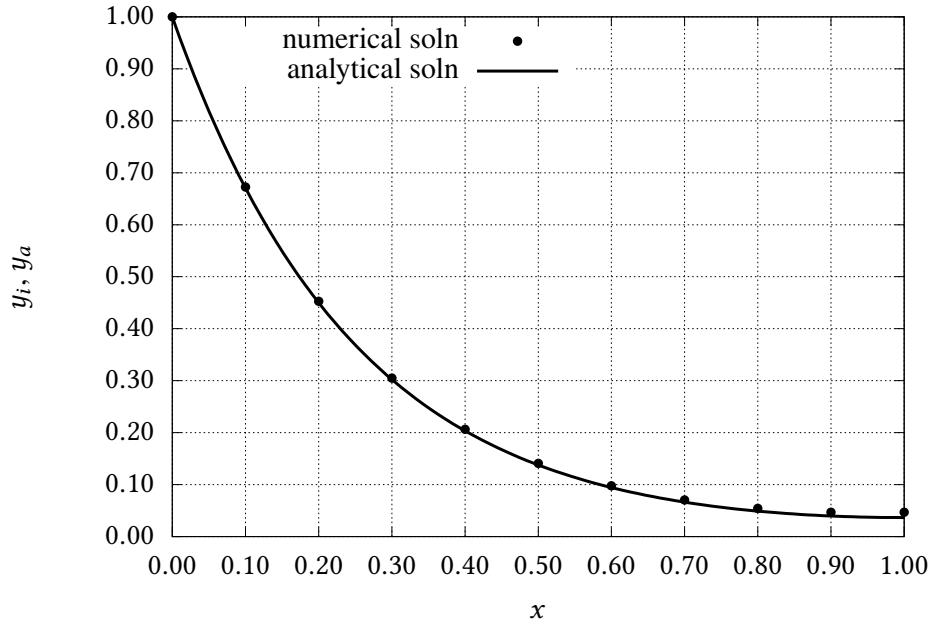


Figura 2.11: Solução numérica de (2.26)–(2.28) com uma discretização de ordem 1 para a condição de contorno direita.

Passamos agora ao programa que resolve o problema, `edo-l2.chpl`, mostrado na listagem 2.15. Para  $N = 10$ , a solução é mostrada na figura 2.11.

Agora nós olhamos para a *taxa de convergência* do método numérico para grades progressivamente refinadas. Isso é mostrado na figura 2.12. Observe que, ao contrário da figura 2.10, agora o erro da solução claramente cai com o aumento do número de pontos da grade, embora os erros seja muito maiores que antes. Isso significa que os erros do esquema numérico agora superam os erros de arredondamento da solução da matriz tridiagonal.

Interessantemente, o expoente da reta no gráfico log-log da figura 2.12 é  $-1.0744$ . Isso significa que o erro decresce *linearmente* com a diminuição de  $\Delta x$ , apesar do esquema centrado para a derivada segunda ser de  $\mathcal{O}(\Delta x^2)$ ! A culpada, neste caso, é a aproximação para a derivada da condição de contorno (2.32), que é de  $\mathcal{O}(\Delta x)$  apenas.

Vamos portanto tentar melhorar a acurácia do esquema. Fazemos isso colocando um *ponto-fantasma* em  $x_{N+1}$ . Em lugar de (2.28), temos agora a condição de contorno

$$\frac{dy(L)}{dx} = \frac{y_{N+1} - y_{N-1}}{2\Delta x} + \mathcal{O}(\Delta x^2) = 0, \quad (2.34)$$

$$\begin{aligned} y_{N+1} &= y_{N-1}, \\ -y_{N-1} + (2 + \kappa^2)y_N - y_{N+1} &= 0, \\ -2y_{N-1} + (2 + \kappa^2)y_N &= 0, \end{aligned} \quad (2.35)$$

Em resumo, agora temos  $N$  (e não mais  $N - 1$ ) incógnitas, e a última linha da matriz do sistema é (2.35). Passamos agora ao programa que resolve o problema, `edo-l2b.chpl`, mostrado na listagem 2.16. Para  $N = 10$ , a solução é mostrada na figura 2.13.

Agora nós olhamos para a *taxa de convergência* do método numérico para grades progressivamente refinadas. Isso é mostrado na figura 2.14. A inclinação da reta no gráfico log-log é  $-1.99077$  – na prática,  $-2$  – o que está de acordo com a ordem da aproximação da derivada no contorno direito. Note entretanto que o erro volta a subir para  $N = 100000$ : provavelmente, os erros de arredondamento da solução da matriz tridiagonal voltam, neste ponto, a ficar mais importantes do que a ordem de convergência do esquema numérico.

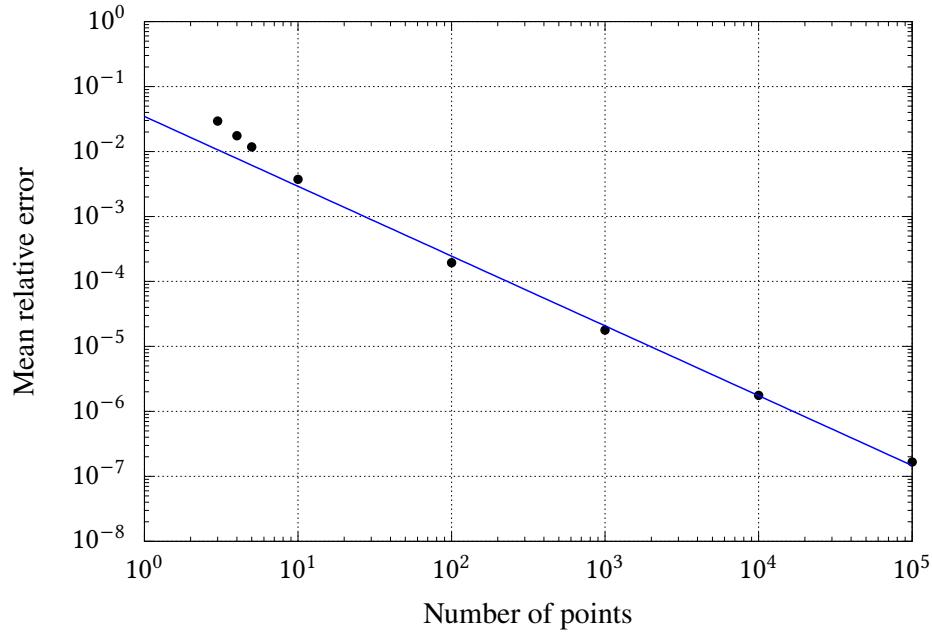


Figura 2.12: Erro da solução numérica de (2.26)–(2.28) em função do número de pontos  $N$  da grade. O coeficiente angular da reta é  $\sim -1$ .

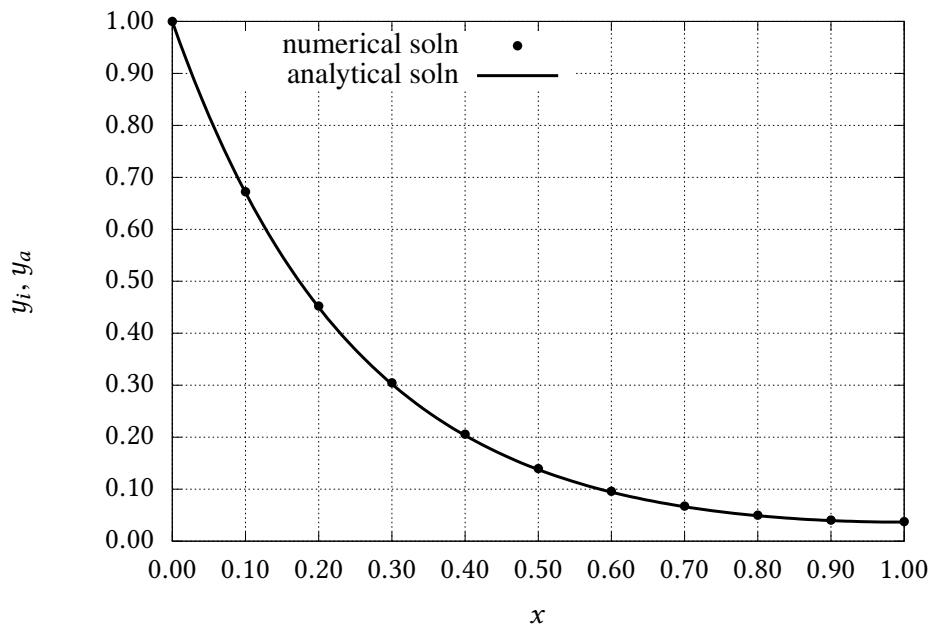


Figura 2.13: Solução numérica de (2.26)–(2.28) com uma discretização de ordem 2 para a condição de contorno direita.

Listagem 2.16: `edo-l2b.chpl` – Solução do problema de valor de contorno (2.26)–(2.28) com um esquema de ordem 2 para a condição de contorno direita.

---

```

1 // -----
2 // edo-l2b: solves  $d2y/dx^2 - k^2 y = 0$  with an  $O(dx^2)$  right boundary condition
3 //
4 config const N = 4;                                // start with 4 points (two interior)
5 config const k = 4;
6 config const L = 1.0;                               // and a length of 1.0
7 config const y0 = 1.0;                             // left boundary condition
8 const dx = L/N;                                  // delta x
9 var x: [0..N+1] real =
10    [i in 0..N+1] i*dx;                           // this is an expression forall
11 var y: [0..N+1] real = 0.0;                      // the unknowns
12 var e: [0..N+1] real = 0.0;                      // the errors
13 var a,b,c,d : [1..N] real;                      // the tridiag matrix and forcing d
14 assert( N >= 4 );                            // don't bother with N < 4
15 const kappa = k*dx;                            // the grid dimensionless wavenumber
16 const B2 = (2.0 + kappa**2);                   // the main diagonal
17 a[1..N-1] = -1.0;                            // the lower diagonal but for N
18 a[N] = -2;                                 // the right boundary condition
19 b[1..N] = B2;                                // sets the diagonal values
20 c[1..N] = -1.0;                            // the upper diagonal
21 d[1] = y0;                                 // the left boundary condition
22 use tridiag;
23 tridiag(a,b,c,d,y[1..N]);                  // the solution to the problem
24 y[0] = 1.0;                                // BC enforced after tridiag
25 y[N+1] = y[N-1];                           // BC enforced after tridiag
26 use IO only openWriter;
27 use IO.FormattedIO;
28 var founam = "edo-l2b-%06i.out".format(N);
29 const fou = openWriter(founam,locking=false);
30 for i in 0 .. N+1 do {
31   var ya = anasol(x[i]);
32   e[i] = abs(y[i] - ya);
33   fou.writef("%8.4dr %8.4dr %8.4dr %12.4er\n",x[i],y[i],ya,e[i]);
34 }
35 fou.close();
36 const em = (+ reduce e[1..N])/(N);
37 writef("%06i %12.4er\n",N,em);
38 //
39 // the analytical solution
40 //
41 use Math only cosh, sinh, tanh;
42 proc anasol(
43   const in x: real
44   ): real {
45   return y0*(cosh(k*x) - tanh(k*L)*sinh(k*x));
46 }
```

---

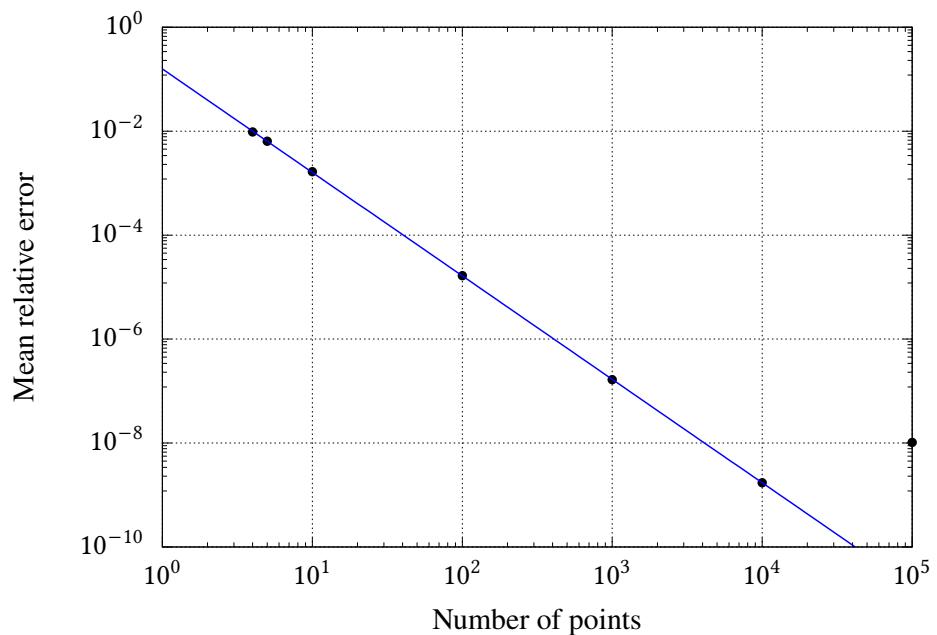


Figura 2.14: Erro da solução numérica de (2.26)–(2.28) em função do número de pontos  $N$  da grade. O coeficiente angular da reta é  $\sim -2$

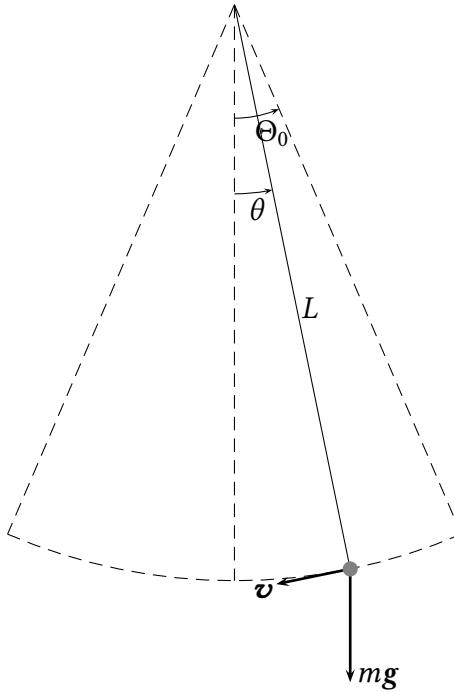


Figura 2.15: Um pêndulo (possivelmente) não-linear.

## 2.10 – Trabalhos computacionais

Esta seção contém diversas propostas de trabalhos computacionais. Eles são mais longos que os exercícios propostos, e requerem considerável dedicação e *tempo*. Os trabalhos desta seção mostram diversas aplicações de métodos numéricos, e lhe dão a oportunidade de ganhar uma prática considerável em programação. Não há, intencionalmente, solução destes trabalhos. Cabe a você, talvez juntamente com o seu professor, certificar-se de que os programas estão corretos. Vários dos trabalhos incluem soluções analíticas que podem ajudar nessa verificação.

### O método de Runge-Kutta e um pêndulo não-linear

A figura 2.15 mostra um pêndulo cujo cabo tem comprimento  $L$ , de massa  $m$ . O pêndulo sempre parte de uma posição angular inicial  $\theta = \Theta_0$ , com velocidade inicial nula. O comprimento de arco descrito pelo pêndulo a partir do ponto inicial; sua velocidade escalar; e sua aceleração escalar, são

$$\begin{aligned} s &= L(\Theta_0 - \theta), \\ v &= \frac{ds}{dt} = -L \frac{d\theta}{dt}, \\ a &= \frac{dv}{dt} = -L \frac{d^2\theta}{dt^2}. \end{aligned}$$

A 2ª lei de Newton nos dá

$$\begin{aligned} -mL \frac{d^2\theta}{dt^2} &= mg \sin(\theta), \\ \frac{d^2\theta}{dt^2} + \frac{g}{L} \sin(\theta) &= 0. \end{aligned} \tag{2.36}$$

A dimensão da equação (2.36) é

$$\frac{1}{T^2},$$

mas ela pode ser adimensionalizada via

$$\begin{aligned}\tau &= t \sqrt{\frac{g}{L}}, \\ \frac{d\theta}{dt} &= \frac{d\theta}{d\tau} \frac{d\tau}{dt} = \frac{d\theta}{d\tau} \sqrt{\frac{g}{L}}, \\ \frac{d^2\theta}{dt^2} &= \frac{d[d\theta/dt]}{d\tau} \frac{d\tau}{dt} = \frac{d^2\theta}{d\tau^2} \frac{g}{L}.\end{aligned}$$

Substituindo agora na equação (2.36), obtém-se

$$\frac{d^2\theta}{d\tau^2} + \sin(\theta) = 0, \quad \theta(0) = \Theta_0, \quad \theta'(0) = 0 \quad (2.37)$$

(note que nós agora incluímos as condições inciais).

Essa equação diferencial não pode ser resolvida por métodos analíticos em termos apenas em funções transcendentes elementares (funções baseadas nas funções trigonométricas e na função exponencial (incluindo as inversas)). Se a posição angular inicial  $\Theta_0$  for “pequena”, podemos aproximar o seno por uma série de Taylor apenas até o primeiro termo,  $\sin \theta \approx \theta$ , e transformar a equação diferencial em

$$\begin{aligned}\frac{d^2\theta}{d\tau^2} + \theta &= 0, \quad \theta(0) = \Theta_0, \quad \theta'(0) = 0; \\ \theta(\tau) &= A \cos(\tau) + B \sin(\tau), \\ A &= \Theta_0, \quad B = 0 \Rightarrow \\ \theta &= \Theta_0 \cos(\tau) = \Theta_0 \cos\left(\sqrt{\frac{g}{L}}\tau\right).\end{aligned}$$

Talvez a característica mais interessante dessa solução seja que o período de oscilação *não depende da amplitude*  $\Theta_0$ : de acordo com a solução analítica (aproximada) acima, ele é obtido da seguinte forma:

$$\begin{aligned}\cos\left(\frac{2\pi t}{T}\right) &= \cos\left(\sqrt{\frac{g}{L}}t\right), \\ \frac{2\pi t}{T} &= \sqrt{\frac{g}{L}}t, \\ T &= 2\pi \sqrt{\frac{L}{g}}.\end{aligned}$$

As coisas ficam ainda mais simples nas variáveis adimensionais: o período em unidades de  $\tau$  é, simplesmente,

$$T_\tau = 2\pi.$$

Para o pêndulo não linear, por outro lado, é razoável prever, com base nas ferramentas de análise dimensional discutidas no início do curso, que o período de oscilação tem a forma

$$T = f(\Theta_0) \sqrt{\frac{g}{L}},$$

onde  $f(0) = 2\pi$ . O objetivo deste trabalho é a obtenção “experimental” de  $f(\Theta_0)$ .

O plano agora é resolver a equação correta numericamente para um grande número de condições inciais  $\Theta_0$ , e plotar o período de cada uma dessas soluções contra  $\Theta_0$ .

Para resolver o problema não-linear, escrevemos, a partir de (2.37):

$$\frac{d\theta}{d\tau} = \omega, \quad \theta(0) = \Theta_0, \quad (2.38)$$

$$\frac{d\omega}{d\tau} = -\operatorname{sen}(\theta), \quad \omega(0) = 0, \quad (2.39)$$

e resolvemos o sistema (2.38)–(2.39) com o método de Runge-Kutta de 4<sup>a</sup> ordem para um grande número de condições iniciais  $\Theta_0$ , a saber:  $\Theta_0 = 0.05, \Theta_0 = 0.1, \dots, \Theta_0 = 3$ . Para cada um desses valores, nós determinamos o período da solução. Em termos da variável adimensional independente  $\tau$ , o período é justamente o valor de  $f(\Theta_0)$ .

O cálculo do período  $f(\Theta_0)$  exige um algoritmo próprio, *além da solução numérica por Runge-Kutta de (2.38)–(2.39)*. Uma idéia simples e que funciona é a seguinte:

1. Adote um passo pequeno para a solução numérica de (2.38)–(2.39). Por exemplo,  $\Delta\tau = 0.001$ , e simule até  $\tau = 50$  para obter um certo número de períodos.
2. *Para cada  $\Theta_0$ , resolva numericamente o problema*, gerando uma lista de valores  $\theta_0 = \Theta_0, \theta_1, \dots, \theta_N$ , que são a solução numérica do problema.
3. Percorra a lista de  $\theta$ s para  $i = 1, \dots, N$ : toda vez que  $p = -\theta_{i-1}/\theta_i > 0$ , a função trocou de sinal. Obtenha o zero da função  $\theta(t)$  por interpolação linear:

$$\tau_k = \tau_{i-1} + \frac{p}{1+p} \Delta\tau.$$

Adicione  $\tau_k$  a uma lista separada com os zeros de  $\theta(t)$ .

4. Calcule as diferenças entre esses zeros:

$$\delta_k = \tau_k - \tau_{k-1}.$$

5. Obtenha a média aritmética  $\bar{\delta}_k$  dos  $\delta_k$ s. O período será

$$f(\Theta_0) = 2\bar{\delta}_k$$

6. Guarde esse  $f(\Theta_0)$ , e prossiga para o próximo  $\Theta_0$  em 2.

Você deve plotar os seus resultados para conferir. A função  $f(\Theta_0)$  é mostrada na figura 2.16

## Ressonância e um pêndulo não-linear

A figura 2.15 mostra um pêndulo de massa  $m$  forçado por uma força  $F$  que é sempre tangente à trajetória circular, cujo cabo tem comprimento  $L$ . O pêndulo sempre parte de uma posição angular inicial  $\theta = \Theta_0$ , com velocidade inicial nula. O comprimento de arco descrito pelo pêndulo a partir do ponto inicial; sua velocidade escalar; e sua aceleração escalar, são

$$\begin{aligned} s &= L(\Theta_0 - \theta), \\ v &= \frac{ds}{dt} = -L \frac{d\theta}{dt}, \\ a &= \frac{dv}{dt} = -L \frac{d^2\theta}{dt^2}. \end{aligned}$$

Faça  $F(t) \equiv -mg\phi(t)$  o valor da componente tangencial da força (com sinal). A 2<sup>a</sup> lei de Newton nos dá

$$\begin{aligned} -mL \frac{d^2\theta}{dt^2} &= mg \operatorname{sen} \theta + F(t), \\ \frac{d^2\theta}{dt^2} + \frac{g}{L} \operatorname{sen} \theta &= -\frac{F(t)}{mL}, \\ \frac{d^2\theta}{dt^2} + \frac{g}{L} \operatorname{sen} \theta &= \frac{mg}{mL} \phi(t). \end{aligned} \quad (2.40)$$

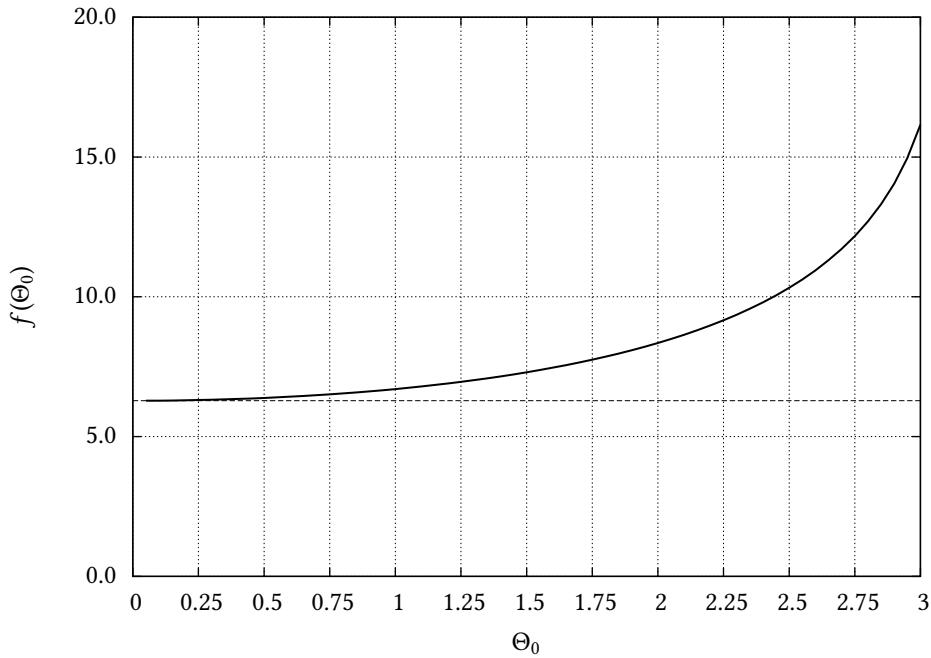


Figura 2.16: O período de um pêndulo não linear em função da amplitude inicial  $\Theta_0$ . A linha tracejada é o valor teórico  $f(0) = 2\pi$  para oscilações de pequena amplitude.

A dimensão da equação (2.40) é

$$\frac{1}{T^2},$$

mas ela pode ser adimensionalizada via

$$\begin{aligned} \tau &= t \sqrt{\frac{g}{L}}, \\ \frac{d\theta}{dt} &= \frac{d\theta}{d\tau} \frac{d\tau}{dt} = \frac{d\theta}{d\tau} \sqrt{\frac{g}{L}}, \\ \frac{d^2\theta}{dt^2} &= \frac{d}{d\tau} \left[ \frac{d\theta}{dt} \right] \frac{d\tau}{dt} = \frac{d^2\theta}{d\tau^2} \frac{g}{L}. \end{aligned}$$

Substituindo agora na equação (2.40), obtém-se

$$\frac{d^2\theta}{d\tau^2} + \sin \theta = \phi(\tau), \quad \theta(0) = \Theta_0, \quad \theta'(0) = 0 \quad (2.41)$$

(note que nós agora incluímos as condições inciais).

Essa equação diferencial não pode ser resolvida por métodos analíticos em termos apenas de funções algébricas e funções transcendentais elementares (funções baseadas nas funções trigonométricas e na função exponencial (incluindo as inversas)). Se a posição angular inicial  $\Theta_0$  for “pequena”, podemos aproximar o seno por uma série de Taylor apenas até o primeiro termo,  $\sin \theta \approx \theta$ , e transformar a equação diferencial em

$$\frac{d^2\theta}{d\tau^2} + \theta = \phi(\tau), \quad \theta(0) = \Theta_0, \quad \theta'(0) = 0. \quad (2.42)$$

Se

$$\phi(\tau) = \sin(\tau), \quad (2.43)$$

aparecerá ressonância em (2.42). Nossa questão é: aparecerá também ressonância em (2.41)?

Para resolver o problema não-linear, escrevemos, a partir de (2.41):

$$\frac{d\theta_n}{d\tau} = \omega_n, \quad \theta_n(0) = \Theta_0, \quad (2.44)$$

$$\frac{d\omega_n}{d\tau} = -\sin \theta_n + \sin(\tau), \quad \omega_n(0) = 0. \quad (2.45)$$

O problema linear tem solução numérica muito parecida:

$$\frac{d\theta_l}{d\tau} = \omega_l, \quad \theta_l(0) = \Theta_0, \quad (2.46)$$

$$\frac{d\omega_l}{d\tau} = -\theta_l + \sin(\tau), \quad \omega_l(0) = 0, \quad (2.47)$$

Em (2.44)–(2.47) nós adicionamos os subscritos  $n$  e  $l$  para distinguir a solução *linear* da solução *não-linear*.

Seu objetivo é comparar as duas soluções: resolva os sistemas (2.44)–(2.45) e (2.46)–(2.47) utilizando o método de Runge-Kutta de 4ª ordem com  $\Theta_0 = 1,5$ . Você deve utilizar um passo  $\Delta\tau = 0,01$  em  $\tau$ , e marchar de  $\tau = 0$  até  $\tau = 1000$ .

Plote no mesmo gráfico  $\tau \times \theta_l(\tau)$  e  $\tau \times \theta_n(\tau)$ , e mostre graficamente que o sistema linear exibe ressonância. O que você pode dizer sobre o sistema não-linear?

Além disso, considere, e responda tão bem quanto possível, as seguintes questões:

- O que acontece com a solução linear quando você reduz  $\Delta\tau$  para 0,001? E para 0,0001?
- O que acontece com a solução não-linear quando você reduz  $\Delta\tau$  para 0,001? E para 0,0001?
- O que está acontecendo? É possível tirar alguma conclusão sobre a solução do sistema não-linear?

Atenção: os valores de  $\theta(\tau)$  crescem muito além de  $2\pi$  (uma volta completa), e portanto é difícil interpretar fisicamente a solução numérica após valores de  $\tau$  da ordem de  $2\pi$ . Não se preocupe com isso, concentrando-se nas propriedades matemáticas da solução.

## A distribuição de Rayleigh

A distribuição de Rayleigh é uma *distribuição de probabilidade*. A função densidade de probabilidade é

$$f_R(x) = \frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}, \quad x \geq 0. \quad (2.48)$$

A função distribuição acumulada é

$$F_R(x) = \int_0^x f_R(t) dt = 1 - e^{-\frac{x^2}{2\sigma^2}}. \quad (2.49)$$

É fácil ver que, como acontece com toda função densidade de probabilidade, a sua integral é igual a um:

$$\int_0^\infty f_R(x) dx = F_R(\infty) = 1. \quad (2.50)$$

Neste trabalho, é proibido usar  $F_R(x)$ : você pode usá-la “por fora” para testar seus resultados, mas ela não pode aparecer no seu programa. Todo o trabalho envolve apenas a manipulação de  $f_R(x)$  em (2.48), ou de sua série de Taylor.

## Parte I

Obtenha a série de Taylor, em torno de  $x = 0$ , de  $f_R(x)$  para  $\sigma = 1$ :

$$f_S(x) = c_1x + c_2x^2 + c_3x^3 + c_4x^4 + \dots \quad (2.51)$$

(isto é, obtenha os  $c_n$ s em função de  $n$ , analiticamente). Escreva uma função que calcule a série com uma acurácia  $\epsilon = 10^{-6}$ . O programa deve gerar um arquivo de saída com 3 colunas:  $x$ ,  $f_S(x)$ , e  $f_R(x)$ . Naturalmente, você deve encontrar  $f_S(x) \approx f_R(x)$ . No entanto, para  $x$  “grande” (digamos,  $x > \sqrt{10}$ ), começam a aparecer problemas numéricos em  $f_S(x)$ .

Plote  $f_S(x)$  e  $f_R(x)$ , e mostre os problemas que aparecem em  $f_S(x)$  para  $x$  grande.

Uma estratégia para lidar com o destrambelhamento de  $f_S(x)$  para  $x$  grande é a seguinte: se  $x > \sqrt{10}$ , calcule:

$$\begin{aligned} y &= x/\sqrt{10}, \\ \eta &= \lfloor x/\sqrt{10} \rfloor, \\ \lambda &= \sqrt{10}\eta, \\ z &= x/\lambda, \\ N &= \lambda^2 = 10\eta^2. \end{aligned}$$

$\eta$  é o maior inteiro menor ou igual a  $y$ . Em Python, isso é calculado com `eta = floor(y)`. Repare que  $N$  é, convenientemente, um número inteiro por definição.

Agora,

$$\begin{aligned} f_R(x) &= xe^{-x^2/2} \\ &= \lambda \left[ \frac{x}{\lambda} \right] \exp \left[ -\lambda^2 \frac{1}{2} \left( \frac{x}{\lambda} \right)^2 \right] \\ &= \lambda z \left[ \exp(-z^2/2) \right]^{\lambda^2} \\ &= \lambda z \left[ \exp(-z^2/2) \right]^N. \end{aligned}$$

A série de Taylor de  $\exp(-z^2/2)$  está intimamente relacionada com  $f_S(z)$ :

$$\exp(-z^2/2) = f_T(z) = f_S(z)/z = c_1 + c_2z + c_3z^2 + c_4z^3 + \dots,$$

onde os  $c_n$ s são os mesmos de (2.51). Portanto, quando  $x > \sqrt{10}$ , calcule  $y$ ,  $\eta$ ,  $\lambda$ ,  $z$  e  $N$ , calcule  $f_T(z)$ , eleve a  $N$ , e multiplique por  $\lambda z$ .

## Parte II

Verifique numericamente a validade de (2.50), usando um esquema de integração numérica de sua escolha. O problema é lidar com o limite superior da integral, que é  $\infty$ . Como você pode fazer para garantir que sua integral numérica é uma boa aproximação de (2.50)? Afinal, é impossível colocar  $\infty$  em um programa de computador...

Dica: o capítulo sobre integração numérica de ? tem várias sugestões para lidar com esse problema. Você pode usar as sugestões de ?, ou usar uma idéia sua.

Atenção! A sua solução também deve fazer parte do programa de computador produzido para o trabalho, e o resultado numérico de (2.50) deve ser impresso na tela, com 10 casas decimais.

## Obtenção de curvas de remanso pelo método de Runge-Kutta

Este texto é uma adaptação de um trabalho publicado pelo autor em Congresso Científico (?).

**Importância dos cálculos de curvas de remanso** Um problema clássico em hidráulica de canais é a obtenção de curvas de remanso (??). O problema e suas soluções têm grande aplicação na medida em que suas hipóteses – regime permanente e escoamento gradualmente variado – ocorrem frequentemente em canais naturais e artificiais. Algumas aplicações importantes são a determinação de cotas da superfície da água a montante de um reservatório, e o cálculo de superfícies-chave em postos fluviométricos sujeitos à influência de remanso.

As soluções clássicas de curvas de remanso não enfatizam que se trata em última análise da solução de uma equação diferencial não-linear. Neste trabalho, adaptado de ?, formularemos o problema em função do nível d'água  $Z$  e da distância  $x$  desde a primeira seção de jusante, obteremos a equação diferencial correspondente  $dZ/dx = f(x, Z)$  e a resolveremos usando uma ferramenta padrão, que é o método de Runge-Kutta.

Cabe notar que a proposta de calcular curvas de remanso com o método de Runge-Kutta, é bastante antiga. Ver, por exemplo, ?. Coincidemente, o exemplo utilizado por ? é o mesmo apresentado em ?, e discutido aqui!!

**Obtenção da equação diferencial do problema** A figura 2.17 mostra as características gerais de um canal. A equação dinâmica de escoamento em canais em regime permanente é

$$\frac{d}{dx} \frac{Q^2}{A} + gA \frac{dZ}{dx} + gAS_f = 0, \quad (2.52)$$

onde  $Q$  é a vazão,  $A$  é a área molhada,  $g$  é a aceleração da gravidade,  $Z$  é a cota da superfície da água e  $S_f$  é a perda de carga. Outros elementos geométricos são a cota do fundo  $Z_f$ , a profundidade  $h$ , o raio hidráulico  $R$  e o perímetro molhado  $P$ . A declividade do fundo é  $S_o = -dZ_f/dx$ . Admitindo-se que a vazão é constante em  $x$ , obtém-se

$$-\frac{Q^2}{A^2} \frac{dA}{dx} + gA \frac{dZ}{dx} + gAS_f = 0, \quad (2.53)$$

A derivada  $dA/dx$  em (2.53) é uma derivada total. A área molhada por outro lado é função de  $x$  e  $Z$ . Então, sendo  $B$  a largura superficial,

$$\frac{dA}{dx} = \frac{\partial A}{\partial x} \Big|_z + \frac{\partial A}{\partial Z} \Big|_x \frac{dZ}{dx} = \frac{\partial A}{\partial x} \Big|_z + B \frac{dZ}{dx} \quad (2.54)$$

Portanto,

$$\frac{dZ}{dx} = \frac{\frac{Q^2}{A^2} \frac{\partial A}{\partial x} \Big|_z - gAS_f}{gA - \frac{Q^2}{A^2} B} = f(x, Z) \quad (2.55)$$

é a equação diferencial não-linear que precisa ser integrada para a obtenção de curvas de remanso. Isto pode ser feito de maneira eficiente pelo método de Runge-Kutta de 4<sup>a</sup> ordem (?). Métodos mais tradicionais em hidráulica envolvem a solução numérica (por diferenças finitas) para  $x$  como variável independente da equação diferencial

$$\frac{dx}{dh} = \frac{1 - F^2}{S_0 - S_f} \quad (2.56)$$

onde  $h$  é a profundidade média do escoamento e  $F$  é o número de Froude (?), p. 202), ou a solução iterativa da equação de energia entre duas seções consecutivas em canais não-prismáticos, como por exemplo o *step method* descrito por ?, p. 218–222.

As principais vantagens de utilizar a equação (2.55) em conjunto com o método de Runge-Kutta são:

1. A equação diferencial está explicitada para  $dZ/dx$ , e não  $dx/dZ$ . A solução dá de forma direta a cota  $Z(x)$  para cada seção cuja abscissa é  $x$ .
2. O método de Runge-Kutta não é iterativo. Sua programação é trivial.

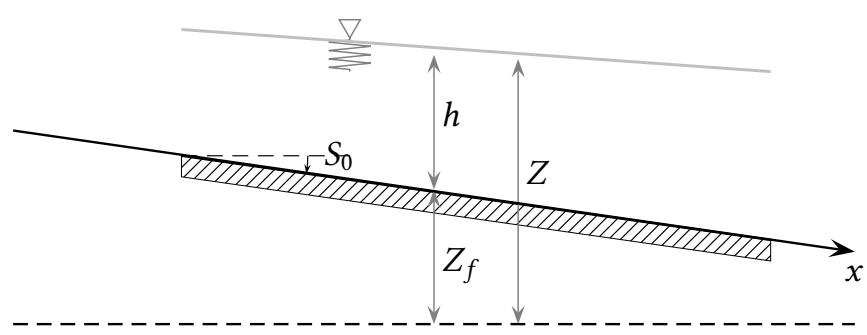
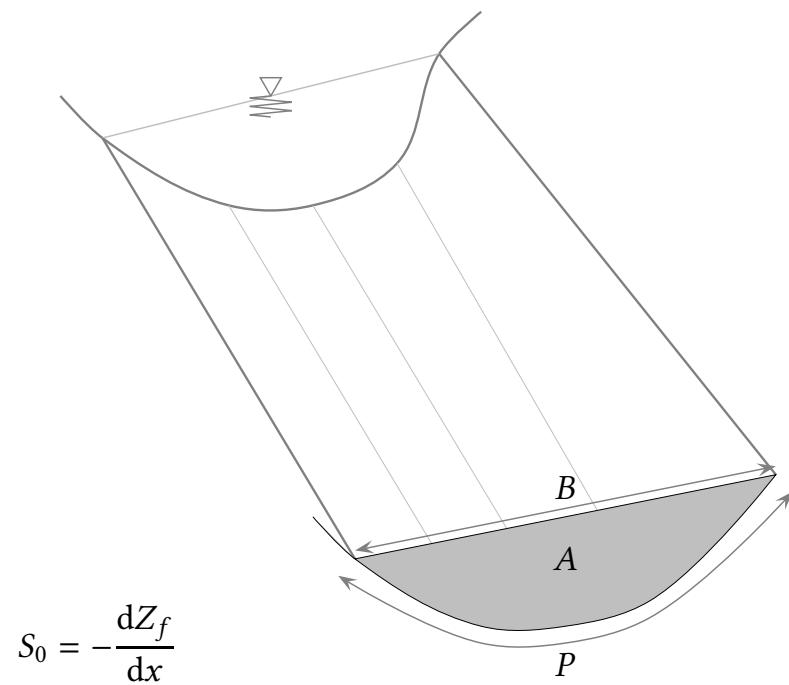


Figura 2.17: Características geométricas de um canal.

Tabela 2.1: Curva de remanso em canal trapezoidal – solução de Ven Te Chow

seção	$x$	$A$	$B$	$R$	$S_f$	$V$	$Z$
	m	$\text{m}^2$	m	m	–	$\text{m s}^{-1}$	m
00	0.00	13.94	12.19	1.08	3.73e-4	0.81	1.52
01	-47.24	13.20	11.95	1.05	4.31e-4	0.86	1.54
02	-96.93	12.48	11.70	1.01	5.08e-4	0.91	1.56
03	-150.27	11.77	11.46	0.97	6.03e-4	0.96	1.58
04	-208.48	11.08	11.22	0.94	7.10e-4	1.02	1.61
05	-273.71	10.41	10.97	0.90	8.52e-4	1.09	1.66
06	-352.04	9.74	10.73	0.87	1.02e-3	1.16	1.72
07	-400.51	9.42	10.61	0.84	1.14e-3	1.20	1.77
08	-461.77	9.10	10.49	0.83	1.24e-3	1.25	1.84
09	-500.18	8.94	10.42	0.82	1.31e-3	1.27	1.88
10	-547.73	8.78	10.36	0.81	1.38e-3	1.29	1.94
11	-584.30	8.68	10.33	0.80	1.43e-3	1.31	1.99
12	-632.46	8.59	10.29	0.80	1.46e-3	1.32	2.06
13	-674.83	8.53	10.27	0.79	1.51e-3	1.33	2.12
14	-731.82	8.47	10.24	0.78	1.53e-3	1.34	2.21

3. Fica conceitualmente simples definir pontos  $x$  onde se deseja calcular parâmetros hidráulicos que não coincidem com nenhuma seção tabulada. Neste caso, usa-se interpolação linear para a obtenção de  $A(x, Z)$  e  $B(x, Z)$  entre as seções imediatamente a jusante e a montante do ponto  $x$ .

A derivada  $\frac{\partial A}{\partial x}|_Z$  é calculada por um esquema simples de diferenças finitas:

$$\left. \frac{\partial A}{\partial x} \right|_Z \approx \frac{A(x_j, Z) - A(x_m, Z)}{x_j - x_m}, \quad (2.57)$$

onde  $x_j$  e  $x_m$  são as abscissas de seções imediatamente a jusante e a montante do ponto  $x$ .

**Trabalho** Você deve fazer uma comparação do método de cálculo apresentado acima com um resultado clássico. Escolhemos um exemplo do livro de ?. A tabela 2.1, adaptada do exemplo 10.1 de Chow mostra o resultado o cálculo realizado por Chow, *com um método diferente*, de uma curva de remanso em um canal de seção trapezoidal, largura da base de 6,10 m e inclinação dos taludes de 1:2 (*i.e.*, dois passos na horizontal para um na vertical), com coeficiente de Manning  $n = 0,025$ , declividade  $S_0 = 0,0016$  e uma vazão constante de  $11,33 \text{ m}^3 \text{ s}^{-1}$ . Todas as unidades foram convertidas para o sistema internacional (SI), e alguns parâmetros que não constam do exemplo de ?, mas podem ser calculados, tais como a cota da linha de água, foram adicionados.

O seu trabalho deve calcular a curva de remanso pelo método de Runge-Kutta, e comparar os resultados com a tabela 2.1. Suas condições inciais são os valores da seção 00 na tabela

Os cálculos devem ser realizados de *jusante para montante*. Como a equação (2.52) pressupõe que a velocidade é positiva no sentido positivo dos  $x$ s você deve arbitrar  $x = 0$  na seção de montante e marchar no sentido negativo, com  $\Delta x = -10 \text{ m}$ , até  $x = -1000 \text{ m}$ . Para uma seção trapezoidal com a geometria dada, os parâmetros hidráulicos (geométricos e dinâmicos) são calculados na seguinte ordem:

$$Z_f = -S_0 x, \quad (2.58)$$

$$h = Z - Z_f, \quad (2.59)$$

$$B = b_f + 4h, \quad (2.60)$$

$$P = b_f + 2 * \sqrt{5}h, \quad (2.61)$$

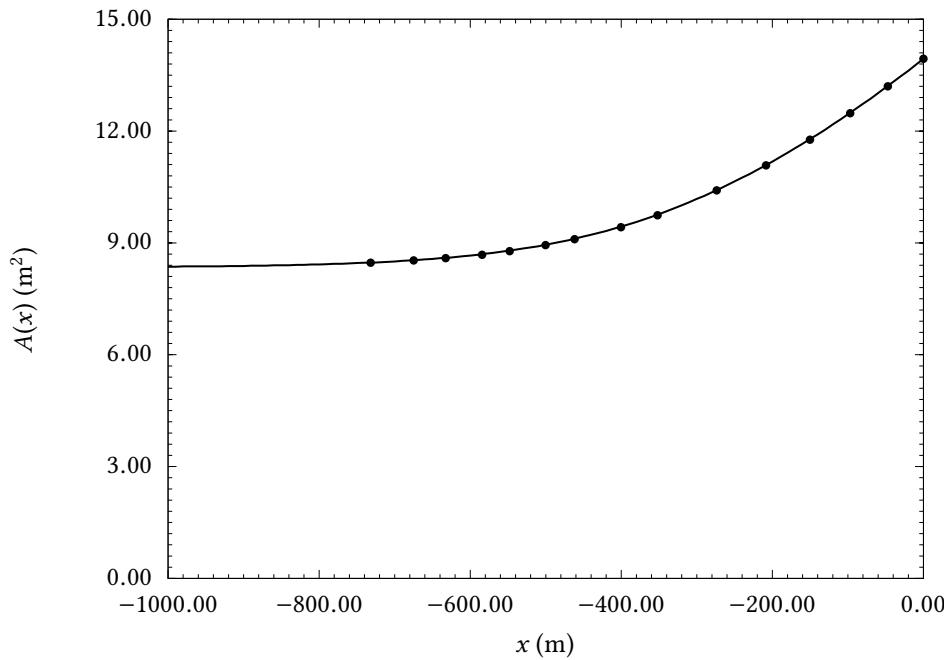


Figura 2.18: Área molhada em função da distância de jusante do exemplo 10.1 de ?, calculados por Chow (círculos) versus resultados do método de Runge-Kutta (linha contínua).

$$A = (B + b_f)h/2, \quad (2.62)$$

$$R = A/P, \quad (2.63)$$

$$V = Q/A, \quad (2.64)$$

$$S_f = \frac{n^2 V^2}{R^{4/3}} \quad (2.65)$$

Você deve:

1. Gerar um arquivo de saída `rkrem.out` contendo, em cada linha,  $x$ ,  $A$ ,  $B$ ,  $R$ ,  $V$ ,  $S_f$  e  $Z$  no formato

```
'%6.2f %5.2f %5.2f %5.2f %8.2e %5.2f %4.2f %4.2f\n'.
```

2. Interpolar os valores de  $x$  da tabela 2.1 e produzir uma tabela similar com os *seus* resultados.
3. Gerar figuras semelhantes às figuras 2.18–2.21 contendo as curvas calculadas  $A(x)$ ,  $S_f(x)$ ,  $V(x)$  e  $Z(x)$  e os pontos correspondentes da tabela 2.1 para comparação.

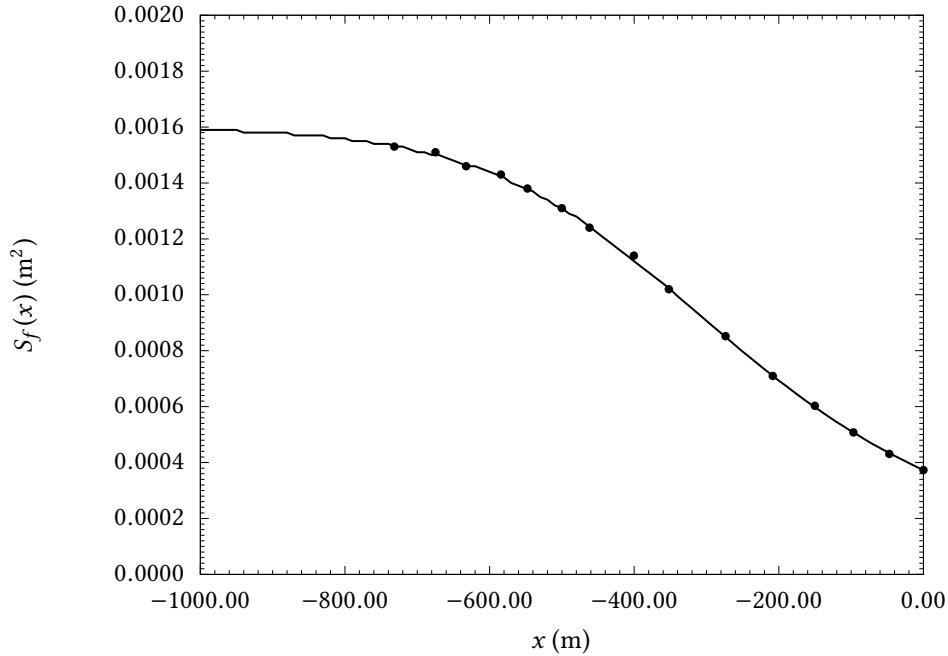


Figura 2.19: Perda de carga em função da distância de jusante do exemplo 10.1 de ?, calculados por Chow (círculos) versus resultados do método de Runge-Kutta (linha contínua).

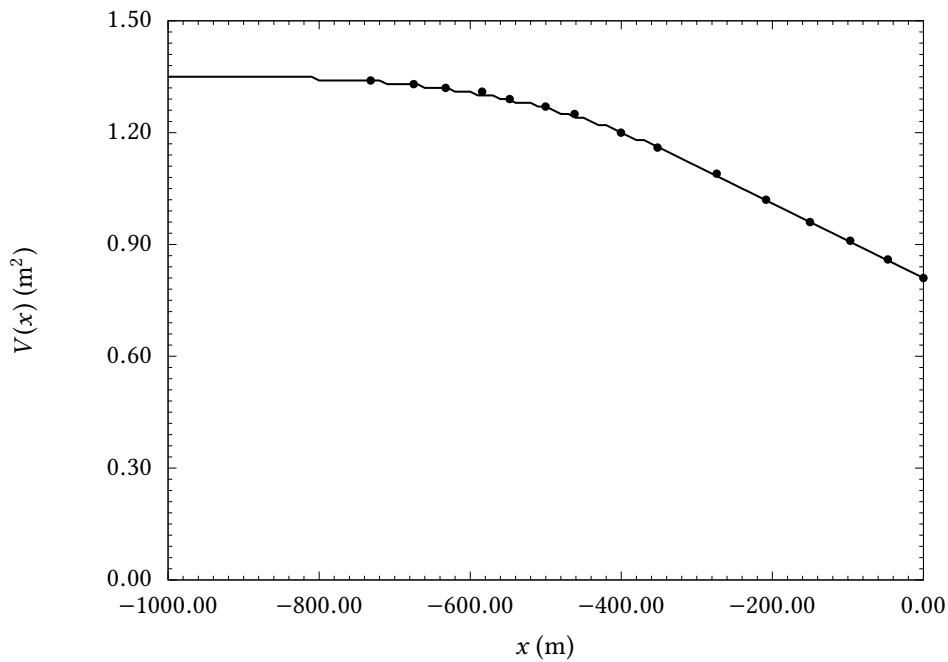


Figura 2.20: Velocidade em função da distância de jusante do exemplo 10.1 de ?, calculados por Chow (círculos) versus resultados do método de Runge-Kutta (linha contínua).

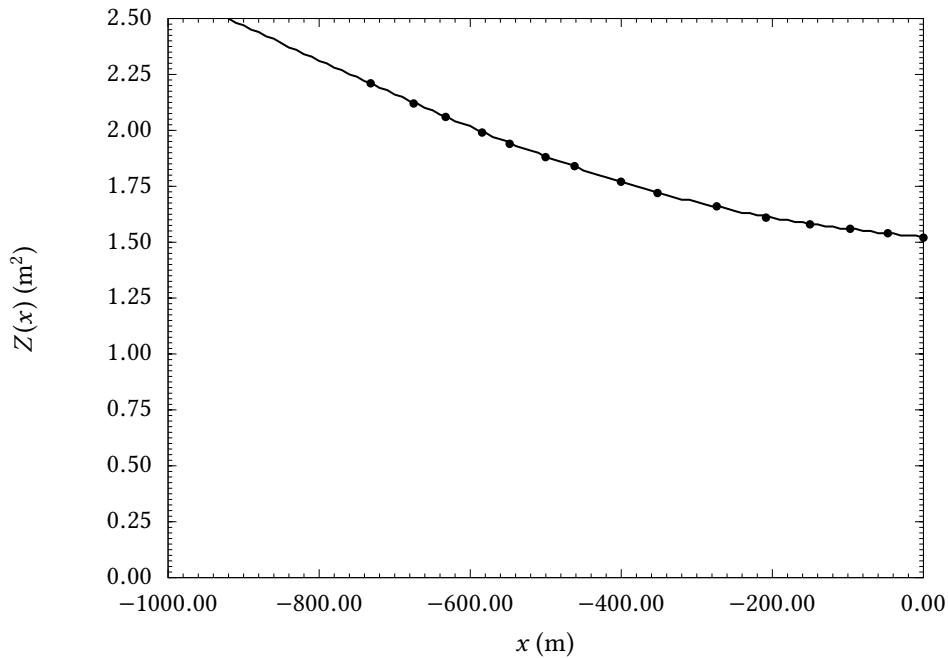


Figura 2.21: Cota em função da distância de jusante do exemplo 10.1 de ?, calculados por Chow (círculos) versus resultados do método de Runge-Kutta (linha contínua).



Figura 2.22: A traça (*Spruce budworm*) *Choristoneura orae*. Fonte: Wikipedia. ©entomart (<http://www.entomart.be>).

Tabela 2.2: Parâmetros do modelo de ?

Parâmetro	Unidade	Valor
$r_B$	ano <sup>-1</sup>	1.52
$K'$	larvas galho <sup>-1</sup>	355
$\beta$	larvas acre <sup>-1</sup> ano <sup>-1</sup>	43200
$\alpha'$	larvas galho <sup>-1</sup>	1.11
$r_S$	ano <sup>-1</sup>	0.095
$K_S$	galhos acre <sup>-1</sup>	25440
$K_E$	—	1
$r_E$	ano <sup>-1</sup>	0.92
$P'$	larva <sup>-1</sup>	0.00195
$T$	—	0.05

### Um modelo ecológico para *Choristoneura orae*

A *Choristoneura orae* (figura 2.22) é uma traça conhecida nos EUA como *Spruce Budworm*. ? propuseram um modelo matemático simples de crescimento da traça: se  $B$  é a sua densidade populacional, o modelo é

$$\frac{dB}{dt} = r_B B \left( 1 - \left( \frac{B}{K'S} \right) \frac{T^2 + E^2}{E^2} \right) - \frac{\beta}{K_S} \left( \frac{B^2}{(\alpha'S)^2 + B^2} \right), \quad (2.66)$$

$$\frac{dS}{dt} = r_S S \left( 1 - \frac{SK_E}{E} \right), \quad (2.67)$$

$$\frac{dE}{dt} = r_E E \left( 1 - \frac{E}{K_E} \right) - \left( \frac{P'B}{S} \right) \frac{E^2}{T^2 + E^2}. \quad (2.68)$$

Todos os parâmetros são definidos em (?). As variáveis que evoluem são  $B$  (a densidade de larvas, em larvas galho<sup>-1</sup>;  $S$  (a área relativa de galhos) e  $E$  (a energia por galho).

Aqui, nós nos limitamos a listá-los (em suas unidades originais) na tabela 2.2

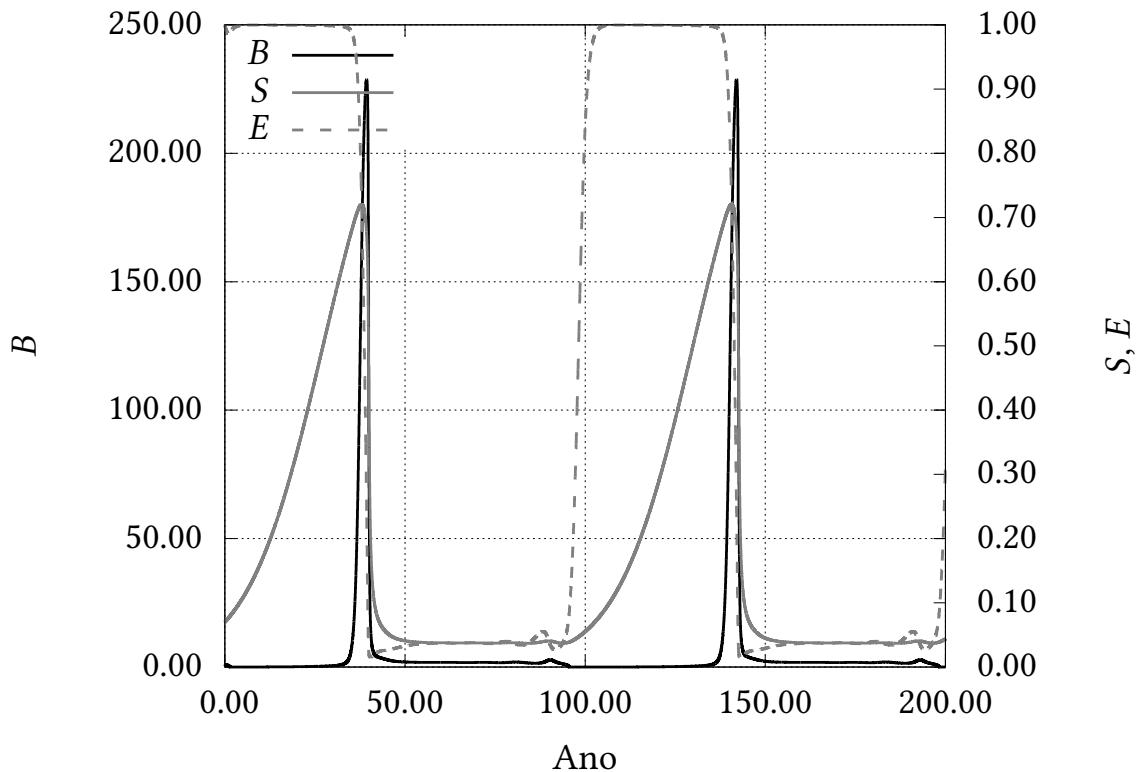


Figura 2.23: Simulação de eclosão de uma praga de traças, utilizando o modelo de ?.

Utilize as condições iniciais para o sistema,  $B(0) = 1$ ,  $S(0) = 0.070$  e  $E(0) = 1$ . Resolva o sistema de equações (2.66)–(2.68), utilizando um passo de tempo de  $1/400$  ano e um tempo total de simulação de 200 anos. O seu resultado deve ser o mostrado na figura 2.23. Compare com a figura 5 de ?.

### O oscilador de van der Pol

Resolva o sistema de equações diferenciais

$$\begin{aligned}\frac{dx}{dt} &= \mu \left( x - \frac{1}{3}x^3 - y \right), \\ \frac{dy}{dt} &= \frac{1}{\mu}x,\end{aligned}$$

com  $x(0) = 0.01$ ,  $y(0) = 0.01$ , e  $\mu = 5$ . Plote o resultado até  $t = 50$ .

O seu resultado deve ser o mostrado na figura 2.24

Agora, desenvolva uma método numérico para obter a amplitude e o período de oscilação de  $x(t)$ . Note que existe um transiente, até que o sistema entre em um regime estacionário (não confunda estacionário com  $x$  constante!). Você deve desprezar esse transiente antes de obter a amplitude e o período.

Finalmente, para as mesmas condições iniciais dadas acima, faça um grande número de simulações variando  $\mu$  de 0,1 em 0,1, desde 0,1 até 10. Plote agora o período  $T$  e a amplitude  $A$  contra  $\mu$ . Você deve comparar seus resultados com os obtidos por ?.

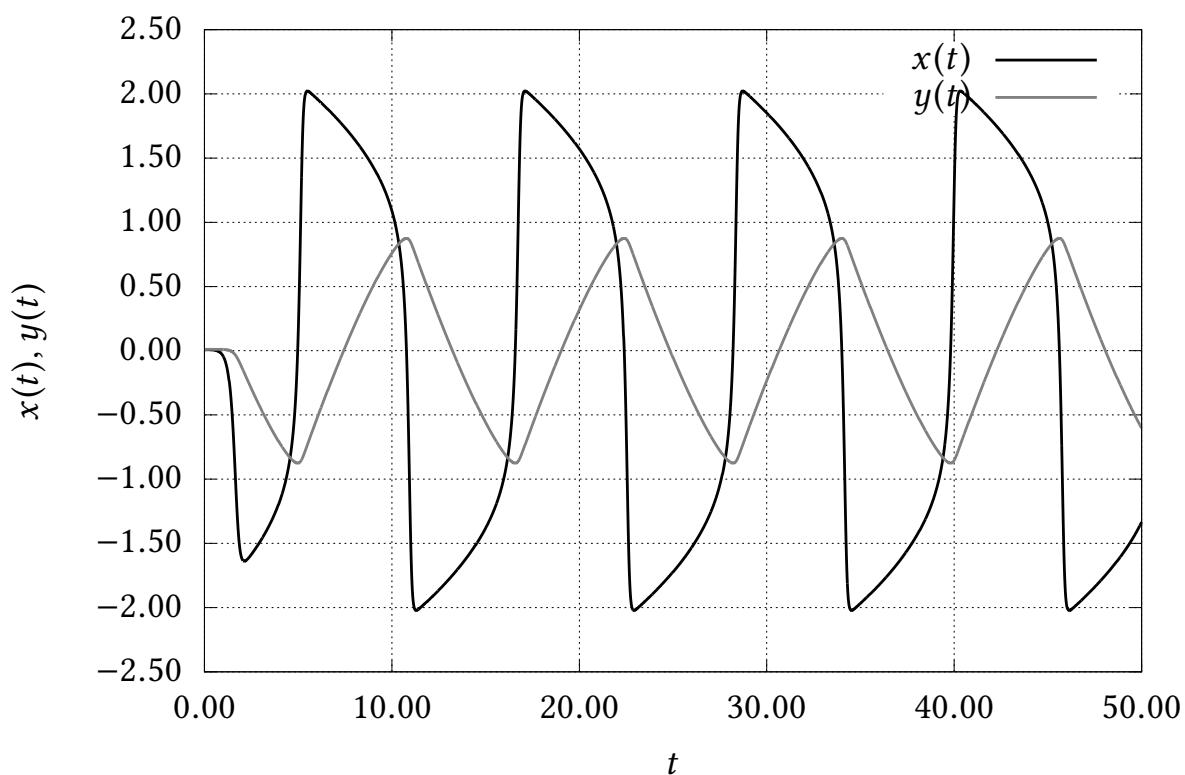


Figura 2.24: O oscilador não-linear de van der Pol ( $\mu = 5$ ).

# 3

## Solução numérica de equações diferenciais parciais

---

Dada uma função  $u$  de duas variáveis (digamos,  $x$  e  $t$ ) e uma equação diferencial parcial para a mesma, é possível encontrar representações aproximadas de  $u$  na forma  $u(x_i, t_n)$  em um reticulado no seu domínio. Essas soluções aproximadas são chamadas de soluções numéricas, e podem ser obtidas de diversas formas. Neste capítulo, nós fazemos uma breve introdução ao método de diferenças finitas para a sua obtenção.

Um elemento chave das soluções numéricas é que nós terminamos por obter sistemas algébricos de equações, cuja solução produz, em geral, uma parte dos pontos  $u(x_i, t_n)$ . À medida que o algoritmo da solução progride, mais e mais pontos são obtidos.

### 3.1 – Advecção pura: a onda cinemática

Considere a equação

$$\frac{\partial \phi}{\partial t} + c \frac{\partial \phi}{\partial x} = 0, \quad \phi(x, 0) = g(x). \quad (3.1)$$

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

$$\phi(x, t) = g(x - ct). \quad (3.2)$$

Seja então o problema

$$\frac{\partial \phi}{\partial t} + 2 \frac{\partial \phi}{\partial x} = 0, \quad (3.3)$$

$$\phi(x, 0) = 2x(1 - x). \quad (3.4)$$

A condição inicial, juntamente com  $u(x, 1)$ ,  $u(x, 2)$  e  $u(x, 3)$  estão mostrados na figura 3.1. Observe que a solução da equação é uma simples onda cinemática.

Vamos adotar a notação

$$\phi_i^n \equiv \phi(x_i, t_n), \quad (3.5)$$

$$x_i = i\Delta x, \quad (3.6)$$

$$t_n = n\Delta t, \quad (3.7)$$

com

$$\Delta x = L/N_x, \quad (3.8)$$

$$\Delta t = T/N_t, \quad (3.9)$$

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.

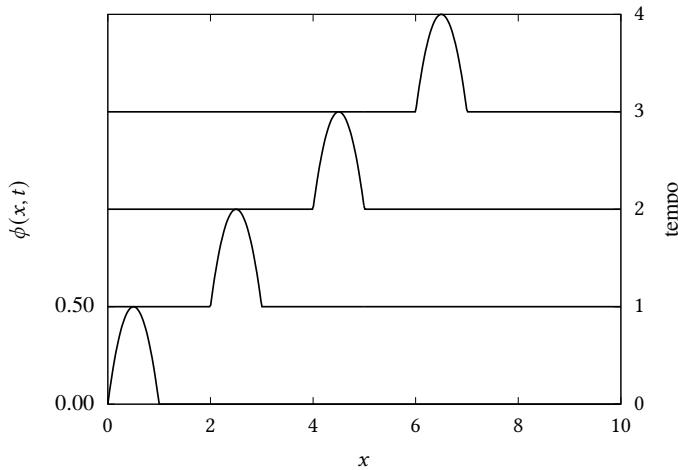


Figura 3.1: Solução analítica das equações 3.3–3.4.

Uma maneira simples de transformar as derivadas parciais em diferenças finitas na equação (3.3) é fazer

$$\frac{\partial \phi}{\partial t} \Big|_{i,n} = \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + \mathcal{O}(\Delta t), \quad (3.10)$$

$$\frac{\partial \phi}{\partial x} \Big|_{i,n} = \frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x} + \mathcal{O}(\Delta x^2). \quad (3.11)$$

Substituindo na equação (3.3), obtemos o esquema de diferenças finitas *explícito*:

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= -c \left( \frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x} \right), \\ \phi_i^{n+1} &= \phi_i^n - \frac{c\Delta t}{2\Delta x} (\phi_{i+1}^n - \phi_{i-1}^n), \end{aligned} \quad (3.12)$$

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

O programa que implementa o esquema instável é o `onda1d_ins.chpl`, mostrado na listagem 3.1. 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 lido pelo próximo programa na sequência, `surf1d_ins.py`, mostrado na listagem 3.2. O único trabalho desse programa é selecionar algumas “linhas” da saída de `onda1d-ins.py`; no caso, nós o rodamos com o comando

`[ surf1d-ins.py 3 250 ]`,

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

O resultado dos primeiros 750 intervalos de tempo de simulação é mostrado na figura 3.2. 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 3.1 (que vão até  $t = 4$ ). Claramente, o esquema explícito que nós programamos jamais nos levará a uma solução numérica satisfatória para tempos  $t \sim 1$ .

Por que o esquema utilizado em (3.12) 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, (3.12) jamais será calculada com precisão infinita. O que o computador realmente calcula é um valor *arredondado*  $\tilde{\phi}_i^n$ . Por enquanto, nós só vamos fazer essa

Listagem 3.1: onda1d\_ins.chpl – Solução de uma onda cinemática 1D com um método explícito instável

---

```

1 // -----
2 // onda1d_ins resolve uma equação de onda cinemática com um método explícito
3 //
4 // uso: ./onda1d_ins
5 //
6 use IO only openWriter, binarySerializer;;
7 const fou = openWriter("onda1d_ins.dat",
8                     serializer = new binarySerializer(), locking=false);
9 const dx = 0.01;
10 const dt = 0.0005;
11 writef("// dx = %9.4dr\n",dx);
12 writef("// dy = %9.4dr\n",dt);
13 const nx = round(10.0/dx):int;           // número de pontos em x
14 const nt = round(1.0/dt):int;           // número de pontos em t
15 writef("// nx = %9i\n",nx);
16 writef("// nt = %9i\n",nt);
17 var phi: [0..1,0..nx] real;             // apenas 2 posições no tempo
18                                         // são necessárias!
19                                         // monta a condição inicial
20 for i in 0..nx do {
21     var xi = i*dx;
22     phi[0,i] = CI(xi);
23 }
24 fou.write(phi[0,0..nx]);                // imprime a condição inicial
25 var iold = 0;
26 var inew = 1;
27 const c = 2.0;                         // celeridade da onda
28 const couhalf = c*dt/(2.0*dx);         // metade do número de Courant
29 for n in 1..nt do {                   // loop no tempo
30     writeln("n = ",n);
31     for i in 1..nx-1 do {              // loop no espaço
32         phi[inew,i] = phi[iold,i] - couhalf*(phi[iold,i+1] - phi[iold,i-1]);
33     }
34     phi[inew,0] = 0.0;
35     phi[inew,nx] = 0.0;
36     fou.write(phi[inew,0..nx]);        // imprime uma linha com os novos dados
37     iold <=> inew;                  // troca os índices
38 }
39 fou.close();                           // define a condição inicial
40 proc CI(const in x: real): real {
41     if 0 <= x && x <= 1.0 then {
42         return 2.0*x*(1.0-x);
43     }
44     else {
45         return 0.0;
46     }

```

---

Listagem 3.2: `surf1d_ins.chpl` – Seleciona alguns intervalos de tempo da solução numérica para plotagem

---

```

1 // -----
2 // surf1d_ins.chpl: imprime em <arq> <m>+1 saídas de onda1d_ins a cada <n>
3 // intervalos de tempo
4 //
5 // uso: ./surf1d_ins --m=<m> --n=<n>
6 // -----
7 const dx = 0.01;
8 const dt = 0.0005;
9 writef("// dx = %9.4dr\n",dx);
10 writef("// dy = %9.4dr\n",dt);
11 const nx = round(10.0/dx):int;           // número de pontos em x
12 writef("// nx = %9i\n",nx);
13 config const m: int;                   // m saídas
14 config const n: int;                   // a cada n intervalos de tempo
15 writef("// m = %9i\n",m);
16 writef("// n = %9i\n",n);
17 use IO only openReader, openWriter, binaryDeserializer;;
18 const fin = openReader("onda1d_ins.dat",
                        deserializer = new binaryDeserializer(), locking=false);
19 var phi: [0..nx] real;                 // dados de um intervalo de tempo
20 var v: [0..m,0..nx] real;              // um array com m+1 intervalos de tempo
21 fin.read(phi);                      // lê a condição inicial
22 v[0,0..nx] = phi;                   // inicializa a lista da "transposta"
23 for it in 1..m do {                  // para <m> instantes:
24     for ir in 1..n do {              // lê <n> vezes, só guarda a última
25         fin.read(phi);
26     }
27     v[it,0..nx] = phi;             // guarda a última
28 }
29 }
30 const founam = "surf1d_ins.dat";
31 writeln(founam);
32 const fou = openWriter(founam,
                        locking=false); // abre o arquivo de saída
33 for i in 0..nx do {
34     fou.writef("%10.6dr",i*dx);      // escreve o "x"
35     fou.writef("%10.6dr",v[0,i]);      // escreve a cond inicial
36     for k in 1..m do {
37         fou.writef("%10.6dr",v[k,i]);    // escreve o k-ésimo
38     }
39     fou.writef("\n");
40 }
41 }
42 fou.close();

```

---

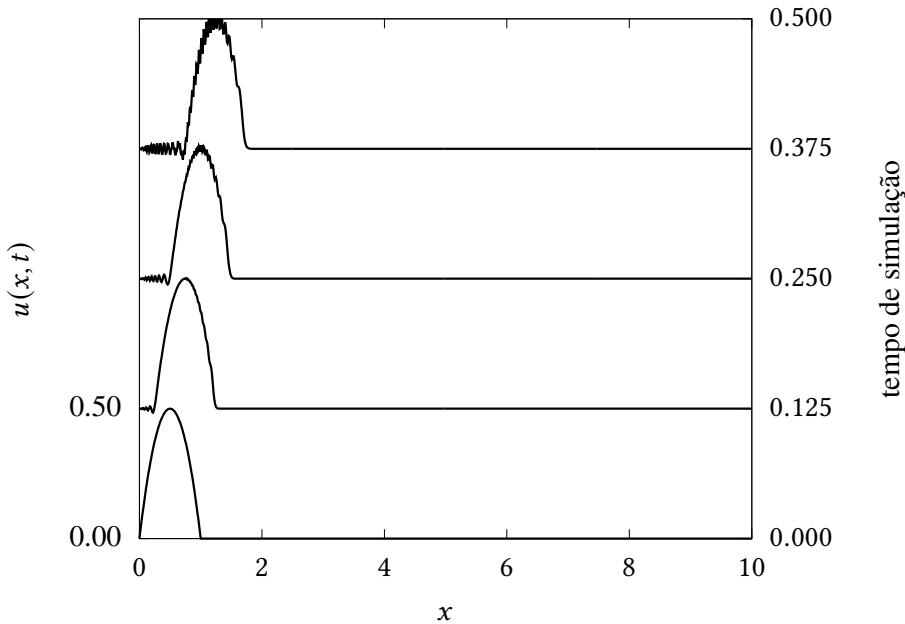


Figura 3.2: Solução numérica produzida por `onda1d-ins.chpl`, para  $t = 250\Delta t$ ,  $500\Delta t$  e  $750\Delta t$ .

distinção de notação, entre  $\tilde{\phi}$  e  $\phi$ , aqui, onde ela importa. O *erro de arredondamento* é

$$\epsilon_i^n \equiv \tilde{\phi}_i^n - \phi_i^n. \quad (3.13)$$

Note que (3.12) se aplica tanto para  $\phi$  quanto para  $\tilde{\phi}$ ; subtraindo as equações resultantes para  $\tilde{u}_i^{n+1}$  e  $u_i^{n+1}$ , obtém-se a *mesma* equação para a evolução de  $\epsilon_i^n$ :

$$\epsilon_i^{n+1} = \epsilon_i^n - \frac{Co}{2} (\epsilon_{i+1}^n - \epsilon_{i-1}^n), \quad (3.14)$$

onde

$$Co \equiv \frac{c\Delta t}{\Delta x} \quad (3.15)$$

é o *número de Courant*. Isso só foi possível porque (3.12) é uma equação *linear* em  $\phi$ . 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  $\epsilon_i^n$ , na forma

$$\begin{aligned} t_n &= n\Delta t, \\ x_i &= i\Delta x, \\ \epsilon_i^n &= \sum_{l=1}^{N/2} \xi_l e^{at_n} e^{ik_l x_i}, \end{aligned} \quad (3.16)$$

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  $x$ .

Argumentando novamente com a linearidade, desta vez de (3.14), ela vale para cada *modo*  $l$  de (3.16), donde

$$\xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} = \xi_l e^{at_n} e^{ik_l i \Delta x} - \frac{Co}{2} \left( \xi_l e^{at_n} e^{ik_l (i+1)\Delta x} - \xi_l e^{at_n} e^{ik_l (i-1)\Delta x} \right); \quad (3.17)$$

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

$$e^{a\Delta t} = 1 - \frac{Co}{2} \left( e^{+ik_l \Delta x} - e^{-ik_l \Delta x} \right)$$

$$= 1 - i\text{Co} \sin k_l \Delta x. \quad (3.18)$$

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,

$$\begin{aligned} e^{(\alpha-i\beta)\Delta t} &= 1 - i\text{Co} \sin k_l \Delta x; \\ e^{\alpha\Delta t} [\cos(\beta\Delta t) - i \sin(\beta\Delta t)] &= 1 - i\text{Co} \sin k_l \Delta x; \Rightarrow \\ e^{\alpha\Delta t} \cos(\beta\Delta t) &= 1, \end{aligned} \quad (3.19)$$

$$e^{\alpha\Delta t} \sin(\beta\Delta t) = \text{Co} \sin(k_l \Delta x). \quad (3.20)$$

As duas últimas equações formam um sistema não-linear nas incógnitas  $\alpha$  e  $\beta$ . O sistema pode ser resolvido:

$$\tan(\beta\Delta t) = \text{Co} \sin(k_l \Delta x) \Rightarrow \beta\Delta t = \arctan(\text{Co} \sin(k_l \Delta x)).$$

Note que  $\beta \neq 0$ , donde  $e^{\alpha\Delta t} > 1$  via (3.19), e o esquema de diferenças finitas é *incondicionalmente instável*.

**Lax** Uma alternativa que produz um esquema estável é o método de Lax:

$$\phi_i^{n+1} = \frac{1}{2} [(\phi_{i+1}^n + \phi_{i-1}^n) - \text{Co}(\phi_{i+1}^n - \phi_{i-1}^n)]. \quad (3.21)$$

Agora que nós já sabemos que esquemas numéricos podem ser instáveis, devemos fazer uma análise de estabilidade *antes* de tentar implementar (3.21) numericamente. Vamos a isso: utilizando novamente (3.16) e substituindo em (3.21), temos

$$\begin{aligned} \xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} &= \frac{1}{2} \left[ \left( \xi_l e^{at_n} e^{ik_l(i+1)\Delta x} + \xi_l e^{at_n} e^{ik_l(i-1)\Delta x} \right) \right. \\ &\quad \left. - \text{Co} \left( \xi_l e^{at_n} e^{ik_l(i+1)\Delta x} - \xi_l e^{at_n} e^{ik_l(i-1)\Delta x} \right) \right]; \\ e^{a\Delta t} &= \frac{1}{2} \left[ \left( e^{+ik_l \Delta x} + e^{-ik_l \Delta x} \right) - \text{Co} \left( e^{+ik_l \Delta x} - e^{-ik_l \Delta x} \right) \right]; \\ e^{a\Delta t} &= \cos(k_l \Delta x) - i\text{Co} \sin(k_l \Delta x). \end{aligned} \quad (3.22)$$

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

$$\text{Co} \leq 1, \quad (3.23)$$

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

A “mágica” de (3.21) é que ela introduz um pouco de *difusão numérica*; de fato, podemos reescrevê-la na forma

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= -c \frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x} + \frac{\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{2\Delta t} \\ &= -c \frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x} + \left( \frac{\Delta x^2}{2\Delta t} \right) \frac{\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{\Delta x^2}. \end{aligned} \quad (3.24)$$

Não custa repetir: (3.24) é *idêntica* a (3.21). Porém, comparando-a com (3.12) (nossa esquema instável inicialmente empregado), nós vemos que ela também é equivalente a essa última, *com o termo adicional*  $(\Delta x^2/2\Delta t)(\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n)/\Delta x^2$ . O que esse termo adicional significa? A resposta é *uma derivada numérica de ordem 2*. De fato, considere as expansões em série de Taylor

$$\begin{aligned}\phi_{i+1} &= \phi_i + \frac{d\phi}{dx} \Big|_i \Delta x + \frac{1}{2} \frac{d^2\phi}{dx^2} \Big|_i \Delta x^2 + \mathcal{O}(\Delta x^3), \\ \phi_{i-1} &= \phi_i - \frac{d\phi}{dx} \Big|_i \Delta x + \frac{1}{2} \frac{d^2\phi}{dx^2} \Big|_i \Delta x^2 + \mathcal{O}(\Delta x^3),\end{aligned}$$

e some:

$$\begin{aligned}\phi_{i+1} + \phi_{i-1} &= 2\phi_i + \frac{d^2\phi}{dx^2} \Big|_i \Delta x^2 + \mathcal{O}(\Delta x^4), \\ \frac{d^2\phi}{dx^2} \Big|_i &= \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} + \mathcal{O}(\Delta x^2).\end{aligned}\quad (3.25)$$

Portanto, a equação (3.24) – ou seja: o esquema de Lax (3.21) – pode ser interpretada *também* como uma solução aproximada da equação de advecção-difusão

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

com

$$D = \frac{\Delta x^2}{2\Delta t}.$$

Note que  $D$  tem dimensões de *difusividade*:  $\llbracket D \rrbracket = L^2 T^{-1}$ . 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 arredondamento.

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 isso com (por exemplo) a magnitude das velocidades introduzidas pela difusão numérica; devemos portanto verificar se

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

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  $Co < 1$ , o critério de que a solução seja *também* fisicamente acurada demanda que  $Co \gg 1/2$ . Na prática, isso significa que, para  $c = 2$ , *ou* o esquema é estável com muita difusão numérica, ou ele é instável. Na prática, isso restringe o uso de (3.21).

Mesmo assim, vamos programá-lo! O programa `onda1d_lax.chpl` está mostrado na listagem 3.3. Ele usa os mesmos valores  $\Delta t = 0,0005$  e  $\Delta x = 0,01$ , ou seja,  $Co = 0,10$ .

O programa gera um arquivo de saída binário, que por sua vez é lido pelo próximo programa na sequência, `surf1d_lax.chpl`, mostrado na listagem 3.4. O único trabalho desse programa é selecionar algumas “linhas” da saída de `onda1d_lax.chpl`; no caso, nós o rodamos com o comando

`[ ./surf1d-lax 3 500 ] ,`

Listagem 3.3: onda1d\_lax.chpl – Solução de uma onda cinemática 1D com o método de Lax

---

```

1 // -----
2 // onda1d-lax resolve uma equação de onda cinemática com um método explícito
3 //
4 // uso: ./onda1d-lax
5 //
6 use IO only openWriter, binarySerializer;
7 const fou = openWriter("onda1d_lax.dat",
8     serializer = new binarySerializer(), locking=false);
9 const dx = 0.01;
10 const dt = 0.0005;
11 writef("// dx = %9.4dr\n",dx);
12 writef("// dy = %9.4dr\n",dt);
13 const nx = round(10.0/dx):int;           // número de pontos em x
14 const nt = round(1.0/dt):int;           // número de pontos em t
15 writef("// nx = %9i\n",nx);
16 writef("// nt = %9i\n",nt);
17 var u: [0..1,0..nx] real;               // apenas 2 posições no tempo são
18                                         // necessárias
19 for i in 0..nx do {                   // monta a condição inicial
20     var xi = i*dx;
21     u[0,i] = CI(xi);
22 }
23 fou.write(u[0,0..nx]);                 // imprime a condição inicial
24 var iold = 0;
25 var inew = 1;
26 const c = 2.0;                         // celeridade da onda
27 const cou = c*dt/(dx);                 // número de Courant
28 writef("Co = %10.6dr\n",cou);
29 for n in 1..nt do {                   // loop no tempo
30     for i in 1..nx-1 do {             // loop no espaço
31         u[inew,i] = 0.5*((u[iold,i+1] + u[iold,i-1]) -
32             cou*(u[iold,i+1] - u[iold,i-1]));
33         u[inew,0] = 0.0;
34         u[inew,nx] = 0.0;
35     }
36     fou.write(u[inew,0..nx]);          // imprime uma linha com os novos dados
37     iold <=> inew;                  // troca os índices
38 }
39 fou.close();
40 proc CI(const in x: real): real {      // define a condição inicial
41     if 0 <= x && x <= 1.0 then {
42         return 2.0*x*(1.0-x);
43     }
44     else {
45         return 0.0;
46     }
47 }
```

---

Listagem 3.4: `surf1d_lax.chpl` – Seleciona alguns intervalos de tempo da solução numérica para plotagem

---

```

1 // -----
2 // surf1d_lax.chpl: imprime em <arq> <m>+1 sadas de onda1d_lax a cada <n>
3 // intervalos de tempo
4 //
5 // uso: ./surf1d_lax --m=<m> --n=<n>
6 // -----
7 const dx = 0.01;
8 const dt = 0.0005;
9 writef("// dx = %9.4dr\n",dx);
10 writef("// dy = %9.4dr\n",dt);
11 const nx = round(10.0/dx):int;           // numero de pontos em x
12 writef("// nx = %9i\n",nx);
13 config const m: int;                   // m sadas
14 config const n: int;                   // a cada n intervalos de tempo
15 writef("// m = %9i\n",m);
16 writef("// n = %9i\n",n);
17 use IO only openReader, openWriter, binaryDeserializer;;
18 const fin = openReader("onda1d_lax.dat",
                        deserializer = new binaryDeserializer(), locking=false);
19 var u: [0..nx] real;                  // dados de um intervalo de tempo
20 var v: [0..m,0..nx] real;             // um array com m+1 intervalos de tempo
21 fin.read(u);                         // l a condio inicial
22 v[0,0..nx] = u;                     // inicializa a lista da "transposta"
23 for it in 1..m do {                 // para <m> laptantes:
24     for ir in 1..n do {              // l <n> vezes, s guarda a ltima
25         fin.read(u);
26         }
27     v[it,0..nx] = u;                // guarda a ltima
28 }
29 }
30 const founam = "surf1d_lax.dat";
31 writeln(founam);
32 const fou = openWriter(founam,
                        locking=false); // abre o arquivo de sada
33 for i in 0..nx do {
34     fou.writef("%10.6dr",i*dx);      // escreve o "x"
35     fou.writef("%10.6dr",v[0,i]);    // escreve a cond inicial
36     for k in 1..m do {
37         fou.writef("%10.6dr",v[k,i]); // escreve o k-simo
38     }
39     fou.writef("\n");
40 }
41 }
42 fou.close();

```

---

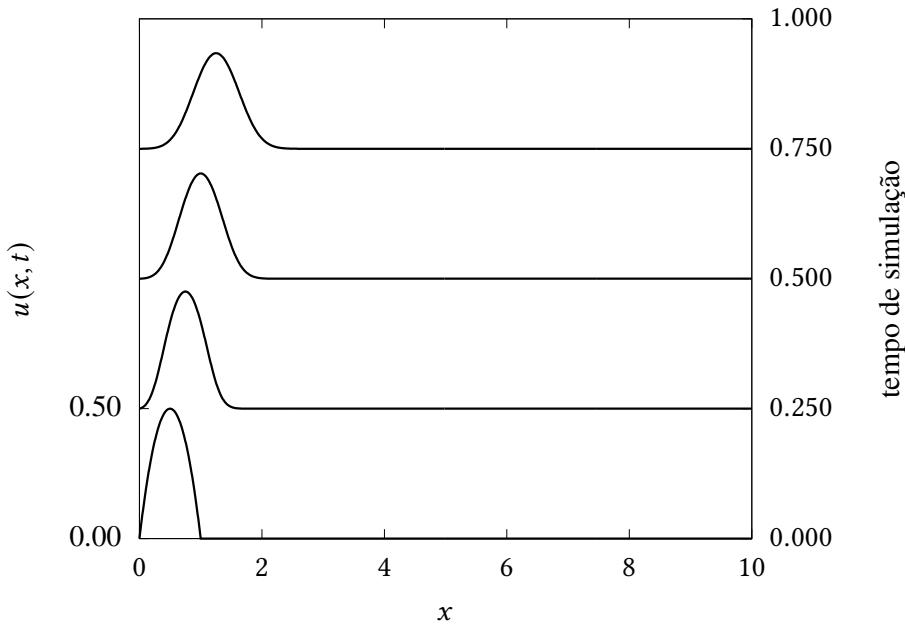


Figura 3.3: Solução numérica produzida por `onda1d_lax.chpl`, para  $t = 500\Delta t$ ,  $1000\Delta t$  e  $1500\Delta t$ .

o que significa selecionar 3 saídas (além da condição inicial), de 500 em 500 intervalos de tempo  $\Delta t$ . Com isso, nós conseguimos chegar até o instante 0,75 da simulação.

O resultado dos primeiros 1500 intervalos de tempo de simulação é mostrado na figura 3.3. Observe que agora não há oscilações espúrias: o esquema é estável no tempo. No entanto, a solução está “amortecida” pela difusão numérica!

**Upwind** Um esquema que é conhecido na literatura como indicado por representar melhor o termo advectivo em (3.1) é o esquema de diferenças regressivas; nesse esquema, chamado de esquema *upwind* – literalmente, “corrente acima” na literatura de língua inglesa – a discretização utilizada é

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= -c \frac{\phi_i^n - \phi_{i-1}^n}{\Delta x}, \\ \phi_i^{n+1} &= \phi_i^n - \text{Co} [\phi_i^n - \phi_{i-1}^n]. \end{aligned} \quad (3.26)$$

Claramente, estamos utilizando um esquema de  $\mathcal{O}(\Delta x)$  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 tanta 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 isso:

$$\begin{aligned} \xi_l e^{a(t_n + \Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} - \text{Co} [\xi_l e^{at_n} e^{ik_l i \Delta x} - \xi_l e^{at_n} e^{ik_l (i-1) \Delta x}] \\ e^{a \Delta t} e^{ik_l i \Delta x} &= e^{ik_l i \Delta x} - \text{Co} [e^{ik_l i \Delta x} - e^{ik_l (i-1) \Delta x}] \\ e^{a \Delta t} &= 1 - \text{Co} [1 - e^{-ik_l \Delta x}] \\ e^{a \Delta t} &= 1 - \text{Co} + \text{Co} \cos(k_l \Delta x) - i \text{Co} \sin(k_l \Delta x). \end{aligned} \quad (3.27)$$

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

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

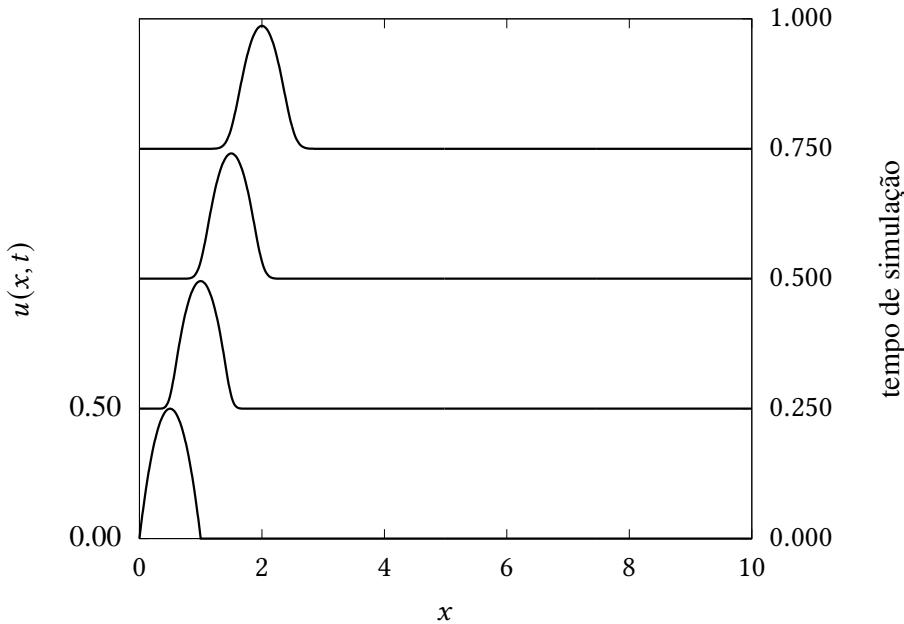


Figura 3.4: Solução numérica produzida pelo esquema *upwind*, para  $t = 250, 500$  e  $750$ .

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,

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

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

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

Reencontramos, portanto, a condição (3.23), mas em um outro esquema de diferenças finitas. A lição não deve ser mal interpretada: longe de supor que (3.23) vale sempre, é a análise de estabilidade que deve ser 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  $\Delta t$  e de  $\Delta x$  de antes. As mudanças necessárias nos códigos computacionais são óbvias, e são deixadas a cargo do(a) leitor(a).

A figura 3.4 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 também está ocorrendo, embora em grau muito menor.

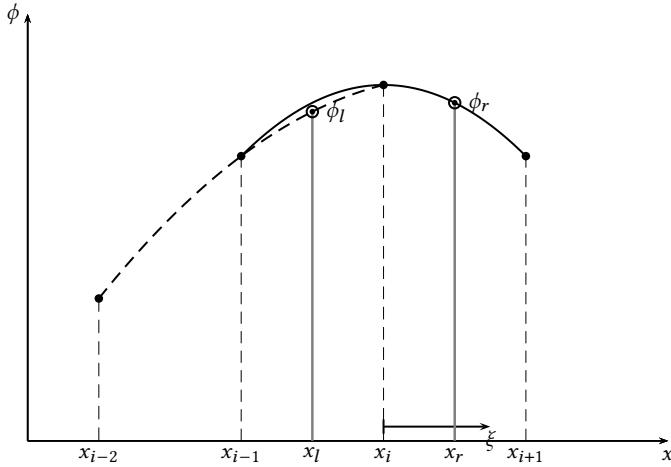


Figura 3.5: Quick's interpolation scheme.

**Quick** [?] proposed two quadratic interpolations, centered at  $x_{i-1}$  and  $x_i$ , to calculate a centered approximation for  $d\phi/dx$  at  $x_i$ , as depicted in figure 3.5. For definitiveness, consider the parabola through  $(x_{i-1}, \phi_{i-1})$ ,  $(x_i, \phi_i)$ , and  $(x_{i+1}, \phi_{i+1})$ , and set a local axis  $\xi$  centered at  $x_i$  such that  $\xi = x - x_i$ . The equation for the parabola is then

$$\phi = a_0 + a_1\xi + a_2\xi^2, \quad (3.28)$$

such that

$$\phi_i = a_0, \quad (3.29)$$

$$\phi_{i-1} = a_0 - a_1\Delta x + a_2\Delta x^2, \quad (3.30)$$

$$\phi_{i+1} = a_0 + a_1\Delta x + a_2\Delta x^2. \quad (3.31)$$

The values of  $a_1$  and  $a_2$  can easily be obtained by subtracting and summing (3.30) and (3.31):

$$\begin{aligned} \phi_{i+1} - \phi_{i-1} &= 2a_1\Delta x, \\ a_1 &= \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x}; \end{aligned} \quad (3.32)$$

and

$$\begin{aligned} \phi_{i+1} + \phi_{i-1} &= 2a_0 + 2a_2\Delta x^2 \\ &= 2\phi_i + 2a_2\Delta x^2; \end{aligned} \quad (3.33)$$

$$a_2 = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{2\Delta x^2}. \quad (3.34)$$

The value of  $\phi_r$  is then obtained for  $\xi = \Delta x/2$  as

$$\begin{aligned} \phi_r &= \phi_i + \frac{(\phi_{i+1} - \phi_{i-1})}{2\Delta x} \frac{\Delta x}{2} + \frac{(\phi_{i+1} - 2\phi_i + \phi_{i-1})}{2\Delta x^2} \frac{\Delta x^2}{4} \\ &= \phi_i + \frac{1}{4}(\phi_{i+1} - \phi_{i-1}) + \frac{1}{8}(\phi_{i+1} - 2\phi_i + \phi_{i-1}) \\ &= \frac{1}{2}(\phi_i + \phi_{i+1}) - \frac{1}{8}(\phi_{i+1} - 2\phi_i + \phi_{i-1}). \end{aligned} \quad (3.35)$$

By the same token we have, for  $\phi_l$ ,

$$\phi_l = \frac{1}{2}(\phi_{i-1} + \phi_i) - \frac{1}{8}(\phi_i - 2\phi_{i-1} + \phi_{i-2}). \quad (3.36)$$

Listagem 3.5: quick\_grid.chpl – A stable grid for QUICK’s solution of (3.1)

---

```

1 const dx = 0.01;
2 const dt = 0.000002;
3 writef("// dx = %9.6dr\n",dx);
4 writef("// dt = %9.6dr\n",dt);
5 const nx = round(10.0/dx):int;           // número de pontos em x
6 const nt = round(1.0/dt):int;           // número de pontos em t
7 writef("// nx = %9i\n",nx);
8 writef("// nt = %9i\n",nt);
9 const c = 2.0;                          // celeridade da onda
10 const cou = c*dt/(dx);                // número de Courant
11 writef("Co = %10.6dr\n",cou);

```

---

Discretization of (3.1) now produces

$$\begin{aligned}
\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= -c \frac{\phi_r^n - \phi_l^n}{\Delta x}, \\
\phi_i^{n+1} - \phi_i^n &= -\frac{c\Delta t}{\Delta x} \left\{ \left[ \frac{1}{2}(\phi_i^n + \phi_{i+1}^n) - \frac{1}{8}(\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n) \right] - \left[ \frac{1}{2}(\phi_{i-1}^n + \phi_i^n) - \frac{1}{8}(\phi_{i-2}^n - 2\phi_{i-1}^n + \phi_i^n) \right] \right\} \\
&= -Co \left\{ \frac{1}{2}(\phi_{i+1}^n - \phi_{i-1}^n) - \frac{1}{8} (\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n - \phi_{i-2}^n + 2\phi_{i-1}^n - \phi_i^n) \right\} \\
&= -Co \left\{ \frac{1}{2}(\phi_{i+1}^n - \phi_{i-1}^n) - \frac{1}{8} (-\phi_{i-2}^n + 3\phi_{i-1}^n - 3\phi_i^n + \phi_{i+1}^n) \right\} \\
&= -\frac{Co}{8} (\phi_{i-2}^n - 7\phi_{i-1}^n + 3\phi_i^n + 3\phi_{i+1}^n),
\end{aligned}$$

or

$$\phi_i^{n+1} = \phi_i^n - \frac{Co}{8} (\phi_{i-2}^n - 7\phi_{i-1}^n + 3\phi_i^n + 3\phi_{i+1}^n). \quad (3.37)$$

Although QUICK is a very popular scheme in computational fluid dynamics, for the case of the pure kinematic wave equation (3.1) it turns out to be *almost* unconditionally unstable! This is clearly discussed in ?; their Eq. (18) for the simple case of (3.1) gives

$$Co \leq \frac{1}{2} \left( \frac{\pi}{N} \right)^2, \quad (3.38)$$

where  $n$  is the number of grid points. Consider again listing 3.3; for  $\Delta x = 0.01$ , we have  $N = 100$ , and now the stability condition is

$$Co \leq \frac{1}{2} \left( \frac{\pi}{100} \right)^2 = 0.000493!$$

For the same number of points in  $x$ , let us set up a grid that meets that requirement, in file quick\_grid.chpl. We now have  $Co = 0.0004$ , and proceed to write a numerical solution using QUICK, in listing 3.6.

In order to “see” the results, we have surf1d\_quick.chpl in listing 3.7, that we run with

[ ./surf1d\_quick -m=3 -n=125000 ].

The results of using QUICK are shown, for the same instants of time used before, in figure 3.6. Visually, QUICK displays the least numerical diffusion of all schemes used so far, but at a huge cost in numerical processing. Maybe this will change when physical diffusion is included, later on.

---

Listagem 3.6: onda1d\_quick.chpl – Solution of a 1D kinematic wave using QUICK

---

```

1 // -----
2 // onda1d_quick resolve uma equação de onda cinemática com o método quick. Note
3 // o uso do ponto-fantasma i=-1 devido à ordem do esquema
4 //
5 // uso: ./onda1d_quick
6 // -----
7 use quick_grid; //
8 use IO only openWriter, binarySerializer;
9 const fou = openWriter("onda1d_quick.dat",
10           serializer = new binarySerializer(), locking=false);
11 var phi: [0..1,-1..nx] real;           // apenas 2 posições no tempo são
12                                // necessárias
13 for i in -1..nx do {                 // monta a condição inicial
14   var xi = i*dx;
15   phi[0,i] = CI(xi);
16 }
17 fou.write(phi[0,-1..nx]);            // imprime a condição inicial
18 var iold = 0;
19 var inew = 1;
20 for n in 1..nt do {                // loop no tempo
21   if n % 1000 == 0 then writeln(n);
22   for i in 1..nx-1 do {            // loop no espaço
23     phi[inew,i] = phi[iold,i] - (cou/8.0)*(phi[iold,i-2] - 7.0*phi[iold,i-1] +
24       3.0*phi[iold,i] + 3.0*phi[iold,i+1]);
25     phi[inew,-1] = 0.0;
26     phi[inew,0] = 0.0;
27     phi[inew,nx] = 0.0;
28   }
29   fou.write(phi[inew,-1..nx]);      // imprime uma linha com os novos dados
30   iold <=> inew;                  // troca os índices
31 }
32 fou.close();
33 proc CI(const in x: real): real {    // define a condição inicial
34   if 0 <= x && x <= 1.0 then {
35     return 2.0*x*(1.0-x);
36   }
37   else {
38     return 0.0;
39   }
40 }
```

---

Listagem 3.7: `surf1d_quick.chpl` – Seleciona alguns intervalos de tempo da solução numérica com QUICK para plotagem

---

```

1 // -----
2 // surf1d_quick.chpl: imprime em <arq> <m>+1 sadas de onda1d_quick a cada <n>
3 // intervalos de tempo
4 //
5 // uso: ./surf1d_quick --m=<m> --n=<n>
6 // -----
7 use quick_grid;
8 config const m: int;                      // m sadas
9 config const n: int;                      // a cada n intervalos de tempo
10 writef("// m = %9i\n",m);
11 writef("// n = %9i\n",n);
12 writef("// n Delta t = %9.4dr\n",n*dt);
13 use IO only openReader, openWriter, binaryDeserializer;;
14 const fin = openReader("onda1d_quick.dat",
15                         serializer = new binaryDeserializer(), locking=false);
16 var phi: [-1..nx] real;                   // dados de um intervalo de tempo
17 var v: [0..m,-1..nx] real;                // um array com m+1 intervalos de tempo
18 fin.read(phi);                          // l a condio inicial
19 v[0,-1..nx] = phi;                     // inicializa a lista da "transposta"
20 for it in 1..m do {                     // para <m> instantes:
21     for ir in 1..n do {                 // l <n> vezes, s guarda a ltima
22         fin.read(phi);
23     }
24     v[it,-1..nx] = phi;               // guarda a ltima
25 }
26 const founam = "surf1d_quick.dat";
27 writeln(founam);
28 const fou = openWriter(founam,
29                         locking=false); // abre o arquivo de sada
30 for i in -1..nx do {
31     fou.writef("%10.6dr",i*dx);        // escreve o "x"
32     fou.writef("%10.6dr",v[0,i]);       // escreve a cond inicial
33     for k in 1..m do {
34         fou.writef("%10.6dr",v[k,i]);   // escreve o k-simo
35     }
36     fou.writef("\n");
37 }
38 fou.close();

```

---

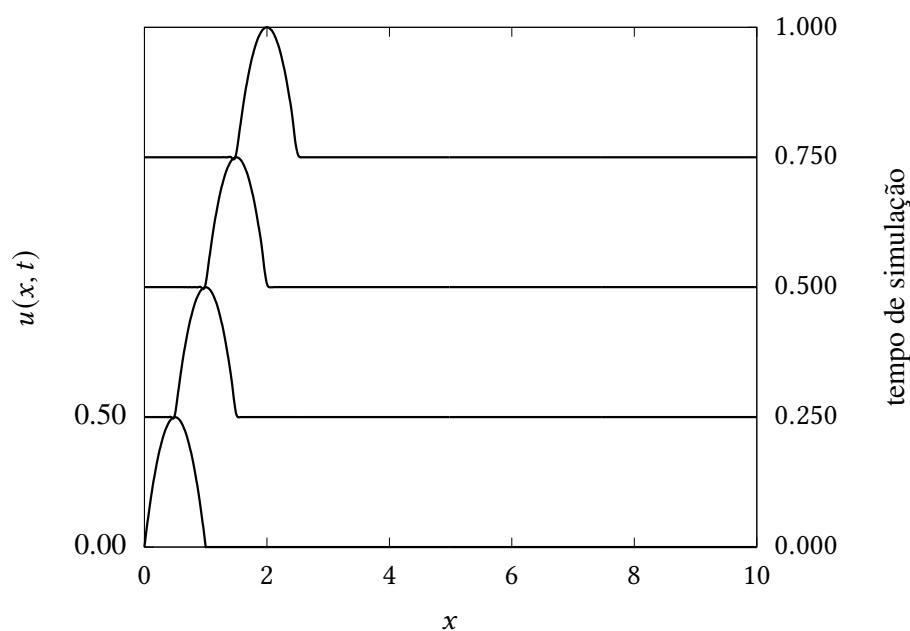


Figura 3.6: Solução numérica produzida pelo esquema QUICK, para  $t = 250, 500$  e  $750$ .

**Quickest** Consider a control “volume” centered at  $x_i$  with faces at  $x_l$  and  $x_r$ . The convective fluxes through the faces are  $-c\phi_l$  and  $+c\phi_r$ . Let us expand the analytical solution of (3.1) in a Taylor series up to order 3,

$$\phi(\xi, t) = \phi(0) + \frac{\partial\phi(0, t)}{\partial\xi}\xi + \frac{1}{2}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\xi^2 + \frac{1}{6}\frac{\partial^3\phi(0, t)}{\partial\xi^3}\xi^3 + \mathcal{O}(\xi^4);$$

the mean value of  $\phi$  inside the control volume is

$$\begin{aligned}\bar{\phi}(t) &= \frac{1}{\Delta x} \int_{\xi=-\Delta x/2}^{\xi=+\Delta x/2} \left[ \phi(0) + \frac{\partial\phi(0, t)}{\partial\xi}\xi + \frac{1}{2}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\xi^2 + \frac{1}{6}\frac{\partial^3\phi(0, t)}{\partial\xi^3}\xi^3 + \mathcal{O}(\xi^4) \right] d\xi \\ &= \frac{1}{\Delta x} \left[ \phi(0, t)\xi + \frac{1}{2}\frac{\partial\phi(0, t)}{\partial\xi}\xi^2 + \frac{1}{3} \times \frac{1}{2}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\xi^3 + \frac{1}{4} \times \frac{1}{6}\frac{\partial^3\phi(0, t)}{\partial\xi^3}\xi^4 + \mathcal{O}(\xi^5) \right]_{-\Delta x/2}^{+\Delta x/2} \\ &= \frac{1}{\Delta x} \left[ \phi(0, t)\Delta x + \frac{1}{3}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\frac{\Delta x^3}{8} + \mathcal{O}(\Delta x^5) \right] \\ &= \phi(0, t) + \frac{1}{24}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\Delta x^2 + \mathcal{O}(\Delta x^4).\end{aligned}\tag{3.39}$$

The QUICKEST method, also proposed by ?, is a finite-volume version of QUICK which can be derived from (3.39) ???. To see this, write the control-volume balance equivalent to (3.1) as

$$\begin{aligned}\frac{\partial}{\partial t} \int_{-\Delta x/2}^{+\Delta x/2} \phi(\xi, t) d\xi + [-c\phi(-\Delta x/2, t) + c\phi(+\Delta x/2, t)] &= 0, \\ \frac{\partial[\Delta x \bar{\phi}]}{\partial t} + c [\phi(+\Delta x/2, t) - \phi(-\Delta x/2, t)] &= 0, \\ \frac{d\bar{\phi}}{dt} + c \frac{[\phi(+\Delta x/2, t) - \phi(-\Delta x/2, t)]}{\Delta x} &= 0.\end{aligned}\tag{3.40}$$

But now we need to differentiate (3.39)! Let's go:

$$\begin{aligned}\frac{d\bar{\phi}}{dt} &= \frac{\partial}{\partial t} \left[ \phi(0, t) + \frac{1}{24}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\Delta x^2 + \mathcal{O}(\Delta x^4) \right] \\ &= \frac{\partial\phi(0, t)}{\partial t} + \frac{1}{24}\frac{\partial^2}{\partial\xi^2}\frac{\partial\phi(0, t)}{\partial t}\Delta x^2 + \mathcal{O}(\Delta x^4).\end{aligned}$$

Here is the big, mean trick:

$$\begin{aligned}\frac{\partial\phi}{\partial t} &= -c\frac{\partial\phi}{\partial x} = -c\frac{\partial\phi}{\partial\xi}, \\ \frac{\partial^2}{\partial\xi^2}\frac{\partial\phi(0, t)}{\partial t} &= -c\frac{\partial^2}{\partial\xi^2}\frac{\partial\phi(0, t)}{\partial\xi} = -c\frac{\partial}{\partial\xi} \left[ \frac{\partial^2\phi(0, t)}{\partial\xi^2} \right].\end{aligned}$$

Hence,

$$\frac{d\bar{\phi}}{dt} = \frac{\partial\phi(0, t)}{\partial t} - \frac{c}{24}\frac{\partial}{\partial\xi} \left[ \frac{\partial^2\phi(0, t)}{\partial\xi^2} \right] \Delta x^2 + \mathcal{O}(\Delta x^4).$$

Put

$$\begin{aligned}\frac{\partial\phi(0, t)}{\partial t} &\approx \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t}, \\ \frac{\partial}{\partial\xi} \left[ \frac{\partial^2\phi(0, t)}{\partial\xi^2} \right] \Delta x^2 &\approx \frac{1}{\Delta x} \left[ \frac{\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n}{\Delta x^2} - \frac{\phi_{i-2}^n - 2\phi_{i-1}^n + \phi_i^n}{\Delta x^2} \right] \Delta x^2 \\ &= \frac{1}{\Delta x} [(\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n) - (\phi_{i-2}^n - 2\phi_{i-1}^n + \phi_i^n)],\end{aligned}$$

and apply (3.35)–(3.36) for  $\phi(+\Delta x/2, t)$  and  $\phi(-\Delta x/2, t)$  in (3.40), to obtain

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} - \frac{c}{24\Delta x} & [(\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n) - (\phi_{i-2}^n - 2\phi_{i-1}^n + \phi_i^n)] \\ & + \frac{c}{\Delta x} \left[ \frac{1}{2}(\phi_i^n + \phi_{i+1}^n) - \frac{1}{8}(\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n) \right] - \\ & \frac{c}{\Delta x} \left[ \frac{1}{2}(\phi_{i-1}^n + \phi_i^n) - \frac{1}{8}(\phi_{i-2}^n - 2\phi_{i-1}^n + \phi_i^n) \right] = 0, \end{aligned}$$

or

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + \frac{c}{\Delta x} & \left[ \frac{1}{2}(\phi_i^n + \phi_{i+1}^n) - \frac{1}{6}(\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n) \right] - \\ & \frac{c}{\Delta x} \left[ \frac{1}{2}(\phi_{i-1}^n + \phi_i^n) - \frac{1}{6}(\phi_{i-2}^n - 2\phi_{i-1}^n + \phi_i^n) \right] = 0. \end{aligned}$$

We can simplify:

$$\begin{aligned} \phi_i^{n+1} - \phi_i^n &= -\frac{c\Delta t}{\Delta x} \left\{ \left[ \frac{1}{2}(\phi_i^n + \phi_{i+1}^n) - \frac{1}{6}(\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n) \right] - \left[ \frac{1}{2}(\phi_{i-1}^n + \phi_i^n) - \frac{1}{6}(\phi_{i-2}^n - 2\phi_{i-1}^n + \phi_i^n) \right] \right\} \\ &= -\text{Co} \left\{ \frac{1}{2}(\phi_{i+1}^n - \phi_{i-1}^n) - \frac{1}{6}(\phi_{i-1}^n - 2\phi_i^n + \phi_{i+1}^n - \phi_{i-2}^n + 2\phi_{i-1}^n - \phi_i^n) \right\} \\ &= -\text{Co} \left\{ \frac{1}{2}(\phi_{i+1}^n - \phi_{i-1}^n) - \frac{1}{6}(-\phi_{i-2}^n + 3\phi_{i-1}^n - 3\phi_i^n + \phi_{i+1}^n) \right\} \\ &= -\frac{\text{Co}}{6} (\phi_{i-2}^n - 6\phi_{i-1}^n + 3\phi_i^n + 2\phi_{i+1}^n), \end{aligned}$$

or

$$\phi_i^{n+1} = \phi_i^n - \frac{\text{Co}}{6} (\phi_{i-2}^n - 6\phi_{i-1}^n + 3\phi_i^n + 2\phi_{i+1}^n), \quad (3.41)$$

which is the QUICKEST scheme for pure advection.

**Exemplo 3.1** Dada a equação de difusão-advecção

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

onde  $u$  e  $D$  são constantes, faça uma análise de estabilidade de von Neumann para o esquema explícito a seguir:

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + u \frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x} = D \frac{\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{\Delta x^2}.$$

Encontre uma relação entre os números de Courant,  $\text{Co} = u\Delta t/\Delta x$ , e de Fourier,  $\text{Fo} = D\Delta t/\Delta x^2$ , da forma

$$(a + b\text{Fo})^2 + (c + d\text{Co})^2 \leq 1;$$

que garanta a estabilidade do esquema.

SOLUÇÃO

$$\begin{aligned} \epsilon_i^{n+1} - \epsilon_i^n + \frac{u\Delta t}{2\Delta x} (\epsilon_{i+1}^n - \epsilon_{i-1}^n) &= \frac{D\Delta t}{\Delta x^2} (\epsilon_{i+1}^n - 2\epsilon_i^n + \epsilon_{i-1}^n), \\ \epsilon_i^{n+1} &= \epsilon_i^n - \frac{\text{Co}}{2} (\epsilon_{i+1}^n - \epsilon_{i-1}^n) + \text{Fo} (\epsilon_{i+1}^n - 2\epsilon_i^n + \epsilon_{i-1}^n), \\ \xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} - \frac{\text{Co}}{2} \left( \xi_l e^{at_n} e^{ik_l (i+1) \Delta x} - \xi_l e^{at_n} e^{ik_l (i-1) \Delta x} \right) + \end{aligned}$$

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

Faça  $\theta = k_l \Delta x / 2$ ; a condição para que o esquema seja estável é que

$$\begin{aligned}
|e^{a\Delta t}|^2 &< 1, \\
(1 - 4\text{Fo} \sin^2(\theta))^2 + (2\text{Co} \sin(\theta) \cos(\theta))^2 &\leq 1, \\
1 - 8\text{Fo} \sin^2(\theta) + 16\text{Fo}^2 \sin^4(\theta) + 4\text{Co}^2 \sin^2(\theta) \cos^2(\theta) &\leq 1, \\
-8\text{Fo} \sin^2(\theta) + 16\text{Fo}^2 \sin^4(\theta) + 4\text{Co}^2 \sin^2(\theta) \cos^2(\theta) &\leq 0.
\end{aligned} \tag{3.42}$$

Exceto para o caso  $\sin(\theta) \neq 0$ , podemos eliminar o fator comum  $\sin^2(\theta)$ :

$$\begin{aligned}
-8\text{Fo} + 16 \sin^2(\theta) \text{Fo}^2 + 4 \cos^2(\theta) \text{Co}^2 &\leq 0, \\
-2\text{Fo} + 4 \sin^2(\theta) \text{Fo}^2 + \cos^2(\theta) \text{Co}^2 &\leq 0,
\end{aligned} \tag{3.43}$$

onde

$$\text{Co} \leq \left[ \frac{2\text{Fo} (1 + 2 \sin^2(\theta) \text{Fo})}{\cos^2(\theta)} \right]^{1/2}. \tag{3.44}$$

Para cada  $\theta$ , (3.44) define uma região *diferente* do plano  $\text{Fo} \times \text{Co}$ . É preciso portanto *varrer* os valores de  $\theta$  e encontrar a região do plano em que o esquema é estável independentemente de  $\theta$  (lembre-se de que  $\theta$  indica o modo de Fourier que se desestabilizará; basta que um desses modos se desestabilize para que o sistema seja instável). As funções  $\sin^2(\theta)$  e  $\cos^2(\theta)$  possuem período igual a  $\pi$  (figura 3.7). No entanto, a função definida em (3.44) é simétrica em relação a  $\theta = \pi/2$ , como mostra a figura 3.8 (para  $\text{Fo} = 1$ ); portanto, basta “varrer”  $\theta$  desde 0 até  $\pi/2$ .

Antes de fazer essa varredura com “força bruta”, entretanto, vale a pena entender as curvas representadas por 3.43 tanto em geral, quanto para os casos particulares  $\theta = 0$  (entendido como um limite, já que (3.43) foi obtida após a divisão por  $\sin^2(\theta)$ ) e  $\theta = \pi/2$ , que definem a faixa de valores que devemos varrer.

Retornamos portanto à geometria analítica, e utilizamos  $x$  para  $\text{Fo}$  e  $y$  para  $\text{Co}$ . No limite da igualdade, manipulamos agora (3.42) da seguinte forma:

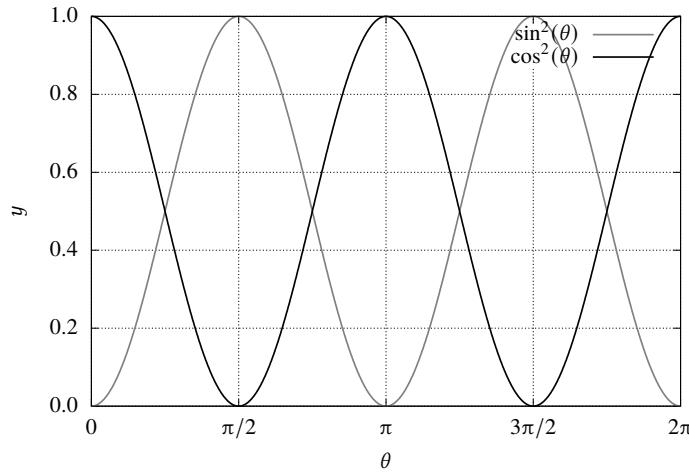
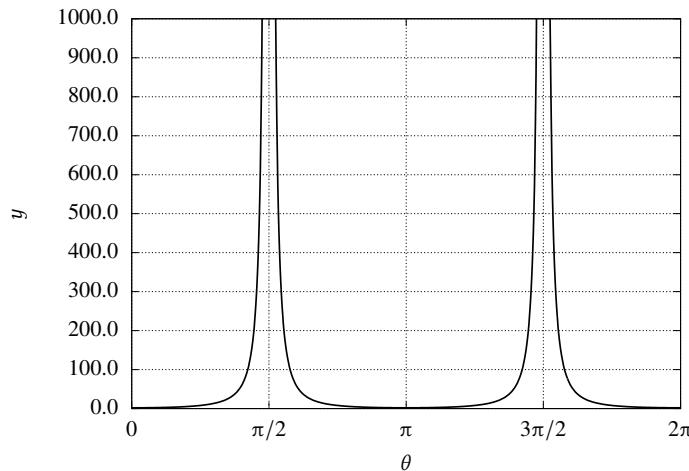
$$\begin{aligned}
(1 - 4 \sin^2(\theta)x)^2 + (\sin(2\theta)y)^2 &= 1, \\
(4 \sin^2(\theta))^2 \left( \frac{1}{4 \sin^2(\theta)} - x \right)^2 + (\sin(2\theta)y)^2 &= 1, \\
(4 \sin^2(\theta))^2 \left( x - \frac{1}{4 \sin^2(\theta)} \right)^2 + (\sin(2\theta)y)^2 &= 1, \\
\frac{\left( x - \frac{1}{4 \sin^2(\theta)} \right)^2}{\frac{1}{(4 \sin^2(\theta))^2}} + \frac{y^2}{\frac{1}{(\sin(2\theta))^2}} &= 1.
\end{aligned}$$

A equação acima tem a forma

$$\frac{(x - a)^2}{a^2} + \frac{y^2}{b^2} = 1,$$

ou seja: trata-se (em geral) da equação de uma elipse com semi-eixos  $a$  e  $b$ , e centrada em  $(a, 0)$ . Para cada  $a(\theta)$ ,  $b(\theta)$ , a região de estabilidade é a meia elipse com  $0 \leq x \leq 2a$  e  $y \geq 0$ . Concluímos que a região de estabilidade que procuramos (para todos os valores de  $\theta$  entre 0 e  $\pi/2$ ) é a interseção de todas as meias elipses correspondentes. Uma considerável introversão pode ser obtida com os dois limites. Para  $\theta = 0$ , (3.43) simplifica-se para

$$-2\text{Fo} + \text{Co}^2 \leq 0,$$

Figura 3.7: As funções  $\sin^2(\theta)$  e  $\cos^2(\theta)$ Figura 3.8: A função definida em 3.44 (para  $Fo = 1$ )

$$Co \leq \sqrt{2Fo}. \quad (3.45)$$

Esta é uma região não mais sob uma elipse, mas sim sob uma parábola. Para  $\theta = \pi/2$ , (3.43) simplifica-se para

$$\begin{aligned} -2Fo + 4Fo^2 &\leq 0, \\ -Fo + 2Fo^2 &\leq 0, \\ Fo(-1 + 2Fo) &\leq 0, \\ Fo &\leq \frac{1}{2}. \end{aligned} \quad (3.46)$$

Voltamos agora para uma varredura sistemática: a figura 3.9 mostra as meias-elipses que definem a região de estabilidade para cada  $\theta$ , em incrementos  $\Delta\theta = \pi/32$  a partir de  $\theta = \pi/32$ , até  $\theta = 15\pi/32$ , utilizando (3.44). Para a primeira metade dessa faixa de valores, ou seja, até  $\theta = 7\pi/32$ ,  $a(\theta) > 1/2$ . Nessa faixa, as ordenadas máximas das meias elipses (mostradas em cinza-claro) caem abaixo da parábola  $Co = \sqrt{2Fo}$  (mostrada com uma linha preta grossa). No valor central  $\theta = \pi/4$ ,  $a = 1/2$  e  $b = 1$  (confira), e a ordenada máxima da elipse,  $(1/2, 1)$ , também é um ponto da parábola (confira). Para a segunda metade da faixa,  $\pi/4 < \theta < \pi/2$ , e as ordenadas máximas das elipses agora ultrapassam a parábola. Os valores de  $a$  variam entre  $a = 1/2$  para  $\theta = \pi/4$  e  $a = 1/4$  para  $\theta = \pi/2$ . Note que este último valor também é um caso degenerado, e não representa mais uma elipse legítima. A interseção de todas as meias elipses é, claramente, a região em cinza-escuro da figura 3.9, representada por

$$Co \leq \sqrt{2Fo}, \quad 0 \leq Fo \leq 1/2 \blacksquare$$

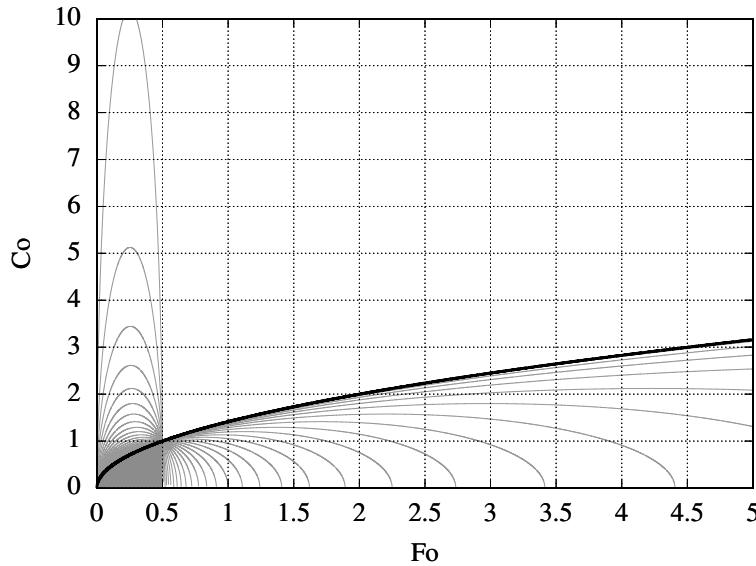


Figura 3.9: Exemplo 3.1: interseção (em cinza escuro) de 32 regiões (em cinza-claro; incrementos de  $\pi/32$ ) na equação (3.44) A parábola em preto representa o caso degenerado  $\theta = 0$ .

### Exercícios Propostos

**3.1** Escreva o programa `onda1d-upw` e `surfa1d-upw`, que implementam o esquema *upwind*. Reproduza a figura 3.4.

**3.2** Seja o esquema de diferenças finitas *upwind* explícito e condicionalmente estável para a equação da onda:

$$u_i^{n+1} = u_i^n - Co[u_i^n - u_{i-1}^n].$$

Considere que a matriz `u` foi alocada com `u = zeros((2,nx+1),float)`, onde `zeros` foi importada de `numpy`, com `nx=1000`, e que você está calculando `u[new]` a partir de `u[old]`, sendo que `old` refere-se ao passo de tempo  $n$ , e `new` ao passo de tempo  $n+1$ . Mostre como, utilizando a técnica de *slicing*, você pode calcular `u[new,1:nx]` em apenas uma linha de código em Python (usando `numpy`); suponha que a variável `Cou`, com o número de Courant, já foi calculada e que ela garante a estabilidade do esquema.

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

**3.4** Dado esquema explícito

$$u_i^{n+1} = 2 [1 - Co^2] u_i^n + Co^2 [u_{i+1}^n + u_{i-1}^n] - u_i^{n-1},$$

onde  $Co$  é o número de Courant, analise a estabilidade do esquema.

**3.5** Considere a seguinte discretização de

$$\frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0$$

(onde  $c > 0$  é constante) ( $t_n = n\Delta t$ ;  $x_i = i\Delta x$ ):

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

Faça uma análise completa de estabilidade de von Neumann do esquema em função do número de Courant  $Co = (c\Delta t)/\Delta x$ . Descubra se o esquema é incondicionalmente instável, condicionalmente estável, ou incondicionalmente estável. Se o esquema for condicionalmente estável, para que valores de  $Co$  ele é estável?

**3.6 Um esquema regressivo (*upwind*) de ordem 2.** Expanda em série de Taylor  $u(x, t)$  desde  $x_i$  até  $x_{i-1}$  e  $x_{i-2}$  (igualmente espaçados), elimine  $\partial^2 u / \partial x^2$  e encontre uma aproximação de diferenças finitas para  $\partial u / \partial x|_{x_i}$  cujo erro é  $O(\Delta x^2)$ .

3.7 Utilizando o esquema *upwind* do Exercício 3.6, podemos discretizar a equação da onda cinemática como

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -c \frac{3u_i^n - 4u_{i-1}^n + u_{i-2}^n}{2\Delta x}.$$

Analise a estabilidade desse esquema.

### 3.2 – Difusão pura

Considere agora a equação da difusão,

$$\frac{\partial \phi}{\partial t} = D \frac{\partial^2 \phi}{\partial x^2}, \quad (3.47)$$

com condições iniciais e de contorno

$$\phi(x, 0) = f(x) \quad (3.48)$$

$$\phi(0, t) = \phi(L, t) = 0. \quad (3.49)$$

A solução analítica é

$$\phi(x, t) = \sum_{n=1}^{\infty} A_n e^{-\frac{n^2 \pi^2 D}{L^2} t} \sin \frac{n \pi x}{L}, \quad (3.50)$$

$$A_n = \frac{2}{L} \int_0^L f(x) \sin \frac{n \pi x}{L} dx. \quad (3.51)$$

Em particular, se

$$D = 2,$$

$$L = 1,$$

$$f(x) = 2x(1-x),$$

$$A_n = 2 \int_0^1 2x(1-x) \sin(n \pi x) dx = \frac{8}{\pi^3 n^3} [1 - (-1)^n].$$

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

$$A_{2n+1} = \frac{16}{\pi^3 (2n+1)^3},$$

$$\phi(x, t) = \sum_{n=0}^{\infty} \frac{16}{\pi^3 (2n+1)^3} e^{-((2n+1)^2 \pi^2 D)t} \sin((2n+1)\pi x) \quad (3.52)$$

O programa `difusao1d-ana.chpl`, mostrado na listagem 3.8, 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 3.9, seleciona alguns instantes de tempo da solução analítica para visualização:

[ `divisao1d-ana -m=3 -n=100` ] .

A figura 3.10 mostra o resultado da solução numérica para  $t = 0$ ,  $t = 0,05$ ,  $t = 0,10$  e  $t = 0,15$ . Esse é praticamente o “fim” do processo difusivo, com a solução analítica tendendo rapidamente para zero.

---

Listagem 3.8: difusao1d-ana.chpl – Solução analítica da equação da difusão

---

```

1 // -----
2 // difusao1d-ana: solução analítica de
3 //
4 // du/dt = D du^2/dx^2
5 //
6 // u(x,0) = 2x(1-x)
7 // u(0,t) = 0
8 // u(1,t) = 0
9 //
10 // uso: ./difusao1d-ana
11 // -----
12 use IO only openWriter, binarySerializer;
13 use Math only exp,pi,sin;
14 const fou = openWriter("difusao1d-ana.dat",
15                         serializer = new binarySerializer(),locking=false);
16 const dx = 0.001;
17 const dt = 0.0005;
18 writef("// dx = %9.4dr\n",dx);
19 writef("// dt = %9.4dr\n",dt);
20 const nx = round(1.0/dx):int;      // número de pontos em x
21 const nt = round(1.0/dt):int;      // número de pontos em t
22 writef("// nx = %9i\n",nx);
23 writef("// nt = %9i\n",nt);
24 const epsilon = 1.0e-6;            // precisão da solução analítica
25 const D = 2.0;                   // difusividade
26 const dpiq = D*pi*pi;           // pi^2 D
27 const dzpic = 16/(pi*pi*pi);    // 16/pi^3
28 var phi: [0..nx] real;          // um array para conter a solução
29 for n in 0..nt do {             // loop no tempo
30     var t = n*dt;
31     if n % 100 == 0 then {
32         writeln(t);
33     }
34     forall i in 0..nx do {       // loop no espaço
35         var xi = i*dx;
36         phi[i] = ana(xi,t);
37     }
38     fou.write(phi);            // imprime uma linha com os novos dados
39 }
40 fou.close();
41 inline proc ana()              // solução analítica
42     const in x:real,
43     const in t:real
44     ): real {
45     var s = 0.0;
46     var ds = epsilon;
47     var n = 0;
48     while abs(ds) >= epsilon do {
49         var dnm1 = 2*n + 1;        // (2n+1)
50         var dnm1q = dnm1*dnm1;    // (2n+1)^2
51         var dnm1c = dnm1q*dnm1;  // (2n+1)^3
52         ds = exp(-dnm1q*dpiq*t);
53         ds *= sin(dnm1*pi*x) ;
54         ds /= dnm1c;
55         s += ds;
56         n += 1;
57     }
58     return s*dzpic;
59 }
```

---

Listagem 3.9: `divisaold-ana.py` – Seleciona alguns instantes de tempo da solução analítica para visualização

---

```

1 // -----
2 // divisao1d-ana.py: imprime em <arg> <m>+1 saídas de difusaold-ana a cada <n>
3 // intervalos de tempo
4 //
5 // uso: ./divisaold-ana.py --m=<m> --n=<n>
6 // -----
7 const dx = 0.001;
8 const dt = 0.0005;
9 writef("// dx = %9.4dr\n",dx);
10 writef("// dt = %9.4dr\n",dt);
11 const nx = round(1.0/dx):int;           // número de pontos em x
12 writef("// nx = %9i\n",nx);
13 config const m = 3;
14 config const n = 100;                   // a cada n intervalos de tempo
15 writef("// m = %9i\n",m);
16 writef("// n = %9i\n",n);
17 use IO only openReader, openWriter, binaryDeserializer;
18 const fin = openReader("difusaold-ana.dat",
19                         deserializer = new binaryDeserializer(), locking=false);
20 var phi: [0..nx] real;                 // uma linha de solução
21 var v: [0..m,0..nx] real;              // m+1 linhas de solução
22 fin.read(phi);                      // lê a condição inicial
23 v[0,0..nx] = phi;                   // inicializa a lista da "transposta"
24 for it in 1..m do {                  // para <m> instantes:
25     for ir in 1..n do {              // lê <ir> vezes, só guarda a última
26         fin.read(phi);
27     }
28     v[it,0..nx] = phi;             // guarda a última
29 }
30 // abre o arquivo de saída
31 const fou = openWriter("divisaold-ana.dat",locking=false);
32 for i in 0..nx do {
33     fou.writef("%10.6dr", (i*dx)); // escreve o "x"
34     fou.writef("%10.6dr", v[0,i]); // escreve a cond inicial
35     for k in 1..m do {
36         fou.writef("%10.6dr", v[k,i]); // escreve o k-ésimo
37     }
38     fou.writef("\n");
39 }
40 fou.close();

```

---

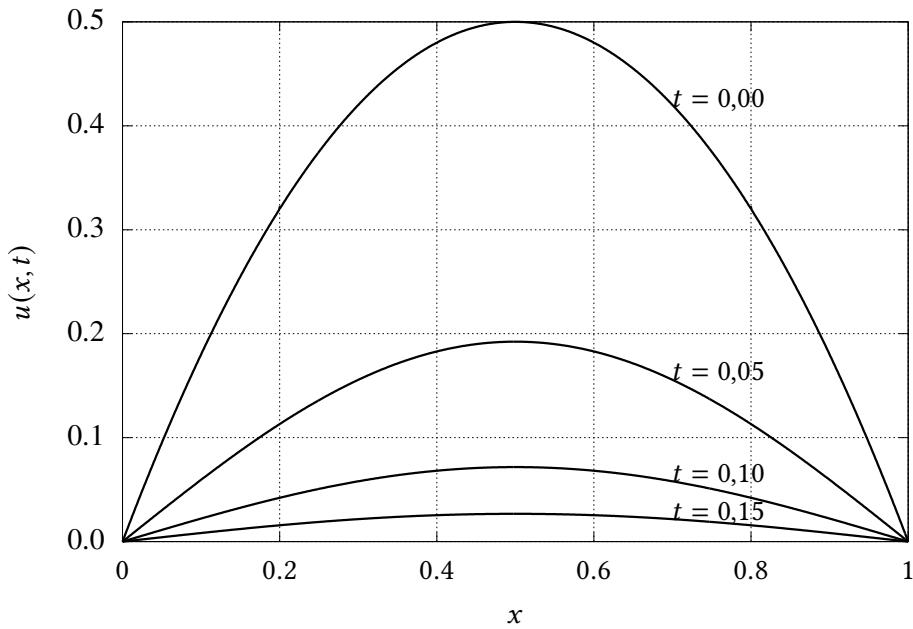


Figura 3.10: Solução analítica da equação de difusão para  $t = 0$ ,  $t = 0,05$ ,  $t = 0,10$  e  $t = 0,15$ .

**Esquema explícito** Talvez o esquema explícito mais óbvio para discretizar (3.47) seja

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} = D \frac{\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{\Delta x^2}. \quad (3.53)$$

A derivada parcial em relação ao tempo é de  $\mathcal{O}(\Delta t)$ , enquanto que a derivada segunda parcial em relação ao espaço é, como vimos em (3.25), de  $\mathcal{O}(\Delta x^2)$ . Mas não nos preocupemos muito, ainda, com a acurácia do esquema numérico. Nossa primeira preocupação, como você já sabe, é outra: o esquema (3.53) é *estável*?

Explicitamos  $\phi_i^{n+1}$  em (3.53):

$$\phi_i^{n+1} = \phi_i^n + \text{Fo} [\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n], \quad (3.54)$$

onde

$$\text{Fo} = \frac{D\Delta t}{\Delta x^2} \quad (3.55)$$

é o *número de Fourier de grade* (? , capítulo 3). A análise de estabilidade de von Neumann agora produz

$$\begin{aligned} \xi_l e^{a(t_n+\Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} + \\ &\quad \text{Fo} [\xi_l e^{at_n} e^{ik_l (i+1) \Delta x} - 2\xi_l e^{at_n} e^{ik_l i \Delta x} + \xi_l e^{at_n} e^{ik_l (i-1) \Delta x}], \\ e^{a\Delta t} &= 1 + \text{Fo} [e^{+ik_l \Delta x} - 2 + e^{-ik_l \Delta x}] \\ &= 1 + 2\text{Fo} [\cos(k_l \Delta x) - 1] \\ &= 1 - 4\text{Fo} \sin^2\left(\frac{k_l \Delta x}{2}\right) \end{aligned} \quad (3.56)$$

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

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

ou

$$-8\text{Fo} \sin^2\left(\frac{k_l \Delta x}{2}\right) + 16\text{Fo}^2 \sin^4\left(\frac{k_l \Delta x}{2}\right) \leq 0,$$

$$8\text{Fo} \sin^2\left(\frac{k_l \Delta x}{2}\right) \left[-1 + 2\text{Fo} \sin^2\left(\frac{k_l \Delta x}{2}\right)\right] \leq 0,$$

$$\text{Fo} \leq \frac{1}{2}. \quad (3.57)$$

Podemos agora calcular o número de Fourier que utilizamos para plotar a solução analítica (verifique nas listagens 3.8 e 3.9):

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

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

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

Nós esperamos que nosso esquema explícito agora rode muito lentamente. Mas vamos implementá-lo. O programa que implementa o esquema é o `diffusao1d-exp.chpl`, mostrado na listagem 3.10.

O programa `divisao1d-exp.chpl`, mostrado na listagem 3.11, seleciona alguns instantes de tempo da solução analítica para visualização:

```
[ ./divisao1d-exp --m=3 --n=5000 ].
```

O resultado da solução numérica com o método explícito está mostrado na figura 3.11: ele é impressionantemente bom, embora seja computacionalmente muito caro. A escolha judiciosa de  $\Delta t$  e  $\Delta x$  para obedecer ao critério (3.57) 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 3.10: difusao1d-exp.py – Solução numérica da equação da difusão: método explícito.

---

```

1 // -----
2 // difusao1d-exp resolve uma equação de difusão com um método explícito
3 //
4 // uso: ./difusao1d-exp
5 //
6 use IO only openWriter, binarySerializer;
7 const fou = openWriter("difusao1d-exp.dat",
8     serializer = new binarySerializer(), locking=false);
9 const dx = 0.01;
10 const dt = 0.00001;
11 writef("// dx = %9.4dr\n",dx);
12 writef("// dt = %9.4dr\n",dt);
13 const nx = round(1.0/dx):int;           // número de pontos em x
14 const nt = round(1.0/dt):int;           // número de pontos em t
15 writef("// nx = %9i\n",nx);
16 writef("// nt = %9i\n",nt);
17 var phi: [0..1,0..nx] real;            // apenas 2 posições no tempo
18                                         // são necessárias!
19 for i in 0..nx do {                  // monta a condição inicial
20     var xi = i*dx;
21     phi[0,i] = CI(xi);
22 }
23 fou.write(phi[0..nx]);                // imprime a condição inicial
24 var iold = 0;
25 var inew = 1;
26 const D = 2.0;                      // difusividade
27 const Fon = D*dt/((dx)**2);          // número de Fourier
28 writef("Fo = %10.6dr\n",Fon);
29 for n in 1..nt do {                 // loop no tempo a partir de 1
30     writeln(n);
31     for i in 1..nx-1 do {            // loop no espaço
32         phi[inew,i] = phi[iold,i] +
33             Fon*(phi[iold,i+1] - 2*phi[iold,i] + phi[iold,i-1]);
34     }
35     phi[inew,0] = 0.0;              // condição de contorno, x = 0
36     phi[inew,nx] = 0.0;             // condição de contorno, x = 1
37     fou.write(phi[inew,0..nx]);      // imprime uma linha com os novos dados
38     iold <=> inew;                // troca os índices
39 }
40 fou.close();                         // define a condição inicial
41 proc CI()                           // define a condição inicial
42     const in x: real
43     ):real {
44     if 0 <= x && x <= 1.0 then {
45         return 2.0*x*(1.0-x);
46     }
47     else {
48         return 0.0;
49     }
50 }
```

---

Listagem 3.11: `divisaoid-exp.chpl` – Seleciona alguns instantes de tempo da solução analítica para visualização

---

```

1 // -----
2 // divisaoid-exp.py: imprime em <arq> <m>+1 saídas de difusaoid-exp a cada <n>
3 // intervalos de tempo
4 //
5 // uso: ./divisaoid-exp.py --m=<m> --n=<n>
6 // -----
7 const dx = 0.01;
8 const dt = 0.00001;
9 writef("// dx = %9.4dr\n",dx);
10 writef("// dt = %9.4dr\n",dt);
11 const nx = round(1.0/dx):int;      // número de pontos em x
12 const nt = round(1.0/dt):int;      // número de pontos em t
13 writef("// nx = %9i\n",nx);
14 config const m = 3;                // m saídas
15 config const n = 5000;              // a cada n intervalos de tempo
16 writef("// m = %9i\n",m);
17 writef("// n = %9i\n",n);
18 use IO only openReader, openWriter, binaryDeserializer;
19 const fin = openReader("difusaoid-exp.dat",
20                         deserializer = new binaryDeserializer(), locking=false);
21 var phi: [0..nx] real;             // uma linha de solução
22 var v: [0..m,0..nx] real;          // m+1 linhas de solução
23 fin.read(phi);                   // lê a condição inicial
24 v[0,0..nx] = phi;                // inicializa a lista da "transposta"
25 for it in 1..m do {               // para <m> instantes:
26     for ir in 1..n do {           // lê <n> vezes, só guarda a última
27         fin.read(phi);
28     }
29     v[it,0..nx] = phi;           // guarda a última
30 }
31 // abre o arquivo de saída
32 const founam = "divisaoid-exp.dat";
33 const fou = openWriter(founam, locking=true);
34 for i in 0..nx do {
35     fou.writef("%10.6dr", (i*dx)); // escreve o "x"
36     fou.writef("%10.6dr", v[0,i]); // escreve a cond inicial
37     for k in 1..m do {
38         fou.writef("%10.6dr", v[k,i]); // escreve o k-ésimo
39     }
40     fou.writef("\n");
41 }
42 fou.close();

```

---

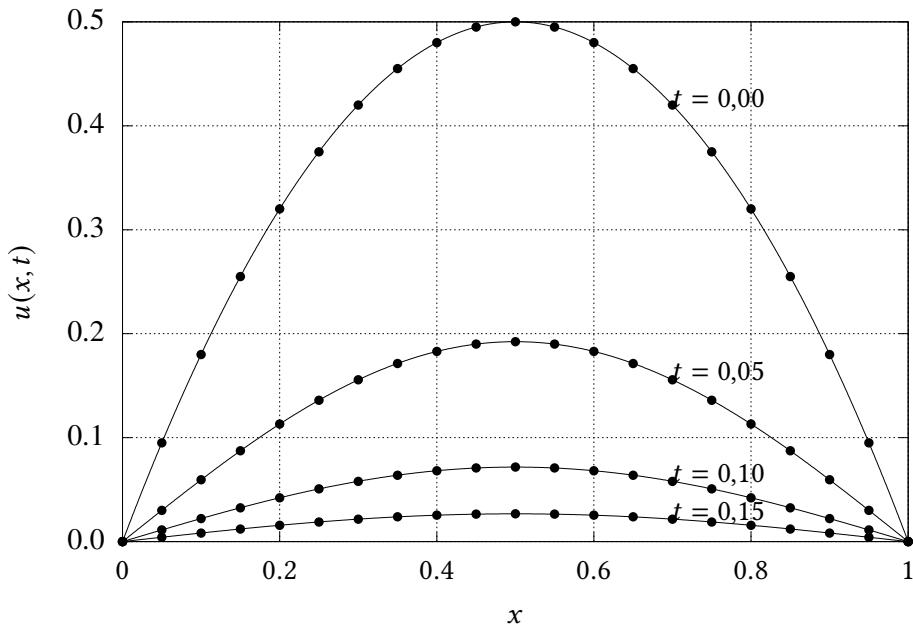


Figura 3.11: Solução numérica com o método explícito (3.54) (círculos) versus a solução analítica (linha cheia) da equação de difusão para  $t = 0$ ,  $t = 0,05$ ,  $t = 0,10$  e  $t = 0,15$ . Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

**Esquemas implícitos** Embora o esquema explícito que nós utilizamos acima seja acurado, ele é *lento*. Embora nossos computadores estejam ficando a cada dia mais rápidos, isso não é desculpa para utilizar mal nossos recursos computacionais. Vamos portanto fazer uma mudança fundamental nos nossos esquemas de diferenças finitas: vamos calcular a derivada espacial no instante  $n + 1$ :

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= D \frac{\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}}{\Delta x^2}, \\ \phi_i^{n+1} - \phi_i^n &= \text{Fo}(\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}), \\ -\text{Fo}\phi_{i-1}^{n+1} + (1 + 2\text{Fo})\phi_i^{n+1} - \text{Fo}\phi_{i+1}^{n+1} &= \phi_i^n. \end{aligned} \quad (3.58)$$

Reveja a discretização (3.5)–(3.9): para  $i = 1, \dots, N_x - 1$ , (3.58) acopla 3 valores das incógnitas  $\phi^{n+1}$  no instante  $n + 1$ . Quando  $i = 0$ , e quando  $i = N_x$ , não podemos utilizar (3.58), porque não existem os índices  $i = -1$ , e  $i = N_x + 1$ . Quando  $i = 1$  e  $i = N_x - 1$ , (3.58) precisa ser modificada, para a introdução das *condições de contorno*: como  $\phi_0^n = 0$  e  $\phi_{N_x}^n = 0$  para qualquer  $n$ , teremos

$$(1 + 2\text{Fo})\phi_1^{n+1} - \text{Fo}\phi_2^{n+1} = \phi_1^n, \quad (3.59)$$

$$-\text{Fo}\phi_{N_x-2}^{n+1} + (1 + 2\text{Fo})\phi_{N_x-1}^{n+1} = \phi_{N_x-1}^n. \quad (3.60)$$

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

$$\left[ \begin{array}{cccccc} 1 + 2\text{Fo} & -\text{Fo} & 0 & \dots & 0 & 0 \\ -\text{Fo} & 1 + 2\text{Fo} & -\text{Fo} & 0 & \dots & 0 \\ \vdots & & & & & \vdots \\ 0 & \dots & 0 & -\text{Fo} & 1 + 2\text{Fo} & -\text{Fo} \\ 0 & 0 & \dots & 0 & -\text{Fo} & 1 + 2\text{Fo} \end{array} \right] \begin{bmatrix} \phi_1^{n+1} \\ \phi_2^{n+1} \\ \vdots \\ \phi_{N_x-2}^{n+1} \\ \phi_{N_x-1}^{n+1} \end{bmatrix} = \begin{bmatrix} \phi_1^n \\ \phi_2^n \\ \vdots \\ \phi_{N_x-2}^n \\ \phi_{N_x-1}^n \end{bmatrix}. \quad (3.61)$$

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

$$\epsilon_i^{n+1} = \epsilon_i^n + \text{Fo}(\epsilon_{i+1}^{n+1} - 2\epsilon_i^{n+1} + \epsilon_{i-1}^{n+1})$$

$$\begin{aligned}
\xi_l e^{a(t_n + \Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} \\
&\quad + \text{Fo} \left( \xi_l e^{a(t_n + \Delta t)} e^{ik_l(i+1)\Delta x} - 2\xi_l e^{a(t_n + \Delta t)} e^{ik_l i \Delta x} \right. \\
&\quad \left. + \xi_l e^{a(t_n + \Delta t)} e^{ik_l(i-1)\Delta x} \right), \\
e^{a\Delta t} &= 1 + e^{a\Delta t} \text{Fo} \left( e^{ik_l \Delta x} - 2 + e^{-ik_l \Delta x} \right), \\
e^{a\Delta t} &= 1 + e^{a\Delta t} 2\text{Fo} (\cos(k_l \Delta x) - 1), \\
e^{a\Delta t} &= 1 - e^{a\Delta t} 4\text{Fo} \sin^2 \left( \frac{k_l \Delta x}{2} \right), \\
e^{a\Delta t} \left[ 1 + 4\text{Fo} \sin^2 \left( \frac{k_l \Delta x}{2} \right) \right] &= 1, \\
|e^{a\Delta t}| &= \frac{1}{1 + 4\text{Fo} \sin^2 \left( \frac{k_l \Delta x}{2} \right)} \leq 1 \quad \text{sempre.} \tag{3.62}
\end{aligned}$$

Portanto, o esquema implícito (3.58) é incondicionalmente estável, e temos confiança de que o programa correspondente não se instabilizará.

Existem várias coisas atraentes para um programador em (3.61). Em primeiro lugar, a matriz do sistema é uma matriz banda tridiagonal; sistemas lineares com esse tipo de matriz são particularmente simples de resolver, e estão disponíveis na literatura (por exemplo: ?, 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. A solução de uma matriz banda tridiagonal já foi mostrada na listagem 2.13.

Em seguida, o programa `difusao1d-imp.py` resolve o problema com o método implícito. Ele está mostrado na listagem 3.12. A principal novidade está nas linhas 38–42, e depois novamente na linha 52. Em Chapel é possível especificar *sub-arrays* com um dispositivo denominado *slicing*, que torna a programação mais compacta e clara. Por exemplo, na linha 39, todos os elementos `A[2]...A[nx-1]` receberem o valor `-Fon`.

Existe um programa `divisao1d-imp.chpl`, 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 3.12

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 (3.47) é aproximada, no esquema implícito (3.58), por um esquema de  $O(\Delta x^2)$ . A derivada temporal, por sua vez, é apenas de  $O(\Delta t)$ . Mas é possível consertar isso! A idéia é substituir (3.58) por

$$\begin{aligned}
\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= \frac{D}{2} \left[ \frac{\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{\Delta x^2} + \frac{\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}}{\Delta x^2} \right], \\
\phi_i^{n+1} &= \phi_i^n + \frac{\text{Fo}}{2} [\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n + \phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}]. \tag{3.63}
\end{aligned}$$

Com essa 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 isso, 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 isso, fazemos

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

Listagem 3.12: difusao1d-imp.chpl – Solução numérica da equação da difusão: método implícito.

```

1 // -----
2 // difusao1d-imp resolve uma equação de difusão com um método implícito
3 //
4 // uso: ./difusao1d-imp
5 //
6 use IO only openWriter, binarySerializer;
7 const fou = openWriter("difusao1d-imp.dat",
8     serializer = new binarySerializer(), locking=false);
9 const dx = 0.01;                      // define a discretização em x
10 const dt = 0.001;                     // define a discretização em t
11 writef("// dx = %9.4dr\n",dx);
12 writef("// dt = %9.4dr\n",dt);
13 const nx = round(1.0/dx):int;         // número de pontos em x
14 const nt = round(1.0/dt):int;         // número de pontos em t
15 writef("// nx = %9i\n",nx);
16 writef("// nt = %9i\n",nt);
17 var phi: [0..1,0..nx] real;           // apenas 2 posições no tempo
18                                // são necessárias!
19 for i in 0..nx do {                 // monta a condição inicial
20     var xi = i*dx ;
21     phi[0,i] = CI(xi);
22 }
23 fou.write(phi[0,0..nx]);             // imprime a condição inicial
24 var iold = 0;
25 var inew = 1;
26 const D = 2.0;                      // difusividade
27 const Fon = D*dt/((dx)**2);          // número de Fourier
28 writef("Fo = %10.6dr\n",Fon);
29 var
30     A,                                // cria a matriz do sistema
31     B,                                // cria a matriz do sistema
32     C,                                // cria a matriz do sistema
33     : [1..nx-1] real;
34 //
35 // cuidado, "linha" e "coluna" abaixo não significam as reais linhas e colunas
36 // do sistema de equações, mas sim a forma de armazenar uma matriz tridiagonal
37 //
38 A[1] = 0.0;                         // zera A[1]
39 A[2..nx-1] = -Fon;                  // preenche a diagonal inferior
40 B[1..nx-1] = 1.0 + 2*Fon;           // preenche a diagonal
41 C[1..nx-2] = -Fon;                  // preenche a diagonal superior
42 C[nx-1] = 0.0;                     // zera C[nx-1]
43 //
44 // importa tridiag
45 //
46 use tridiag;
47 for n in 1..nt do {                // loop no tempo a partir de 1
48     writeln(n);
49 //
50 // atenção: calcula apenas os pontos internos de u!
51 //
52     tridiag(A,B,C,phi[iold,1..nx-1],phi[inew,1..nx-1]); // resolve o sistema
53     phi[inew,0] = 0.0;                   // condição de contorno, x = 0
54     phi[inew,nx] = 0.0;                 // condição de contorno, x = 1
55     fou.write(phi[inew,0..nx]);          // imprime uma linha com os novos dados
56     iold <=> inew;                   // troca os índices
57 }
58 fou.close();                        // fecha o arquivo de saída, e fim.
59 inline proc CI(                    // define a condição inicial
60     const in x: real
61     ): real {
62     if 0 <= x && x <= 1.0 then {
63         return 2.0*x*(1.0-x);
64     }
65     else {
66         return 0.0;
67     }

```

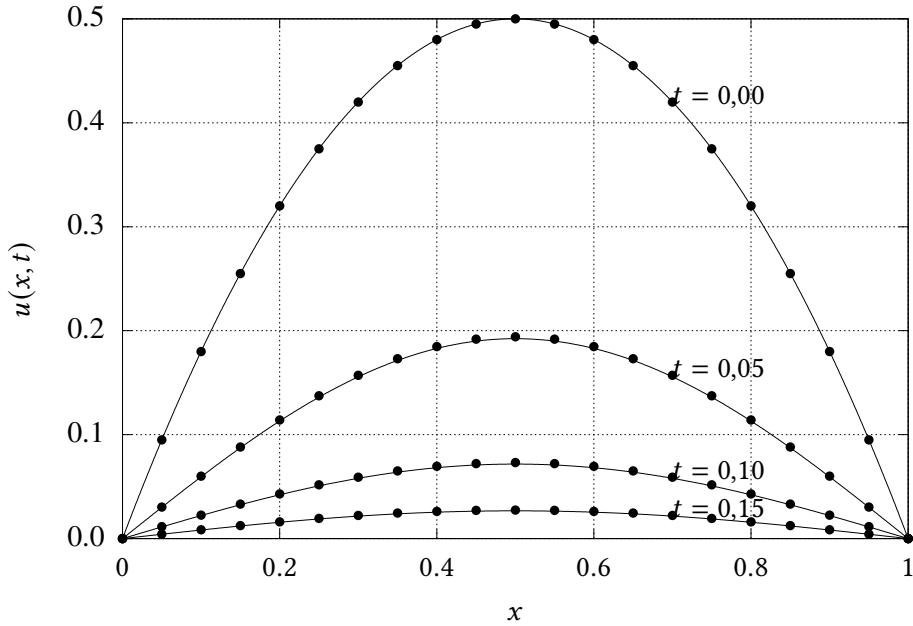


Figura 3.12: Solução numérica com o método implícito (3.58) (círculos) versus a solução analítica (linha cheia) da equação de difusão para  $t = 0$ ,  $t = 0,05$ ,  $t = 0,10$  e  $t = 0,15$ . Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

e substituímos um modo de Fourier:

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

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 (3.58):

$$-\frac{\text{Fo}}{2} u_{i-1}^{n+1} + (1 + \text{Fo}) u_i^{n+1} - \frac{\text{Fo}}{2} u_{i+1}^{n+1} = \frac{\text{Fo}}{2} u_{i-1}^n + (1 - \text{Fo}) u_i^n + \frac{\text{Fo}}{2} u_{i+1}^n \quad (3.64)$$

Para as condições de contorno de (3.49), as linhas correspondentes a  $i = 1$  e  $i = N_x - 1$  são

$$(1 + \text{Fo}) u_1^{n+1} - \frac{\text{Fo}}{2} u_2^{n+1} = (1 - \text{Fo}) u_1^n + \frac{\text{Fo}}{2} u_2^n, \quad (3.65)$$

$$-\frac{\text{Fo}}{2} u_{N_x-2}^{n+1} + (1 + \text{Fo}) u_{N_x-1}^{n+1} = \frac{\text{Fo}}{2} u_{N_x-2}^n + (1 - \text{Fo}) u_{N_x-1}^n \quad (3.66)$$

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.chpl`, é mostrado na listagem 3.13.

A grande novidade computacional de `difusao1d-ckn.py` é a linha 52: é possível escrever (3.64) *vetorialmente*: note que não há necessidade de fazer um *loop* em  $x$  para calcular cada elemento  $D[i]$  individualmente. O mesmo tipo de facilidade está disponível em FORTRAN90, FORTRAN95, etc.. Com isso, a implementação computacional dos cálculos gerada por Chapel (ou pelo compilador FORTRAN) também é potencialmente mais eficiente.

O método de Crank-Nicholson possui acurácia  $\mathcal{O}(\Delta t)^2$ , portanto ele deve ser capaz de dar passos ainda mais largos no tempo que o método implícito (3.58); 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”  $d$ ), e é mostrado na figura 3.13.

Listagem 3.13: `difusao1d-ckn.chpl` – Solução numérica da equação da difusão: esquema de Crank-Nicholson.

```

1 // -----
2 // difusao1d-ckn resolve uma equação de difusão com o método de Crank-Nicholson
3 //
4 // uso: ./difusao1d-ckn
5 // -----
6 use IO only openWriter, binarySerializer;
7 const fou = openWriter("difusao1d-ckn.dat",
8     serializer = new binarySerializer(), locking=false);
9 const dx = 0.01;                      // define a discretização em x
10 const dt = 0.005;                     // define a discretização em t
11 writef("// dx = %9.4dr\n",dx); //
12 writef("// dt = %9.4dr\n",dt);
13 const nx = round(1.0/dx):int;         // número de pontos em x
14 const nt = round(1.0/dt):int;         // número de pontos em t
15 writef("// nx = %9i\n",nx);
16 writef("// nt = %9i\n",nt);
17 var phi: [0..1,0..nx] real;           // apenas 2 posições no tempo
18                                // são necessárias!
19 for i in 0..nx do {                  // monta a condição inicial
20     var xi = i*dx;
21     phi[0,i] = CI(xi);
22 }
23 fou.write(phi[0..nx]);                // imprime a condição inicial
24 var iold = 0;
25 var inew = 1;
26 const Dif = 2.0;                    // difusividade
27 const Fon = Dif*dt/((dx)**2);        // número de Fourier
28 writef("Fo = %10.6dr\n",Fon);
29 var
30     A,                                // cria a matriz do sistema
31     B,                                // cria a matriz do sistema
32     C,                                // cria a matriz do sistema
33     D                                // CKN precisa de um forçante + complicado
34     : [1..nx-1] real;
35 // -----
36 // armazena a matriz do sistema em A, B, C.
37 // -----
38 A[1] = 0.0;                          // zera A[1]
39 A[2..nx-1] = -Fon/2.0;              // preenche a diagonal inferior
40 B[1..nx-1] = 1.0 + Fon;             // preenche a diagonal
41 C[1..nx-2] = -Fon/2.0;              // preenche a diagonal superior
42 C[nx-1] = 0.0;                     // zera C[nx-1]
43 // -----
44 // importa tridiag

```

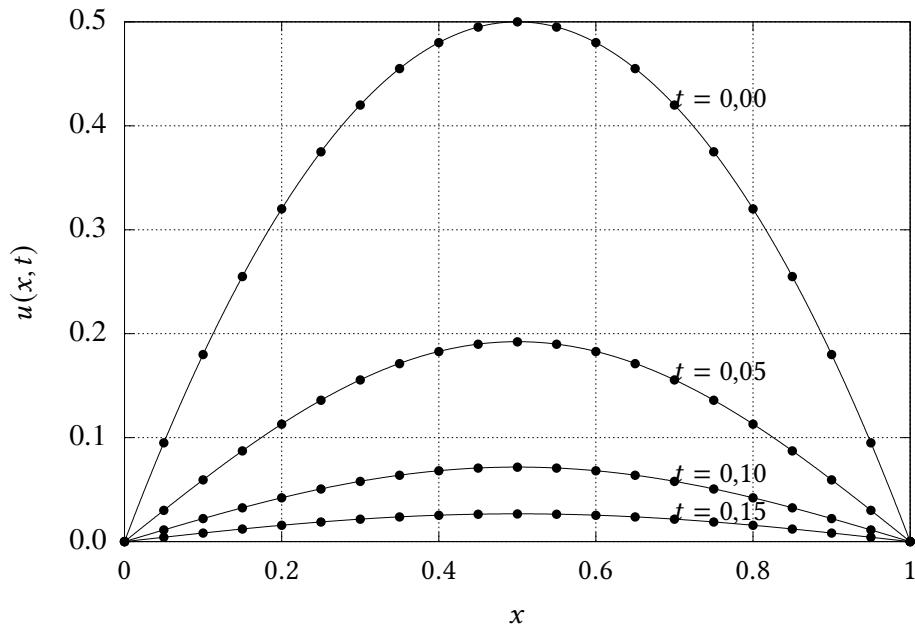


Figura 3.13: Solução numérica com o método de Crank-Nicholson ((3.64)) (círculos) *versus* a solução analítica (linha cheia) da equação de difusão para  $t = 0$ ,  $t = 0,05$ ,  $t = 0,10$  e  $t = 0,15$ . Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

```

45 // -----
46 use tridiag;
47 for n in 1..nt do {           // loop no tempo
48     writeln(n);
49 // -----
50 // recalcular o vetor de carga vetorialmente
51 // -----
52 D = (Fon/2)*phi[iold,0..nx-2] + (1 - Fon)*phi[iold,1..nx-1] +
53     (Fon/2)*phi[iold,2..nx];
54 // -----
55 // atenção: calcula apenas os pontos internos de phi!
56 // -----
57 tridiag(A,B,C,D,phi[inew,1..nx-1]);
58 phi[inew,0] = 0.0;             // condição de contorno, x = 0
59 phi[inew,nx] = 0.0;           // condição de contorno, x = 1
60 fou.write(phi[inew,0..nx]);   // imprime uma linha com os
61                                // novos dados
62 iold <=> inew;              // troca os índices
63 }
64 fou.close();                 // fecha o arquivo de saída, e
65 inline proc CI(             // define a condição inicial
66     const in x: real
67 ): real {
68     if 0 <= x && x <= 1.0 then {
69         return 2.0*x*(1.0-x);
70     }
71     else {
72         return 0.0;
73     }
74 }
```

**Exemplo 3.2** Dada a equação da difusão unidimensional

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}, \quad 0 \leq x \leq L,$$

e o esquema de discretização

$$\begin{aligned}\Delta x &= L/N_x, \\ x_i &= i\Delta x, \quad i = 0, \dots, N_x, \\ t_n &= n\Delta t, \\ u_i^n &= u(x_i, t_n), \\ \frac{1}{2\Delta t} [(u_{i+1}^{n+1} - u_{i+1}^n) + (u_{i-1}^{n+1} - u_{i-1}^n)] &= D \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2},\end{aligned}$$

obtenha o critério de estabilidade por meio de uma análise de estabilidade de von Neumann.

**SOLUÇÃO**

Suponha que a equação diferencial se aplique ao erro:

$$e_i^n = \sum_l \xi_l e^{at_n} e^{ik_l i \Delta x} \Rightarrow$$

Então

$$\begin{aligned}\frac{1}{2\Delta t} [(\xi_l e^{a(t_n+\Delta t)} e^{ik_l(i+1)\Delta x} - \xi_l e^{at_n} e^{ik_l(i+1)\Delta x}) + (\xi_l e^{a(t_n+\Delta t)} e^{ik_l(i-1)\Delta x} - \xi_l e^{at_n} e^{ik_l(i-1)\Delta x})] \\ = D \frac{\xi_l e^{at_n} e^{ik_l(i+1)\Delta x} - 2\xi_l e^{at_n} e^{ik_l i \Delta x} + \xi_l e^{at_n} e^{ik_l(i-1)\Delta x}}{\Delta x^2}; \\ \frac{1}{2\Delta t} [(\xi_l e^{a\Delta t} e^{ik_l(i+1)\Delta x} - \xi_l e^{ik_l(i+1)\Delta x}) + (\xi_l e^{a\Delta t} e^{ik_l(i-1)\Delta x} - \xi_l e^{ik_l(i-1)\Delta x})] = \\ D \frac{\xi_l e^{ik_l(i+1)\Delta x} - 2\xi_l e^{ik_l i \Delta x} + \xi_l e^{ik_l(i-1)\Delta x}}{\Delta x^2}; \\ \frac{1}{2\Delta t} [(e^{a\Delta t} e^{ik_l(i+1)\Delta x} - e^{ik_l(i+1)\Delta x}) + (e^{a\Delta t} e^{ik_l(i-1)\Delta x} - e^{ik_l(i-1)\Delta x})] = \\ D \frac{e^{ik_l(i+1)\Delta x} - 2 e^{ik_l i \Delta x} + e^{ik_l(i-1)\Delta x}}{\Delta x^2}; \\ [(e^{a\Delta t} e^{ik_l(i+1)\Delta x} - e^{ik_l(i+1)\Delta x}) + (e^{a\Delta t} e^{ik_l(i-1)\Delta x} - e^{ik_l(i-1)\Delta x})] = \\ \frac{2D\Delta t}{\Delta x^2} [e^{ik_l(i+1)\Delta x} - 2 e^{ik_l i \Delta x} + e^{ik_l(i-1)\Delta x}].\end{aligned}$$

Segue-se que

$$\begin{aligned}e^{a\Delta t} [e^{ik_l(i+1)\Delta x} + e^{ik_l(i-1)\Delta x}] &= e^{ik_l(i+1)\Delta x} + e^{ik_l(i-1)\Delta x} + \\ &\quad 2\text{Fo} [e^{ik_l(i+1)\Delta x} - 2 e^{ik_l i \Delta x} + e^{ik_l(i-1)\Delta x}] \\ e^{a\Delta t} [e^{ik_l\Delta x} + e^{-ik_l\Delta x}] &= e^{ik_l\Delta x} + e^{-ik_l\Delta x} + 2\text{Fo} [e^{ik_l\Delta x} - 2 + e^{-ik_l\Delta x}] \\ e^{a\Delta t} &= 1 + 2\text{Fo} \frac{2 \cos(k_l \Delta x) - 2}{2 \cos(k_l \Delta x)} \\ &= 1 + 2\text{Fo} \frac{\cos(k_l \Delta x) - 1}{\cos(k_l \Delta x)} \\ &= 1 - 4\text{Fo} \frac{\sin^2(k_l \Delta x / 2)}{\cos(k_l \Delta x)}.\end{aligned}$$

A função

$$f(x) = \frac{\sin^2(x/2)}{\cos(x)}$$

possui singularidades em  $\pi/2 + k\pi$ , e muda de sinal em torno destas singularidades: não é possível garantir que  $|e^{a\Delta t}| \leq 1$  uniformemente, e o esquema é incondicionalmente instável.

**Exemplo 3.3** Considere um esquema de diferenças finitas implícito “clássico” para a equação da difusão:

$$-\text{Fou}_{i-1}^{n+1} + (1 + 2\text{Fo})u_i^{n+1} - \text{Fou}_{i+1}^{n+1} = u_i^n, \quad i = 1, \dots, N_x - 1.$$

onde  $\text{Fo} = D\Delta t/\Delta x^2$ , e  $D$  é a difusividade. Sabemos que a equação acima em geral não vale para a primeira ( $i = 1$ ) e última ( $i = N_x - 1$ ) linhas. Obtenha essas linhas para as condições de contorno

$$\begin{aligned} u(0, t) &= \alpha, \\ \frac{\partial u(L, t)}{\partial x} &= \beta, \end{aligned}$$

sendo  $\alpha$  e  $\beta$  constantes, onde  $x = 0$  corresponde ao ponto de grade  $i = 0$ , e  $x = L$  corresponde ao ponto de grade  $i = N_x$ .

### SOLUÇÃO

A primeira linha fica

$$\begin{aligned} -\text{Fou}\alpha + (1 + 2\text{Fo})u_1^{n+1} - \text{Fou}_2^{n+1} &= u_1^n, \\ (1 + 2\text{Fo})u_1^{n+1} - \text{Fou}_2^n &= u_1^n + \text{Fou}\alpha. \end{aligned}$$

A aproximação da derivada em  $x = L$  é

$$\begin{aligned} \frac{\partial u(L, t)}{\partial x} &\approx \frac{u_{N_x}^{n+1} - u_{N_x-1}^{n+1}}{\Delta x} = \beta \Rightarrow \\ u_{N_x}^{n+1} - u_{N_x-1}^{n+1} &= \beta\Delta x, \\ u_{N_x}^{n+1} &= u_{N_x-1}^{n+1} + \beta\Delta x, \end{aligned}$$

de forma que a última linha fica

$$\begin{aligned} -\text{Fou}_{N_x-2}^{n+1} + (1 + 2\text{Fo})u_{N_x-1}^{n+1} - \text{Fou}_{N_x}^{n+1} &= u_{N_x-1}^n, \\ -\text{Fou}_{N_x-2}^{n+1} + (1 + 2\text{Fo})u_{N_x-1}^{n+1} - \text{Fou}_{N_x}^{n+1} &= u_{N_x-1}^n, \\ -\text{Fou}_{N_x-2}^{n+1} + (1 + 2\text{Fo})u_{N_x-1}^{n+1} - \text{Fou}_{N_x}^{n+1} &= u_{N_x-1}^n, \\ -\text{Fou}_{N_x-2}^{n+1} + (1 + 2\text{Fo})u_{N_x-1}^{n+1} &= u_{N_x-1}^n + \text{Fou}_{N_x}^{n+1}. \end{aligned} \blacksquare$$

**Exemplo 3.4** Considere a equação de advecção-difusão unidimensional

$$\frac{\partial \phi}{\partial t} + U \frac{\partial \phi}{\partial x} = D \frac{\partial^2 \phi}{\partial x^2} - K\phi, \quad 0 \leq x \leq \infty, \quad t > 0, \quad (3.67)$$

com condições iniciais e de contorno

$$\begin{aligned} \phi(x, 0) &= 0, \\ \phi(0, t) &= \Phi_M, \\ \phi(\infty, t) &= 0. \end{aligned}$$

A solução analítica é (??)

$$\begin{aligned} \phi(x, t) = \frac{\Phi_M}{2} \left[ \exp \left( \frac{Ux}{2D} (1 + (1 + 2H)^{1/2}) \right) \operatorname{erfc} \left( \frac{x + Ut(1 + 2H)^{1/2}}{\sqrt{4Dt}} \right) + \right. \\ \left. \exp \left( \frac{Ux}{2D} (1 - (1 + 2H)^{1/2}) \right) \operatorname{erfc} \left( \frac{x - Ut(1 + 2H)^{1/2}}{\sqrt{4Dt}} \right) \right] \quad (3.68) \end{aligned}$$

onde

$$H = 2KD/U^2. \quad (3.69)$$

- a) Mostre que (3.68) atende à equação diferencial. Você pode fazer tudo analiticamente com lápis e papel, ou usar Maxima.
- b) Discretize (3.67) usando um esquema *upwind* implícito para  $\frac{\partial \phi}{\partial x}$  e um esquema totalmente implícito no lado direito. Faça

$$\Delta x = 0,01 = L/N, \quad (3.70)$$

$$x_i = i\Delta x. \quad (3.71)$$

- c) Resolva (3.67) numericamente com o esquema obtido acima para  $U = 1, D = 2, K = 1$  e  $\Phi_M = 1$  e compare graficamente com a solução analítica em  $t = 0,333, t = 0,666$  e  $t = 0,999$ . Por tentativa e erro, escolha  $L$  suficientemente grande para representar numericamente o “infinito”.

## SOLUÇÃO

a)

---

```

1 a : U*x/(2*D) ;
2 h : 2*K*D/U^2 ;
3 s : (1 + 2*h)**(1/2) ;
4 e1 : (x + U*t*s)/(sqrt(4*D*t)) ;
5 e2 : (x - U*t*s)/(sqrt(4*D*t)) ;
6 fi : (1/2)*(exp(a*(1 + s))*erfc(e1) + exp(a*(1-s))*erfc(e2)) ;
7 diff(fi,t) + U*diff(fi,x) - D*diff(fi,x,2) + K*fi ;
8 expand% ;
9 ratsimp% ;

```

---

b)

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + U \frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta x} &= D \frac{\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}}{\Delta x^2} - K\phi_i^{n+1} \\ \phi_i^{n+1} - \phi_i^n + \frac{U\Delta t}{\Delta x}(\phi_i^{n+1} - \phi_{i-1}^{n+1}) &= \frac{D\Delta t}{\Delta x^2}(\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}) - (K\Delta t)\phi_i^{n+1} \\ \phi_i^{n+1} - \phi_i^n + Co(\phi_i^{n+1} - \phi_{i-1}^{n+1}) &= Fo(\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}) - Ka\phi_i^{n+1} \end{aligned}$$

Passando todos os termos em  $(n + 1)$  para o lado esquerdo, e todos os termos em  $n$  para o lado direito, tem-se

$$-(Fo + Co)\phi_{i-1}^{n+1} + (1 + 2Fo + Co + Ka)\phi_i^{n+1} - Fo\phi_{i+1}^{n+1} = \phi_i^n,$$

onde

$$\begin{aligned} Co &= \frac{U\Delta t}{\Delta x}, \\ Fo &= \frac{D\Delta t}{\Delta x^2}, \\ Ka &= K\Delta t. \end{aligned}$$

Como sempre, as condições de contorno produzem linhas especiais: para  $i = 1$  e  $i = N_x$  teremos, respectivamente,

$$\begin{aligned} \phi_0^{n+1} &= \Phi_M \quad \Rightarrow \\ +(1 + 2Fo + Co + Ka)\phi_1^{n+1} - Fo\phi_2^{n+1} &= \phi_1^n + (Fo + Co)\Phi_M; \\ \Phi_N &= 0 \quad \Rightarrow \\ -(Fo + Co)\phi_{N-2}^{n+1} + (1 + 2Fo + Co + Ka)\phi_{N-1}^{n+1} &= \phi_{N-1}^n \end{aligned}$$

A listagem 3.14 mostra a implementação do esquema numérico acima com as condições de contorno. A comparação entre a solução numérica (pontos) e a solução analítica (linhas contínuas) para os 3 instantes especificados está mostrada na figura 3.14

Listagem 3.14: Implementação de um esquema numérico implícito para a equação da difusão-advecção.

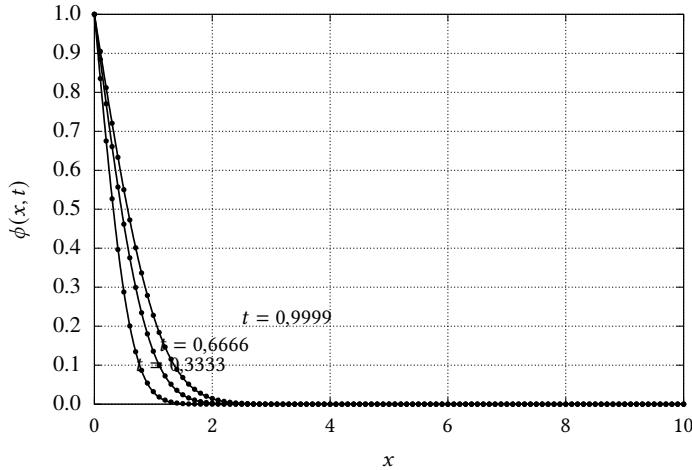


Figura 3.14: Comparação entre as soluções analíticas (linhas) e numéricas com um esquema implícito (pontos) da equação da difusão-advecção com termo de decaimento, para  $t = 0,333$ ,  $t = 0,666$  e  $t = 0,999$ .

### Exercícios Propostos

**3.8** Discretize o problema

$$\frac{d^2\phi}{dx^2} = 0, \quad \phi(0) = 1, \quad \phi(1) = 5$$

com um esquema de diferenças finitas centradas para a derivada segunda. Use  $\Delta x = 0.2$ . Obtenha as matrizes  $[A]$  ( $5 \times 5$ ) e  $[B]$  ( $5 \times 1$ ) do problema

$$[A][\phi] = [B]$$

(ou seja: obtenha  $[A]$  e  $[B]$  em *números*). Não é preciso resolver o sistema de equações. Dica: as condições de contorno modificam a primeira e a última linha de  $[A]$  e de  $[B]$ .

**3.9** Considere a equação diferencial parcial

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}, \quad 0 \leq x \leq L,$$

com condições iniciais e de contorno

$$u(x, 0) = 0, \quad u(0, t) = c, \quad \frac{\partial u}{\partial x}(L, t) = 0,$$

onde  $c$  é uma constante. Dado o esquema de discretização implícito clássico,

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = D \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}$$

para  $N_x = 8$ , obtenha o sistema de equações lineares

$$[A][u]^{n+1} = [b]$$

onde os  $A_{i,j}$ s dependem do número de grade de Fourier, e os  $b_i$ s dependem dos  $u_i^n$ s. Em outras palavras, *escreva explicitamente a matriz quadrada  $[A]$  e a matriz-coluna  $[b]$  para  $N_x = 8$* .

**3.10** O problema difusivo

$$\begin{aligned} \frac{\partial \phi}{\partial t} &= D \frac{\partial^2 \phi}{\partial x^2}, \\ \phi(0, t) &= \phi_0, \end{aligned}$$

$$\frac{\partial \phi(L, t)}{\partial x} = 0,$$

$$\phi(x, 0) = f(x),$$

possui discretização

$$-\text{Fo}\phi_{i-1}^{n+1} + (1 + 2\text{Fo})\phi_i^{n+1} - \text{Fo}\phi_{i+1}^{n+1} = \phi_i^n, \quad (3.72)$$

onde

$$\text{Fo} = \frac{D\Delta t}{\Delta x^2}$$

e  $i = 1, \dots, N - 1$  ( $i$  é o índice do eixo  $x$ ). Modifique (3.72) para levar em conta as condições de contorno, e mostre como ficam as 1ª e última linhas da matriz do sistema de equações que deve ser resolvido a cada passo.

### 3.11 Dada a equação diferencial

$$\frac{\partial \phi}{\partial t} = D \frac{\partial^2 \phi}{\partial x^2} - k\phi,$$

onde  $D > 0$  e  $k > 0$ , a sua discretização com um esquema de diferenças finitas totalmente implícito, progressivo no tempo e centrado no espaço, produz uma equação geral do tipo

$$A\phi_{i-1}^{n+1} + B\phi_i^{n+1} + C\phi_{i+1}^{n+1} = \phi_i^n,$$

onde como sempre  $\phi_i^n$  é a aproximação em grade de  $\phi(i\Delta x, n\Delta t)$ . Obtenha  $A$ ,  $B$  e  $C$  em função dos parâmetros adimensionais

$$\text{Fo} = \frac{D\Delta t}{\Delta x^2},$$

$$\text{Kt} = k\Delta t.$$

### 3.12 Dada a equação diferencial não-linear de Boussinesq,

$$\frac{\partial h}{\partial t} = \frac{\partial}{\partial x} \left[ h \frac{\partial h}{\partial x} \right],$$

$$h(x, 0) = H,$$

$$h(0, t) = H_0,$$

$$\left. \frac{\partial h}{\partial x} \right|_{x=1} = 0,$$

obtenha uma discretização linearizada da mesma em diferenças finitas do tipo

$$h(x_i, t_n) = h(i\Delta x, n\Delta t) = h_i^n$$

da seguinte forma:

- discretize a derivada parcial em relação ao tempo com um esquema progressivo no tempo entre  $n$  e  $n + 1$ ;
- aproxime  $h$  dentro do colchete por  $h_i^n$  (este é o truque que lineariza o esquema de diferenças finitas) e *mantenha-o assim*;
- utilize esquemas de diferenças finitas implícitos centrados no espaço para as derivadas parciais em relação a  $x$ , *exceto no termo  $h_i^n$  do item anterior*.

*Não mexa com as condições de contorno.*

### 3.13 Dada a equação da onda com as condições iniciais abaixo,

$$\frac{\partial^2 \phi}{\partial t^2} = \frac{\partial^2 \phi}{\partial x^2}, \quad \phi(x, 0) = x(1 - x), \quad \frac{\partial \phi(x, 0)}{\partial t} = 1, \quad 0 \leq x \leq 1, \quad 0 \leq t \leq 1,$$

- Obtenha uma discretização de diferenças finitas totalmente implícita.

- b) Sua discretização certamente envolve  $\phi_i^{n-1}$ ,  $\phi_i^n$ , e  $\phi_i^{n+1}$  simultaneamente. Portanto, se  $N = 1/\Delta x$  é o número de intervalos discretizados em  $x$ , você obviamente precisa alocar no mínimo uma matriz  $3 \times (N + 1)$  para marchar no tempo os valores de  $\phi$  (certo?). À medida que você marcha no tempo  $t$ , os índices das 3 linhas dessa matriz (que vamos chamar de  $m, n, p$ ) devem ser como se segue:

$t$	$(m, n, p)$
0	0, 1, 2
$\Delta t$	1, 2, 0
$2\Delta t$	2, 0, 1
$3\Delta t$	0, 1, 2
⋮	⋮

Escreva um trecho de programa em Python que transforma a “velha” tripla  $(m, n, p)$  na “nova” tripla  $(m, n, p)$  segundo o esquema acima.

### 3.3 – Difusão em 2 Dimensões: ADI, e equações elíticas

Consider the two-dimensional diffusion equation

$$\frac{\partial \phi}{\partial t} = D \left[ \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \right] \quad (3.73)$$

with simple initial and boundary conditions

$$\phi(0, x, y) = 1, \quad 0 < x < L, \quad 0 < y < M, \quad (3.74)$$

$$\phi(t, 0, y) = \phi(t, L, y) = 0, \quad t > 0, \quad 0 \leq y \leq M, \quad (3.75)$$

$$\phi(t, x, 0) = \phi(t, x, M) = 0, \quad t > 0, \quad 0 \leq x \leq L. \quad (3.76)$$

This has an analytical solution (modified from ?, section 13–7) of the form

$$\begin{aligned} \phi(t, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} A_{mn} \sin\left(\frac{(2m+1)\pi x}{L}\right) \sin\left(\frac{(2n+1)\pi y}{M}\right) \times \\ \exp\left\{-\pi^2 D \left[ ((2m+1)/L)^2 + ((2n+1)/M)^2 \right] t\right\}, \end{aligned} \quad (3.77)$$

where

$$A_{mn} = \frac{16}{\pi^2 (2m+1)(2n+1)}. \quad (3.78)$$

The values of the initial condition  $\phi(x, y, 0)$  in (3.74) at  $x = 0, x = L, y = 0$  and  $y = M$  are intentionally missing; indeed, many textbooks in Applied (or Engineering) Mathematics do not discuss the values of the initial condition at the boundary. Inspection of the Fourier series solution (3.77) at those points reveals that  $\phi = 0$  at these four points. In fact, the series converges pointwise to 1 at all the interior points, but to 0 at the boundary. We will remember this when we implement procedures to calculate the initial condition in the analytical and numerical solutions below.

As done in the previous section, we first write a module with the geometry of the grid and the diffusivity  $D$ . The discretization in 2 dimensions is, naturally,

$$x_i = i\Delta x, \quad i = 0, \dots, N_x, \quad (3.79)$$

$$y_j = j\Delta y, \quad j = 0, \dots, N_y, \quad (3.80)$$

$$\Delta x = L/N_x, \quad (3.81)$$

$$\Delta y = M/N_y, \quad (3.82)$$

where  $N_x$  and  $N_y$  are the numbers of discretization intervals along  $x$  and  $y$ .

For simplicity, put

$$\Delta x = \Delta y = \Delta \ell, \quad (3.83)$$

$$L = M = 1, \quad (3.84)$$

$$N_x = N_y = N_n. \quad (3.85)$$

The number of *internal* points is  $N_{n1} - 1$ . The module, `dif2grid.chpl`, is

Listagem 3.15: `difgrid2d.chpl` – Grid constants for solutions of the two-dimensional diffusion equation.

---

```

1 // =====
2 // ==> difgrid2d: grid constants for solutions of the 2-D diffusion
3 // equation
4 // =====
5 config const Nn = 128;           // number of internal points
6 const Nx = Nn;
7 const Ny = Nn;
8 config const Nt = 1000;          // number of points in t
9 const dl = 1.0/Nn;              // delta l
10 const dx = dl;                 // when needed
11 const dy = dl;                 // when needed
12 const dt = 0.1/Nt;              // delta t
13 config const deln = 100;        // write output every deln steps
14 writef("# dl = %9.4dr\n", dl);
15 writef("# Nn = %9i\n", Nn);
16 writef("# Nt = %9i\n", Nt);
17 config const Dif = 2.0;         // diffusivity

```

---

Now we move to programming the analytical solution. Summing (3.77)–(3.78) efficiently to a certain accuracy requires some rearranging. Note that the arguments of the sine functions involve only odd integers, so we enumerate the first few in table 3.1. The table suggests that we use a main index  $k$ , and then run (simultaneously) “partial sums” over  $m = 0, \dots, k$  and  $n = k - m, \dots, 0$ .

Tabela 3.1: First values of  $m$ ,  $n$ , their sum, and  $(2m + 1)$  and  $(2n + 1)$  in (3.77)–(3.78).

$m$	$n$	$k = m + n$	$2m + 1$	$2n + 1$
0	0	0	1	1
1	0	1	3	1
0	1	1	1	3
0	2	2	1	5
1	1	2	3	3
2	0	2	5	1
0	3	3	1	7
1	2	3	3	5
2	1	3	5	3
3	0	3	7	1
:	:	:	:	:

Moreover, we note (in hindsight) that the Fourier series converges *very slowly* at  $t = 0$ . The situation is much improved for  $t > 0$ , because of the exponential term that attenuates the absolute value of the terms strongly as  $k = m + n$  gets larger. Therefore, we implement the analytical solution at  $t = 0$  separately below (returning 1), and only calculate the sum of the series (to a given accuracy) for  $t > 0$ .

With these preliminary comments in mind, we move to the program that calculates the analytical solution, difana2d:

Listagem 3.16: difana2d.chpl – Analytical solution of the two-dimensional diffusion equation.

```

1 // =====
2 // ==> difana2d: analytical solution of the diffusion equation
3 //
4 //  $d\phi/dt = D (d\phi^2/dx^2 + d\phi^2/dy^2)$ 
5 //
6 //  $\phi(0,x,y) = 1$ ,
7 //  $\phi(0,0,y) = \theta$ ,  $\phi(0,1,y) = \theta$ ,
8 //  $\phi(0,x,0) = \theta$ ,  $\phi(0,x,1) = \theta$ .
9 // =====
10 use Time only stopwatch;
11 var runtime: stopwatch;
12 use unitTime;
13 use Math only exp,sin,pi;
14 use IO only openWriter,binarySerializer;
15 use difgrid2d; // local module
16 config const ananam = "difana2d.dat";
17 const fou = openWriter(ananam,serializer = new binarySerializer(),
18                         locking=false);
19 const epsilon = 1.0e-6; // accuracy of analytical solution
20 const c16pisq = 16/(pi*pi); // 16/pi^2
21 var phi: [0..Nx,0..Ny] real = 0.0; // array with solution
22 var uut = utime();
23 runtime.start();
24 for n in 0..Nt do { // loop over time
25     writeln(n);
26     forall i in 0..Nx do { // loop over space, x
27         forall j in 0..Ny do { // loop over space, y
28             phi[i,j] = ana(n,i,j);
29         }
30     }
31     if n % deln == 0 then { // write soln only every deln timesteps
32         writeln("n/deln = ",n/deln);
33         writeln("n % deln = ",n % deln);
34         fou.write(phi);
35     }
36 }
37 runtime.stop();
38 writeln("rtime = ",runtime.elapsed()/uut);
39 fou.close();
40 // =====
41 // ==> ana: calculates the analytical solution at grid point n,i,j
42 // =====
43 private proc ana(
44     const in n: int,
45     const in i: int,
46     const in j: int
47 ): real {
48     if i == 0 || i == Nx || j == 0 || j == Ny then {
49         return 0.0; // always zero at boundary
50     }
51     else if n == 0 then { // the series does not work for t = 0.
52         return 1.0;
53     }
54     const t: real = n*dt;
55     const x: real = i*dx;
56     const y: real = j*dy;
57     const pi2D = (pi**2)*Dif;
58     var sol = 0.0; // series summation, soln
59     var ds = epsilon;

```

```

60  var k = 0;
61  while abs(ds) >= epsilon do {
62      var ps = 0.0;                                // partial sum over k
63      for m in 0..k do {
64          var n = k - m;
65          var c2m1 = 2*m + 1;
66          var c2n1 = 2*n + 1;
67          var Amn = 1.0/(c2m1*c2n1);    // force floating point
68          var timf = exp(-pi2D*(c2m1**2 + c2n1**2)*t);
69          ps += Amn*sin(c2m1*pi*x)*sin(c2n1*pi*y)*timf;
70      }
71      ds = ps;
72      sol += ps;
73      k += 1;
74  }
75  sol *= c16pisq;
76  return sol;
77 }

```

---

The analytical solution procedure is again called `ana`. Note that, in line 49 we always return 0 at the boundaries, and in line 51, we avoid trying to sum the series at  $t = 0$ . Instead of passing the actual  $t, x, y$  to procedure `ana`, we prefer to pass the corresponding indices  $n, i, j$ , which are exact and avoid roundoff errors and the potential danger of actually missing the if statements in lines 49 and 51.

Note how we sum incrementally over all  $m, n$  terms such that  $m + n = k$ , calculate the “partial sum” corresponding to these terms, and then increment  $k$ . We are also saving on the size of the file output, by only writing one in every `deln` timesteps in line 31. Finally, we are timing the program. We leave to the reader to find that, by using `forall`s in lieu of `for`s in lines 26–27 and compiling with `--fast`, one can cut runtime (approximately) in half.

We now implement a serial version of the ADI (alternating direction implicit) method. Again, we seek a numerical solution

$$\phi_{n,i,j} \approx \phi(n\Delta t, i\Delta x, j\Delta y). \quad (3.86)$$

The ADI is a very clever solution that can still harness the simplicity of the TDMA even in a multi-dimensional (2D or 3D) diffusion equation. The idea is to make the algorithm implicit either in the  $\partial^2 u / \partial x^2$  or the  $\partial^2 u / \partial y^2$ , in succession, over two timesteps  $\Delta t$ . The successive discretizations are

$$\frac{\phi_{n+1,i,j} - \phi_{n,i,j}}{\Delta t} = D \left( \frac{\phi_{n+1,i+1,j} - 2\phi_{n+1,i,j} + \phi_{n+1,i-1,j}}{\Delta x^2} + \frac{\phi_{n,i,j+1} - 2\phi_{n,i,j} + \phi_{n,i,j-1}}{\Delta y^2} \right), \quad (3.87)$$

and

$$\frac{\phi_{n+2,i,j} - \phi_{n+1,i,j}}{\Delta t} = D \left( \frac{\phi_{n+1,i+1,j} - 2\phi_{n+1,i,j} + \phi_{n+1,i-1,j}}{\Delta x^2} + \frac{\phi_{n+2,i,j+1} - 2\phi_{n+2,i,j} + \phi_{n+2,i,j-1}}{\Delta y^2} \right). \quad (3.88)$$

Note that by choosing  $\Delta x = \Delta y = \Delta \ell$ , there is only one Fourier number,

$$Fo = \frac{D\Delta t}{\Delta \ell^2}. \quad (3.89)$$

Then, rearranging (3.87–3.88), we obtain the somewhat tighter versions

$$-Fo \phi_{n+1,i-1,j} + (1 + 2Fo) \phi_{n+1,i,j} - Fo \phi_{n+1,i+1,j} = Fo \phi_{n,i,j-1} + (1 - 2Fo) \phi_{n,i,j} + Fo \phi_{n,i,j+1}, \quad (3.90)$$

$$-Fo \phi_{n+2,i,j-1} + (1 + 2Fo) \phi_{n+2,i,j} - Fo \phi_{n+2,i,j+1} = Fo \phi_{n+1,i-1,j} + (1 - 2Fo) \phi_{n+1,i,j} + Fo \phi_{n+1,i+1,j}. \quad (3.91)$$

We calculate only the inner points, with  $i$  and  $j$  running from 1 to  $N_n - 1$ . The extremities are the boundary conditions, meaning that for  $i = 1, N_x - 1$  the above equations “fold”, giving, for the first and last lines of (3.90),

$$-(1 + 2Fo) \phi_{n+1,1,j} - Fo \phi_{n+1,2,j} = Fo \phi_{n,1,j-1} + (1 - 2Fo) \phi_{n,1,j} + Fo \phi_{n,1,j+1} + Fo \phi_{n+1,0,j}, \quad (3.92)$$

$$\begin{aligned}
 -\text{Fo} \phi_{n+1,N_x-2,j} + (1 + 2\text{Fo})\phi_{n+1,N_x-1,j} = \\
 \text{Fo} \phi_{n,N_x-1,j-1} + (1 - 2\text{Fo})\phi_{n,N_x-1,j} + \text{Fo} \phi_{n,N_x-1,j+1} + \text{Fo} \phi_{n+1,N_x,j}, \quad (3.93)
 \end{aligned}$$

and similarly for the  $y$  direction. With these observations, here is the program difadi2d:

Listagem 3.17: difadi2d.chpl – Numerical (ADI) solution of the two-dimensional diffusion equation.

```

1 // =====
2 // ==> difadi2d: solve the 2-dimensional diffusion equation using the
3 // alternating direction implicit method
4 //
5 // dphi/dt = D(d^2phi/dx^2 + d^2phi/dy^2),
6 //
7 // phi(0,x,y) = 1,
8 // phi(t,0,y) = phi(t,L,y) = 0,
9 // phi(t,x,0) = phi(t,x,M) = 0.
10 // =====
11 use Time only stopwatch;
12 use unittime;
13 const ut = utime();
14 var runtime: stopwatch;
15 runtime.start();
16 use difgrid2d;
17 const Fon = Dif*dt/((dl)**2);           // Fourier number
18 writef("Fo = %10.6dr\n",Fon);
19 use IO only openWriter, binarySerializer;
20 const fou = openWriter("difadi2d.dat",
21                         serializer = new binarySerializer(), locking=false);
22 var phi: [0..1,0..Nx,0..Ny] real = 0.0; // apenas 2 posições no tempo são
23                                         // necessárias!
24 for i in 0..Nx do {                  // monta a condição inicial
25     for j in 0..Ny do {
26         phi[0,i,j] = IC(i,j);
27     }
28 }
29 fou.write(phi[0..Nx,0..Ny]);          // imprime a condição inicial
30 var
31     A,                                // cria a matriz do sistema
32     B,                                // cria a matriz do sistema
33     C,                                // cria a matriz do sistema
34     D
35     : [1..Nn-1] real = 0.0;           // preciso de Nx == Ny !
36 //
37 // monta a matriz do sistema
38 //
39 A[1]      = 0.0;                      // zera A[1,1]
40 A[2..Nn-1] = -Fon;                   // preenche o fim da 1a linha
41 B[1..Nn-1] = 1.0 + 2*Fon;            // preenche a segunda linha
42 C[1..Nn-2] = -Fon;                   // preenche o início da 3a linha
43 C[Nn-1]   = 0.0;                     // zera A[2,Nn-1]
44 //
45 // importa tridiag
46 //
47 use tridiag;
48 var iold = 0;                        // o velho truque!
49 var inew = 1;
50 var n = 0;
51 while n < Nt do {                  // loop no tempo
52     n += 1;                          // incrementa n
53     writeln(n);
54 //
55 // varre na direção x
56 //

```

```

57  for j in 0..Ny do {                                // CC ao longo de x (extr. y)
58      phi[inew,0,j] = BCy(n,j);
59      phi[inew,Nx,j] = BCy(n,j);
60  }
61  for j in 1..Ny-1 do {                            // Ny-1 varreduras em x (loop em y)
62      D[1..Nx-1] = Fon*phi[iold,1..Nx-1,j-1]
63          + (1.0 - 2*Fon)*phi[iold,1..Nx-1,j]
64          + Fon*phi[iold,1..Nx-1,j+1];
65      D[1] += Fon*phi[inew,0,j];                  // CC esquerda
66      D[Nx-1] += Fon*phi[inew,Nx,j];            // CC direita
67      tridiag(A,B,C,D,phi[inew,1..Nx-1,j]);
68  }
69  if n % deln == 0 then {
70      writeln(n);
71      fou.write(phi[inew,0..Nx,0..Ny]);
72  }
73 // -----
74 // varre na direção y
75 // -----
76  inew <=> iold;
77  n += 1;
78  for i in 0..Nx do {                            // CC ao longo de y
79      phi[inew,i,0] = BCx(n,i);
80      phi[inew,i,Ny] = BCx(n,i);
81  }
82  for i in 1..Nx-1 do {                          // Nx-1 varreduras em y (loop em x)
83      D[1..Ny-1] = Fon*phi[iold,i-1,1..Ny-1]
84          + (1.0 - 2*Fon)*phi[iold,i,1..Ny-1]
85          + Fon*phi[iold,i+1,1..Ny-1];
86      D[1] += Fon*phi[inew,i,0];
87      D[Ny-1] += Fon*phi[inew,i,Ny-1];
88      tridiag(A,B,C,D,phi[inew,i,1..Ny-1]);
89  }
90  if n % deln == 0 then {
91      fou.write(phi[inew,0..Nx,0..Ny]);
92      writeln(n);
93  }
94  inew <=> iold;
95 }
96 fou.close();                                     // fecha o arquivo de saída, e fim.
97 runtime.stop();
98 writeln("rtime = ", runtime.elapsed()/ut, " ");
99 // -----
100 // condição inicial
101 // -----
102 inline proc IC(
103     const in i: int,
104     const in j: int
105 ): real {
106     if i == 0 || i == Nx || j == 0 || j == Ny then {
107         return 0.0;
108     }
109     return 1.0;
110 }
111 // -----
112 // condições de contorno
113 // -----
114 inline proc BCy(
115     const in n: int,
116     const in j: int
117 ): real {
118     return 0.0;
119 }
```

---

```

120 inline proc BCx(
121     const in n: int,
122     const in i: int
123     ): real {
124     return 0.0;
125 }
```

---

Parallelization now is achieved rather easily: the two **for** loops implementing the sweeps in the two directions starting at lines 61 and 82 can be replaced by **forall**s. However, *and this is very important*, in this case the forcing array D needs to be declared *inside* the **forall**s, to avoid race conditions. We write a parallel program, difadi2d-p. Here we show the excerpt of the changed code for the **forall** j loops (an analogous one is needed for the **forall** i loops) :

Listagem 3.18: difadi2d-p – Parallel implementation of the ADI method: sweep in the x direction.

---

```

56 forall j in 0..Ny do {                                // CC ao longo de x (extr. y)
57     phi[inew,0,j] = BCy(n,j);
58     phi[inew,Nx,j] = BCy(n,j);
59 }
60 forall j in 1..Ny-1 do {                      // Ny-1 varreduras em x (loop em y)
61     var D: [1..Nx-1] real;                    // o vetor forçante
62     D[1..Nx-1] = Fon*phi[iold,1..Nx-1,j-1]
63     + (1.0 - 2*Fon)*phi[iold,1..Nx-1,j]
64     + Fon*phi[iold,1..Nx-1,j+1];
65     D[1] += Fon*phi[inew,0,j];                // CC esquerda
66     D[Nx-1] += Fon*phi[inew,Nx,j];            // CC direita
67     // solve system in x
68     tridiag(A,B,C,D,phi[inew,1..Nx-1,j]);
69 }                                              // continua
```

---

The corresponding running times (compiling with ‘`--fast`’) of difadi2d and difadi2d-p in the authors’ computer at the time of this writing are 2.02085 s and 0.262357 s. The fully parallel version is *faster by a factor of  $\sim 7.7$*  (the computer has 8 logical cores), close to the theoretical maximum achievable. The ADI implementation presents us with a first example of a program that is large enough and complex enough for the advantages of parallelization to clearly show. But it also shows how simple and effective it is to achieve those performance improvements in Chapel, by simply changing a **for** to a **forall** at the right places – but do not forget the need for the forcing vector to be local now.

After running the analytical and numerical solutions, we want to compare them. Both output binary files, so we write a program, difview2d, to read the binary file and generate a text file with a single chosen timestep (among those available): notice that the `difana2d.dat` and `difadi2d.dat` only contain the solution every `nDEL == 100` timesteps: because each simulation involves 1000 time steps, we only have 11 of them, including the initial condition, to choose from. Here is `difview2d`:

Listagem 3.19: difview2d.chpl – Read a chosen timestep from a binary file and print it in a text file.

---

```

1 // =====
2 // ==> difview2d: write in a text file the chosen time t1 from the output
3 // produced by either difadi2d or difana2d.
4 // =====
5 use IO only fileWriter, openReader, openWriter, binaryDeserializer;
6 use difgrid2d;
7 config const finnam = "finnam.dat";
8 config const founam = "founam.out";
9 config const t1 = 0.01;                         // the time we want to see
10 const n1 = (round(t1/dt):int)/deln;           // where to seek t1,
11 writeln("t1 = ",t1);
12 writeln("dt = ",dt);
13 writeln("deln = ",deln);
14 writeln("n1 = ",n1);                            // just in case
15 const nreal = numBytes(real);                 // to seek a channel correctly
```

---

```

16 writeln("nreal = ",nreal);           // just curious
17 // abre o arquivo com os dados
18 const fin = openReader(finnam,
19                         serializer = new binaryDeserializer(),locking=false);
20 var phi: [0..Nx,0..Ny] real;        // array with soln for time
21 fin.seek(n1*(Nx+1)*(Ny+1)*nreal..);
22 fin.read(phi);                     // read u at this time
23 fin.close();                      // close reading channel
24 const fou = openWriter(founam,locking=false);    // open output channel
25 for i in 0..Nx do {
26     var xi = i*dx;
27     for j in 0..Ny do {
28         var yj = j*dy;
29         fou.writef("%8.4dr"*3 + "\n",i*dx, j*dy,phi[i,j]);
30     }
31 }
32 fou.close();                      // close output channel

```

---

The program is very simple, and the reader should have no difficulty understanding it. We compile it and run it twice, for each data file,

---

```

$ ./difview2d --t1=0.01 --finnam=difana2d.dat --founam=difana2d.out
$ ./difview2d --t1=0.01 --finnam=difadi2d.dat --founam=difadi2d.out

```

---

and plot the two solutions in figure 3.15. In the figure, the continuous mesh represents the analytical solution at  $t = 0.01$ , and the filled circles a sample of the numerical solution. The agreement is excellent, and the ADI algorithm is validated. An (essentially) equal plot is obtained from the output of program `difadi2d-p`.

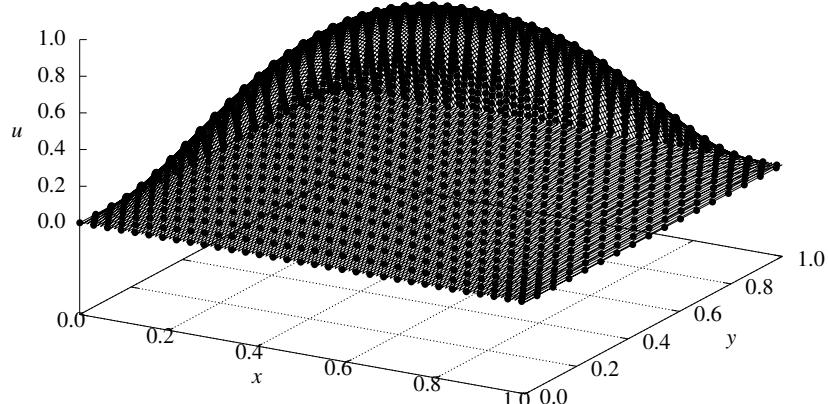


Figura 3.15: Comparison between the analytical (mesh) and numerical (points; ADI) solutions of the two-dimensional diffusion equation.

---

**Exemplo 3.5** Utilizando a análise de estabilidade de von Neumann, mostre que o esquema numéricico correspondente à primeira das equações acima é incondicionalmente estável. Suponha  $\Delta x = \Delta y = \Delta s$ .

SOLUÇÃO

Inicialmente, rearranjamos o esquema de discretização, multiplicando por  $\Delta t$  e dividindo por  $\Delta x^2$ :

$$u_{i,j}^{n+1} - u_{i,j}^n = \text{Fo} [u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1} + u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n],$$

onde

$$\text{Fo} = \frac{D\Delta t}{\Delta s^2}.$$

Faça agora

$$\begin{aligned} t_n &= n\Delta t, \\ x_i &= i\Delta s, \\ y_j &= j\Delta s, \\ \epsilon_i^n &= \sum_{l,m} \xi_{l,m} e^{at_n} e^{ik_l x_i} e^{ik_m y_j}, \end{aligned}$$

e substitua o modo  $(l, m)$  no esquema de discretização:

$$\begin{aligned} \xi_{l,m} e^{a(t_n+\Delta t)} e^{ik_l i \Delta s} e^{ik_m j \Delta s} - \xi_{l,m} e^{at_n} e^{ik_l i \Delta s} e^{ik_m j \Delta s} = \\ \text{Fo} [\xi_{l,m} e^{a(t_n+\Delta t)} e^{ik_l (i+1) \Delta s} e^{ik_m j \Delta s} - 2\xi_{l,m} e^{a(t_n+\Delta t)} e^{ik_l i \Delta s} e^{ik_m j \Delta s} \\ + \xi_{l,m} e^{a(t_n+\Delta t)} e^{ik_l (i-1) \Delta s} e^{ik_m j \Delta s} \\ + \xi_{l,m} e^{at_n} e^{ik_l i \Delta s} e^{ik_m (j+1) \Delta s} - 2\xi_{l,m} e^{at_n} e^{ik_l i \Delta s} e^{ik_m j \Delta s} + \xi_{l,m} e^{at_n} e^{ik_l i \Delta s} e^{ik_m (j-1) \Delta s}]. \end{aligned}$$

Nós imediatamente reconhecemos o fator comum

$$\xi_{l,m} e^{at_n} e^{ik_l i \Delta s} e^{ik_m j \Delta s},$$

e simplificamos:

$$\begin{aligned} e^{a\Delta t} - 1 &= \text{Fo} [e^{a\Delta t} e^{+ik_l \Delta s} - 2e^{a\Delta t} + e^{a\Delta t} e^{-ik_l \Delta s} + e^{+ik_m \Delta s} - 2 + e^{-ik_m \Delta s}]; \\ e^{a\Delta t} [1 - \text{Fo}(e^{+ik_l \Delta s} - 2 + e^{-ik_l \Delta s})] &= 1 + \text{Fo} [e^{+ik_m \Delta s} - 2 + e^{-ik_m \Delta s}], \\ e^{a\Delta t} [1 - 2\text{Fo}(\cos(k_l \Delta s) - 1)] &= 1 + 2\text{Fo}(\cos(k_m \Delta s) - 1), \\ |e^{a\Delta t}| &= \left| \frac{1 + 2\text{Fo}(\cos(k_m \Delta s) - 1)}{1 - 2\text{Fo}(\cos(k_l \Delta s) - 1)} \right|, \\ |e^{a\Delta t}| &= \left| \frac{1 - 4\text{Fo} \operatorname{sen}^2 \left( \frac{k_m \Delta s}{2} \right)}{1 + 4\text{Fo} \operatorname{sen}^2 \left( \frac{k_l \Delta s}{2} \right)} \right| \leq 1 \blacksquare \end{aligned}$$

### 3.4 – Trabalhos Computacionais

Esta seção contém diversas propostas de trabalhos computacionais. Eles são mais longos que os exercícios propostos, e requerem considerável dedicação e *tempo*. Os trabalhos desta seção mostram diversas aplicações de métodos numéricos, e lhe dão a oportunidade de ganhar uma prática considerável em programação. Não há, intencionalmente, solução destes trabalhos. Cabe a você, talvez juntamente com o seu professor, certificar-se de que os programas estão corretos. Vários dos trabalhos incluem soluções analíticas que podem ajudar nessa verificação.

#### Dispersão atmosférica de poluentes

Considere o problema de dispersão atmosférica

$$U \frac{\partial C}{\partial x} = K \frac{\partial^2 C}{\partial z^2}, \quad (3.94)$$

$$\frac{\partial C}{\partial x} = \alpha^2 \frac{\partial^2 C}{\partial z^2}, \quad (3.95)$$

$$\frac{\partial C(x, 0)}{\partial z} = \frac{\partial C(x, h)}{\partial z} = 0, \quad (3.96)$$

$$C(0, z) = \frac{Q}{U} B(z), \quad (3.97)$$

onde  $h = 1000$  m representa a altura da camada-limite atmosférica;  $Q = 1000 \mu\text{g s}^{-1}$  é a vazão mássica de poluente emitida pela chaminé,  $U = 10 \text{ m s}^{-1}$  e  $K = 10 \text{ m}^2 \text{ s}^{-1}$  são duas constantes que representam, respectivamente, uma velocidade de advecção e um coeficiente de difusão turbulenta na vertical, e

$$\alpha^2 = \frac{K}{U}, \quad (3.98)$$

$$B(z) = \begin{cases} \frac{1}{\sigma}, & |z - z_e| \leq \sigma/2, \\ 0, & |z - z_e| > \sigma/2. \end{cases} \quad (3.99)$$

Em (3.99), a função  $B(z)$  representa uma emissão localizada em uma região delgada, de espessura  $\sigma = 10$  m, em torno da altura de emissão (a altura da chaminé)  $z_e = 300$  m.

Postulamos que a solução analítica é

$$C(x, z) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n e^{-k_n^2 \alpha^2 x} \cos(k_n z), \quad (3.100)$$

onde

$$k_n = \frac{\pi n}{h}. \quad (3.101)$$

É fácil verificar que a solução postulada atende à equação diferencial (3.95). Além disso, observe que (3.100) atende automaticamente às condições de contorno (3.96). Finalmente, a integral em  $z$  de (3.100) é constante:

$$\int_0^h C(x, z) dz = \frac{a_0 h}{2}. \quad (3.102)$$

Precisamos dos coeficientes de Fourier: em  $x = 0$ ,

$$\frac{Q}{U} B(z) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(k_n z). \quad (3.103)$$

Para calcular  $a_0$ , simplesmente integre:

$$\begin{aligned} \frac{Q}{U} \int_0^h B(z) dz &= h \frac{a_0}{2} + \underbrace{\sum_{n=1}^{\infty} \int_0^h a_n \cos(k_n z) dz}_{\equiv 0} \\ a_0 &= \frac{2Q}{hU}. \end{aligned} \quad (3.104)$$

Para os demais coeficientes,  $m > 0$ ,

$$\begin{aligned} \frac{Q}{U} B(z) \cos(k_m z) &= \frac{a_0}{2} \cos(k_m z) + \sum_{n=1}^{\infty} a_n \cos(k_n z) \cos(k_m z), \\ Q \int_0^h B(z) \frac{\cos(k_m z)}{U} dz &= \frac{a_0}{2} \int_0^h \cos(k_m z) dz \\ &\quad + \sum_{n=1}^{\infty} a_n \int_0^h \cos(k_n z) \cos(k_m z) dz, \end{aligned}$$

$$\begin{aligned}
\frac{Q}{U\sigma} \int_{z_e-\sigma/2}^{z_e+\sigma/2} \cos(k_m z) dz &= \frac{a_0}{2} \int_0^h \cos(k_m z) dz \\
&\quad + \sum_{n=1}^{\infty} a_n \int_0^h \cos(k_n z) \cos(k_m z) dz, \\
\frac{2Q}{U\sigma k_m} \sin\left(\frac{k_m \sigma}{2}\right) \cos(k_m z_e) &= \frac{a_0}{2} \int_0^h \cos(k_m z) dz \\
&\quad + \sum_{n=1}^{\infty} a_n \int_0^h \cos(k_n z) \cos(k_m z) dz.
\end{aligned} \tag{3.105}$$

As funções no lado direito de (3.105) são ortogonais:

$$\int_0^h \cos(k_n z) \cos(k_m z) dz = \begin{cases} 0, & m \neq n, \\ h/2 & m = n \neq 0. \end{cases} \tag{3.106}$$

Segue-se que

$$a_m = \frac{4Q}{U\sigma\pi m} \sin\left(\frac{k_m \sigma}{2}\right) \cos(k_m z_e). \tag{3.107}$$

- a) Programe a solução analítica truncando a série do 200º harmônico (**obrigatoriamente**):

$$\widehat{C}(x, z) \approx \frac{a_0}{2} + \sum_{n=1}^{N=200} a_n e^{-k_n^2 \alpha^2 x} \cos(k_n z). \tag{3.108}$$

Discuta a convergência da série de Fourier para valores de  $N$  diferentes de 200. Plote alguns resultados.

- b) Resolva (3.95) numericamente, utilizando o esquema implícito (3.58). *Cuidado! O tratamento das condições de contorno é por sua conta.*

A condição inicial, dada por (3.97), apresenta um problema numérico potencialmente grande, e precisa ser discutida com mais detalhe. De fato,  $B(z) \neq 0$  em uma região muito fina, de largura  $\sigma = 10$  m, o que representa apenas 1% do domínio. Portanto, qualquer erro na sua representação repercutirá negativamente no esquema numérico. Para que a solução numérica seja acurada, portanto, os seguintes passos são essenciais:

1. Defina uma discretização vertical  $\Delta z$  bem menor do que  $\sigma$ .
2. Defina  $B(z)$  por pontos com resolução  $\Delta z$ ; seja  $[i_a, i_b]$  o intervalo de índices para os quais  $B(z_i) \neq 0$ . Então, é fundamental que a integral numérica de  $B(z)$  também seja unitária. Em outras palavras, você deve se certificar de que

$$\sum_{i \in [i_a, i_b]} B(z_i) \Delta z = 1.$$

### Solução linearizada da equação de Boussinesq para águas subterrâneas por analogia com a equação da difusão-advencão

Resolva a equação de Boussinesq

$$\frac{\partial \phi}{\partial t} - \left[ \frac{\partial \phi}{\partial x} \right] \frac{\partial \phi}{\partial x} - \phi \frac{\partial^2 \phi}{\partial x^2} = 0$$

com condições iniciais e de contorno

$$\phi(x, 0) = F(x),$$

$$\begin{aligned}\phi(0, t) &= 0, \\ \frac{\partial\phi(1, t)}{\partial x} &= 0.\end{aligned}$$

A solução analítica desse problema é dada por

$$\begin{aligned}\phi(x, t) &= \frac{F(x)}{1 + at}, \\ a &= [B(2/3, 1/2)]^2 / 6, \\ x &= I(2/3, 1/2, F^3), \\ F^3 &= I^{-1}(2/3, 1/2, x), \\ F(x) &= [I^{-1}(2/3, 1/2, x)]^{1/3},\end{aligned}$$

onde  $B$  é a função beta,  $I$  é a função beta incompleta, e  $I^{-1}$  é a sua inversa.

Para isso, compare a equação de Boussinesq com a de advecção-difusão unidimensional

$$\frac{\partial\phi}{\partial t} - U \frac{\partial\phi}{\partial x} - D \frac{\partial^2 u}{\partial x^2} = 0. \quad (3.109)$$

A idéia é linearizar ambos os termos não lineares com

$$\begin{aligned}U &= \left[ \frac{\partial\phi}{\partial x} \right]^n \approx \frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x}, \\ D &= \phi_i^n,\end{aligned}$$

calculados no passo de tempo anterior.

- a) Discretize (3.109) usando um esquema implícito centrado para  $\frac{\partial\phi}{\partial x}$  e  $\frac{\partial^2\phi}{\partial x^2}$ . Faça

$$\Delta x = 0,001, \quad (3.110)$$

$$\Delta t = 0,001. \quad (3.111)$$

- b) Resolva (3.109) numericamente com o esquema obtido acima para  $0 \leq x \leq 1$  e  $0 \leq t \leq 2$ .

A sua solução deve gerar pelo menos uma figura semelhante à figura para os tempos  $t = 0, 0.5, 1, 1.5$  e  $2$ . 3.16

### Solução numérica da espiral de Ekman

Em Mecânica dos Fluidos Geofísica, a espiral de Ekman é o padrão de velocidade que resulta do equilíbrio entre a força de Coriolis e as forças de atrito devidas à turbulência. Para a camada-limite atmosférica, um modelo muito simples (? , seção 13.7) — e irrealista! — postula uma viscosidade cinemática turbulenta  $\nu_W$  constante. O vetor velocidade é  $(u, v)$ , e as equações do movimento nas direções  $x$  e  $y$  são, respectivamente,

$$0 = fv + \nu_W \frac{d^2 u}{dz^2}, \quad (3.112)$$

$$0 = f(U - u) + \nu_W \frac{d^2 v}{dz^2}. \quad (3.113)$$

O vetor  $(U, 0)$  é denominado *vento geostrófico*, e  $f \cong 1 \times 10^{-4} \text{ s}^{-1}$  (a  $45^\circ$  de latitude Norte) é o parâmetro de Coriolis. As condições de contorno do problema são

$$u(0) = 0, \quad v(0) = 0, \quad (3.114)$$

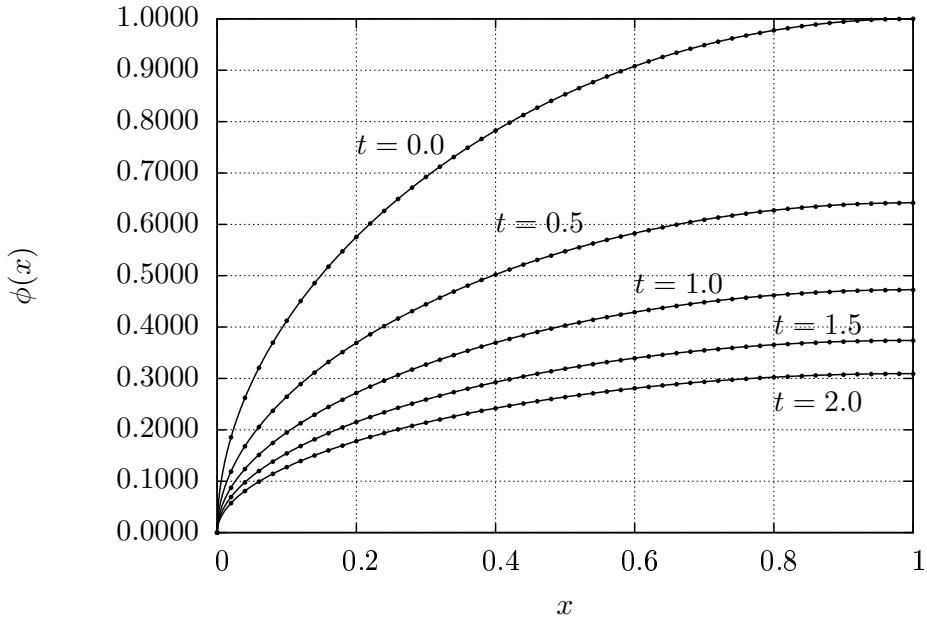


Figura 3.16: Solução correta do problema para  $t = 0, 0.5, 1.0, 1.5, 2.0$ . As linhas cheias são a solução analítica, e os pontos a solução numérica.

$$u(+\infty) = U, \quad v(+\infty) = 0. \quad (3.115)$$

A solução apresentada por Kundu usa variáveis complexas. Constrói-se uma velocidade complexa

$$W \equiv U + iV \quad (3.116)$$

( $i = \sqrt{-1}$ ) de tal forma que (3.112)–(3.113) podem ser escritas compactamente

$$\frac{d^2W}{dz^2} - \frac{if}{v_W}(W - U) = 0 \quad (3.117)$$

(Cuidado!  $z$  é *real*!). A equação tem solução

$$W = U \left[ 1 - e^{-(1+i)z/\delta} \right], \quad (3.118)$$

com

$$\delta \equiv \sqrt{\frac{2v_W}{f}}, \quad (3.119)$$

ou seja:

$$u = U \left[ 1 - e^{-z/\delta} \cos(z/\delta) \right], \quad (3.120)$$

$$v = U e^{-z/\delta} \sin(z/\delta). \quad (3.121)$$

O seu trabalho é obter uma aproximação numérica para (3.118) (ou, o que dá no mesmo, para (3.120)–(3.121)), resolvendo o problema *transiente*

$$\frac{\partial W}{\partial t} = \frac{\partial^2 W}{\partial z^2} - \frac{if}{v_W}(W - U) = 0. \quad (3.122)$$

com as condições iniciais e de contorno

$$W(z, 0) = 0, \quad (3.123)$$

$$W(0, t) = 0, \quad (3.124)$$

$$W(H, t) = U, \quad (3.125)$$

observando que  $W(z, \infty)$  (quando  $\partial W / \partial t \rightarrow 0$ ) é a solução de (3.117). Você utilizará  $H \gg \delta$  para aproximar a solução analítica, que vale para um domínio (semi-)infinito. Valores que dão a ordem de grandeza correta na atmosfera são:  $U = 10 \text{ m s}^{-1}$ ,  $v_W = 50 \text{ m}^2 \text{ s}^{-1}$ ,  $f = 1 \times 10^{-4} \text{ s}^{-1}$ ,  $\delta = 1000 \text{ m}$ ,  $H = 10000 \text{ m}$ . Embora seja possível resolver diretamente (3.122) no computador, é muito útil adimensionalizar! Fazemos

$$t = f\tau,$$

$$z = \delta\zeta,$$

$$W = \phi U,$$

e obtemos o problema (já com os valores numéricos indicados acima):

$$\frac{\partial \phi}{\partial \tau} = \frac{1}{2} \frac{\partial^2 \phi}{\partial \zeta^2} - i(\phi - 1). \quad (3.126)$$

As condições inciais e de contorno são

$$\phi(\zeta, 0) = 0, \quad (3.127)$$

$$\phi(0, t) = 0, \quad (3.128)$$

$$\phi(10, t) = 1. \quad (3.129)$$

O problema a ser resolvido numericamente agora é:

- a) Discretize a equação usando um esquema totalmente implícito para  $\frac{\partial^2 \phi}{\partial \zeta^2}$  e para  $\phi$ . Você deve descrever a discretização e explicar o esquema numérico resultante.
- b) Resolva a equação numericamente com o esquema obtido acima, usando.  $\Delta\zeta = 0,005$  e  $\Delta\tau = 0,01$  em sua solução.

A sua solução deve gerar pelo menos duas figuras semelhantes às figuras 3.17 e 3.18. Elas mostram as partes real e imaginária de  $\phi$  (que correspondem a  $u/U$  e a  $v/U$ ) nos instantes adimensionais  $\tau = 5, 25, 50$  e  $100$ .

*Atenção:* você vai precisar modificar a rotina `tridiag` (Listagem 2.13) para que ela seja capaz de lidar com números complexos.

### Uma análise numérica da equação não-linear de Boussinesq e de suas soluções aproximadas

Dada a equação de Boussinesq para um aquífero subterrâneo (?)

$$\frac{\partial h}{\partial t} = \frac{k_0}{n_e} \frac{\partial}{\partial x} \left[ h \frac{\partial h}{\partial x} \right], \quad t \geq 0; \quad 0 \leq x \leq B \quad (3.130)$$

com condições iniciais e de contorno

$$h(x, 0) = H, \quad h(0, t) = H_0 < H, \quad \frac{\partial h(B, t)}{\partial x} = 0 \quad (3.131)$$

ela pode ser adimensionalizada da seguinte forma (? , seção 10.3.5):

$$\phi \equiv \frac{h}{H}, \quad \eta \equiv \frac{x}{B}, \quad \tau \equiv \frac{k_0 H}{n_e B^2} t. \quad (3.132)$$

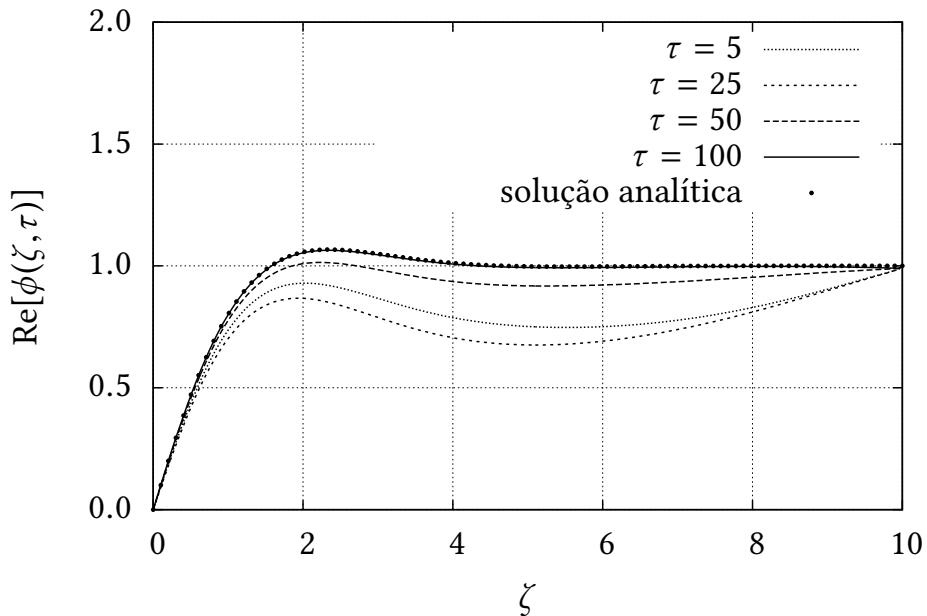


Figura 3.17: Parte real de  $\phi(\zeta, \tau)$ ,  $\tau = 5, 25, 50, 100$ , e solução analítica.

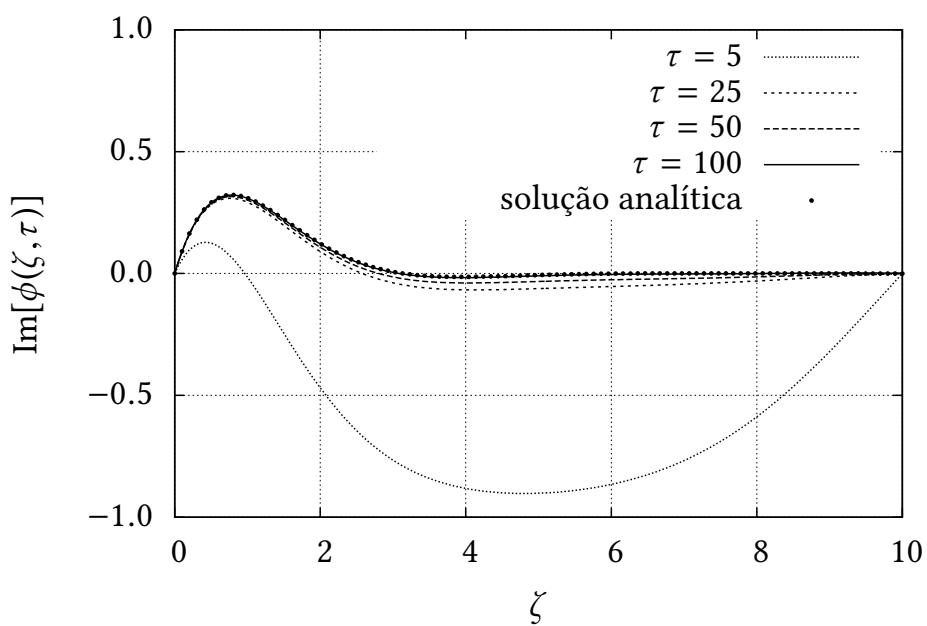


Figura 3.18: Parte imaginária de  $\phi(\zeta, \tau)$ ,  $\tau = 5, 25, 50, 100$ , e solução analítica.

Em (3.130)–(3.132),  $h$  é a carga hidráulica, e representa o nível da superfície freática (“lençol freático”),  $k_0$  é a condutividade hidráulica saturada,  $n_e$  é a porosidade drenável,  $x$  é a posição horizontal, e  $t$  é o tempo.

A vazão em  $x = 0$  também pode ser adimensionalizada:

$$\begin{aligned} q(t) &= k_0 h(0, t) \frac{\partial h(0, t)}{\partial x} \\ &= k_0 H^2 \frac{h(0, t)}{H} \frac{1}{B} \frac{\partial \frac{h(0, t)}{H}}{\partial \frac{x}{B}} \\ &= \frac{k_0 H^2}{B} \left[ \phi \frac{\partial \phi}{\partial \eta} \right] (0, \tau) \\ &= \frac{k_0 H^2}{B} \chi(\tau), \end{aligned} \quad (3.133)$$

onde  $\chi(\tau)$  é a vazão adimensional efluente do maciço poroso em  $\eta = 0$ .

Nas variáveis acima, a equação pode ser reescrita como

$$\frac{\partial \phi}{\partial \tau} = \frac{\partial}{\partial \eta} \left[ \phi \frac{\partial \phi}{\partial \eta} \right] \quad \tau \geq 0, \quad 0 \leq \eta \leq 1, \quad (3.134)$$

com condições iniciais e de contorno

$$\phi(\eta, 0) = 1, \quad \phi(0, \tau) = \Phi_0 < 1, \quad \frac{\partial \phi(1, \tau)}{\partial \eta} = 0. \quad (3.135)$$

### Tempos longos

Quando  $\Phi_0 = 0$ , com a condição inicial

$$\phi(\eta, 0) = F(\eta) \quad (3.136)$$

substituindo a primeira das equações (3.135), existe a solução analítica (?):

$$\phi(\eta, t) = \frac{F(\eta)}{1 + \alpha \tau}, \quad (3.137)$$

$$\chi(\tau) = \frac{\beta}{(1 + \alpha \tau)^2}, \quad (3.138)$$

$$\alpha = [B(2/3, 1/2)]^2 / 6, \quad (3.139)$$

$$\beta = B(2/3, 1/2)/3, \quad (3.140)$$

$$\eta = I_{F^3}(2/3, 1/2), \quad (3.141)$$

onde  $B(\alpha, \beta)$  é a função Beta, e  $I_x(\alpha, \beta)$  é a função Beta incompleta (?), seção 6.4). Note que  $[F(\eta)]^3$  é a função inversa de  $I_x(2/3, 1/2)$ . Uma importante limitação desta solução é que ela somente é válida para uma condição inicial muito específica, dada pela função Beta incompleta inversa  $F(\eta)$ . Outra limitação é que ela só se aplica a  $\Phi_0 = 0$ . Ela pode ser muito útil, entretanto, para verificar a qualidade de soluções numéricas da equação não-linear (3.134).

### Tempos longos, solução linearizada

Considere novamente (3.130), linearizada:

$$\begin{aligned} \frac{\partial h}{\partial t} &= \frac{p k_0 H}{n_e} \frac{\partial^2 h}{\partial x^2}, \\ \frac{\partial(h/H)}{\partial t} &= \frac{p k_0 H}{n_e} \frac{\partial^2(h/H)}{\partial x^2}, \end{aligned}$$

$$\begin{aligned}\frac{\partial \phi}{\partial t} &= \frac{pk_0H}{n_e} \frac{\partial^2 \phi}{\partial x^2}, \\ \frac{\partial \phi}{\partial t} &= \frac{pk_0H}{n_e B^2} \frac{\partial^2 \phi}{\partial \eta^2}, \\ \frac{\partial \phi}{\partial \tau} &= p \frac{\partial^2 \phi}{\partial \eta^2}.\end{aligned}\quad (3.142)$$

A equação (3.142) tem solução em série de Fourier, da forma

$$\phi(\eta, \tau) = \Phi_0 + \frac{4(1 - \Phi_0)}{\pi} \sum_{n=1}^{\infty} \frac{1}{(2n-1)} \sin\left((2n-1)\frac{\pi}{2}\eta\right) \exp\left(-\frac{(2n-1)^2\pi^2}{4}p\tau\right). \quad (3.143)$$

Seguindo ? e ?, nós calcularemos  $p$  em função de  $\Phi_0$  como:

$$p = 0.3465 + 0.6535\Phi_0. \quad (3.144)$$

## Nossos objetivos

O objetivo deste trabalho são:

1. Produzir um esquema numérico de solução que possa ser comparado com a solução analítica (3.137)–(3.141) quando  $\Phi_0 = 0$ .
2. Analisar o comportamento das soluções analíticas linearizadas para diversos casos  $0 < \Phi_0 < 1$ , comparando-as com as soluções numéricas correspondentes da equação não-linear.

### 1<sup>a</sup> tarefa

Crie um programa que calcula a solução analítica dada pelas equações (3.136)–(3.141). Verifique como os perfis  $\phi(\eta, \tau)$  evoluem no tempo. Para fazer isso, ajudará muito se você instalar a módulo adicional `scipy`. As funções Beta e Beta incompleta que aparecem em (3.139), (3.140) e (3.141) são implementadas nas funções `beta` e `betainc` de `scipy.special`. No seu programa, faça então:

```
[ from scipy.special import beta, betainc ],
```

e use `beta` e `betainc` da forma adequada.

Agora, plote a solução analítica  $\phi(\eta, \tau)$  para  $\tau = 0,01, 0,1$  e  $1,0$ .

## Solução numérica da equação não-linear

O primeiro passo é discretizar a equação diferencial não-linear, “linearizando-a” por meio do expediente de aplicar parte da discretização utilizando o instante de tempo anterior. Faça

$$\phi \frac{\partial \phi}{\partial \eta} \Big|_{i+1/2} \approx \frac{\phi_{i+1}^n + \phi_i^n}{2} \left[ \frac{\phi_{i+1}^{n+1} - \phi_i^{n+1}}{\Delta \eta} \right]; \quad (3.145)$$

$$\phi \frac{\partial \phi}{\partial \eta} \Big|_{i-1/2} \approx \frac{\phi_i^n + \phi_{i-1}^n}{2} \left[ \frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta \eta} \right]; \quad (3.146)$$

como estamos usando o “ $\phi$ ” do passo tempo anterior ( $n$ ), isso na prática lineariza o esquema. Vamos simplificar a notação:

$$\bar{\phi}_i^n \equiv \frac{\phi_{i+1}^n + \phi_i^n}{2}, \quad \bar{\phi}_{i-1}^n \equiv \frac{\phi_i^n + \phi_{i-1}^n}{2}, \quad (3.147)$$

o que nos permite reescrever (3.145)–(3.146) como

$$\phi \frac{\partial \phi}{\partial \eta} \Big|_{i+1/2} \approx \bar{\phi}_i^n \left[ \frac{\phi_{i+1}^{n+1} - \phi_i^{n+1}}{\Delta \eta} \right]; \quad \phi \frac{\partial \phi}{\partial \eta} \Big|_{i-1/2} \approx \bar{\phi}_{i-1}^n \left[ \frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta \eta} \right]. \quad (3.148)$$

Podemos agora calcular a segunda derivada (aproximada):

$$\begin{aligned} \frac{\partial}{\partial \eta} \left[ \phi \frac{\partial \phi}{\partial \eta} \right]_i &\approx \frac{1}{\Delta \eta} \left[ \phi \frac{\partial \phi}{\partial \eta} \Big|_{i+1/2} - \phi \frac{\partial \phi}{\partial \eta} \Big|_{i-1/2} \right] \\ &= \frac{1}{\Delta \eta} \left[ \bar{\phi}_i^n \left[ \frac{\phi_{i+1}^{n+1} - \phi_i^{n+1}}{\Delta \eta} \right] - \bar{\phi}_{i-1}^n \left[ \frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta \eta} \right] \right]. \end{aligned} \quad (3.149)$$

O esquema numérico agora é

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta \tau} = \frac{1}{\Delta \eta} \left[ \bar{\phi}_i^n \left[ \frac{\phi_{i+1}^{n+1} - \phi_i^{n+1}}{\Delta \eta} \right] - \bar{\phi}_{i-1}^n \left[ \frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta \eta} \right] \right] \quad (3.150)$$

ou

$$\begin{aligned} \phi_i^{n+1} - \phi_i^n &= \frac{\Delta t}{\Delta \eta^2} \left[ \bar{\phi}_i^n (\phi_{i+1}^{n+1} - \phi_i^{n+1}) - \bar{\phi}_{i-1}^n (\phi_i^{n+1} - \phi_{i-1}^{n+1}) \right] \\ \phi_i^{n+1} - \phi_i^n &= \text{Fo} \left[ \bar{\phi}_i^n (\phi_{i+1}^{n+1} - \phi_i^{n+1}) - \bar{\phi}_{i-1}^n (\phi_i^{n+1} - \phi_{i-1}^{n+1}) \right] \\ \phi_i^{n+1} - \phi_i^n &= \text{Fo} \bar{\phi}_{i-1}^n \phi_{i-1}^{n+1} - \text{Fo} \left( \bar{\phi}_{i-1}^n + \bar{\phi}_i^n \right) \phi_i^{n+1} + \text{Fo} \bar{\phi}_i^n \phi_{i+1}^{n+1}. \end{aligned}$$

O esquema numérico pode ser rearrumado da seguinte maneira:

$$- \left[ \text{Fo} \bar{\phi}_{i-1}^n \right] \phi_{i-1}^{n+1} + \left[ 1 + \text{Fo} \left( \bar{\phi}_{i-1}^n + \bar{\phi}_i^n \right) \right] \phi_i^{n+1} - \left[ \text{Fo} \bar{\phi}_i^n \right] \phi_{i+1}^{n+1} = \phi_i^n \quad (3.151)$$

Talvez seja prático definir os coeficientes a seguir. Para  $1 \leq i \leq N_x - 2$ :

$$A_{i-1}^n \equiv - \left[ \text{Fo} \bar{\phi}_{i-1}^n \right], \quad (3.152)$$

$$B_{i-1}^n \equiv \left[ 1 + \text{Fo} \left( \bar{\phi}_{i-1}^n + \bar{\phi}_i^n \right) \right], \quad (3.153)$$

$$C_{i-1}^n \equiv - \left[ \text{Fo} \bar{\phi}_i^n \right], \quad (3.154)$$

e então reescrever

$$A_{i-1}^n \phi_{i-1}^{n+1} + B_{i-1}^n \phi_i^{n+1} + C_{i-1}^n \phi_{i+1}^{n+1} = \phi_i^n. \quad (3.155)$$

As condições de contorno agora são as seguintes: para a primeira linha, teremos

$$B_0^n \phi_1^{n+1} + C_0^n \phi_2^{n+1} = \phi_i^n - A_0^n \Phi_0. \quad (3.156)$$

Já na última linha,  $i = N_x - 1$ , e teremos

$$\phi_{N_x-1}^{n+1} = \phi_{N_x}^{n+1} \quad (3.157)$$

e portanto

$$A_{N_x-2}^n \equiv - \left[ \text{Fo} \bar{\phi}_{N_x-2}^n \right], \quad (3.158)$$

$$B_{N_x-2}^n \equiv \left[ 1 + \text{Fo} \left( \bar{\phi}_{N_x-2}^n \right) \right]; \quad (3.159)$$

a última linha será

$$A_{N_x-2}^n \phi_{N_x-1}^{n+1} + B_{N_x-2}^n \phi_{N_x-1}^{n+1} = \phi_{N_x-1}^n. \quad (3.160)$$

Como sempre, uma boa notação torna as coisas quase triviais: notação é importante!

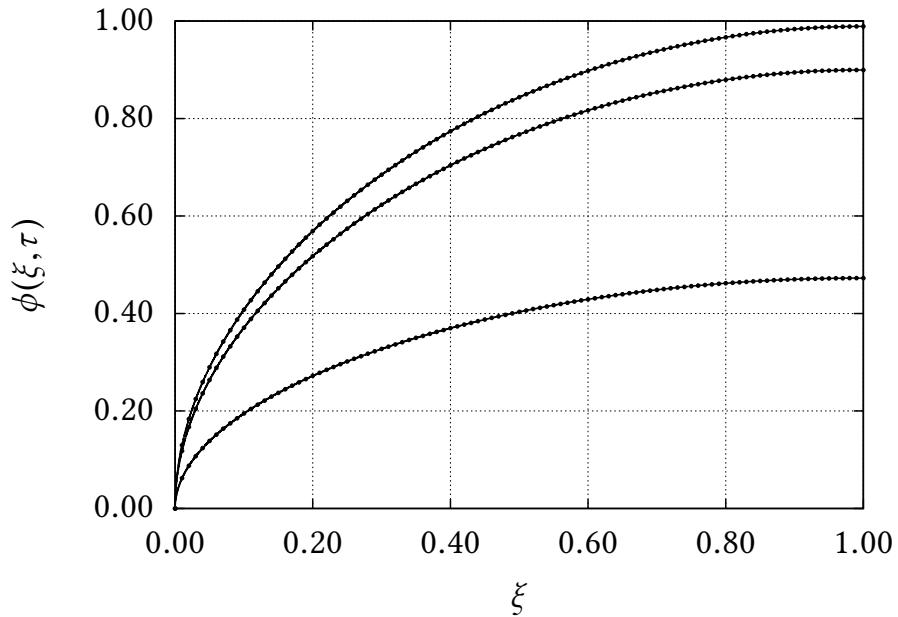


Figura 3.19: Comparaçāo do esquema numérico com a solução analítica de ?. De cima para baixo,  $\tau = 0,01$ ,  $\tau = 0,1$ , e  $\tau = 1,0$

### 2<sup>a</sup> tarefa

Agora, implemente o esquema numérico. Crie um programa para comparar a solução analítica que você programou acima com a solução numérica que você implementou. Atente para a condição inicial que você deve usar, que é a dada pela equação (3.136). Sugestāo: use incrementos  $\Delta\eta = 0,0001$  e  $\Delta\tau = 0,0001$ . Minha comparação é excelente, como podemos ver na figura 3.19. Os instantes mostrados do esquema numérico são os mesmos de antes, é claro:  $\tau = 0,01$ ,  $\tau = 0,1$  e  $\tau = 1,0$ .

### 3<sup>a</sup> tarefa

Escreva um programa para calcular a solução analítica aproximada dada pela equação (3.143). O valor de  $\Phi_0$  deve ser um argumento do programa, para que você possa calcular a solução analítica em função da condição de contorno à esquerda. Novamente, o programa deve imprimir as soluções para  $\tau = 0,01$ ,  $\tau = 0,1$ , e  $\tau = 1,0$ .

### 4<sup>a</sup> tarefa

Agora escreva um programa para resolver o problema definido pelas equações (3.134)–(3.135) numericamente. Esse programa também deve ter com argumento  $\Phi_0$ . Tendo feito isso, compare os resultados da solução não-linear com os resultados da solução linearizada calculada anteriormente, para  $\tau = 0,01$ ,  $\tau = 0,1$ , e  $\tau = 1,0$ . A figura 3.20 mostra a minha comparação para  $\phi(\eta, 0) = 0$ . Repita, para  $\Phi_0 = 0,25$ ,  $0,50$  e  $0,75$ . (vide figuras 3.21, 3.22 e 3.23).

Como você interpreta os resultados obtidos em função de  $\Phi_0$ ?

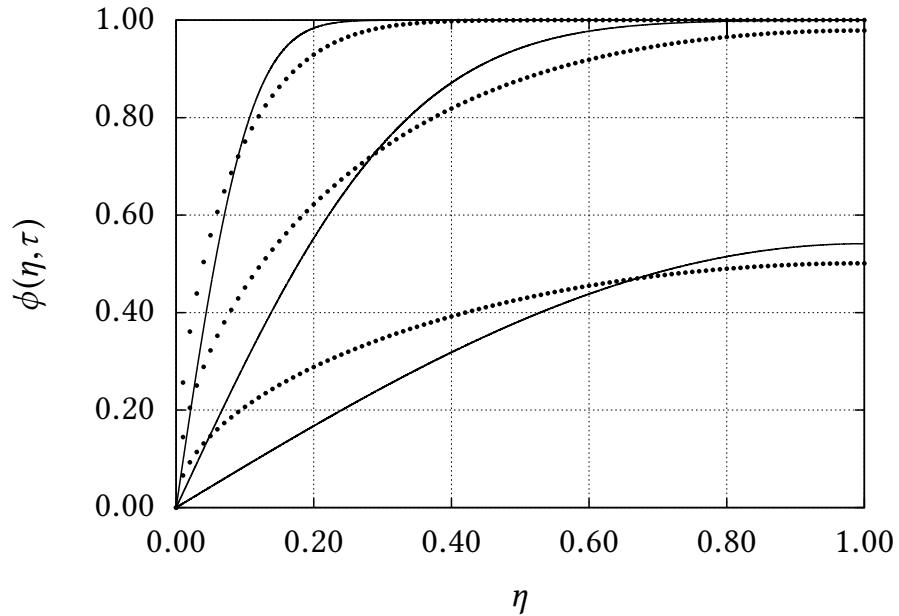


Figura 3.20: Comparação da solução numérica completa de (3.134)–(3.135) com a solução linearizada (3.143) para  $\Phi_0 = 0$ . De cima para baixo,  $\tau = 0,01$ ,  $\tau = 0,1$ , e  $\tau = 1,0$ .

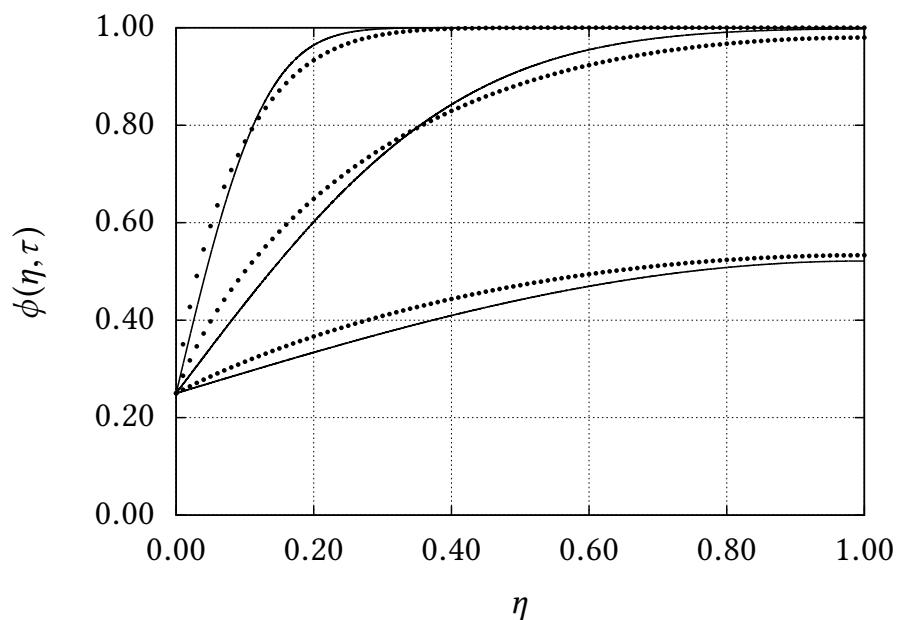


Figura 3.21: Comparação da solução numérica completa de (3.134)–(3.135) com a solução linearizada (3.143) para  $\Phi_0 = 0,25$ . De cima para baixo,  $\tau = 0,01$ ,  $\tau = 0,1$ , e  $\tau = 1,0$ .

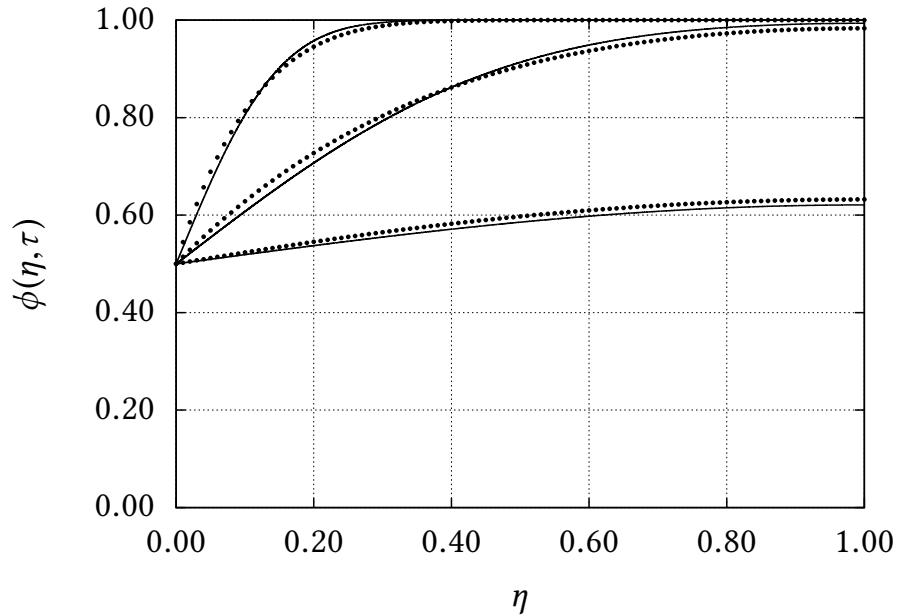


Figura 3.22: Comparação da solução numérica completa de (3.134)–(3.135) com a solução linearizada (3.143) para  $\Phi_0 = 0,50$ . De cima para baixo,  $\tau = 0,01$ ,  $\tau = 0,1$ , e  $\tau = 1,0$ .

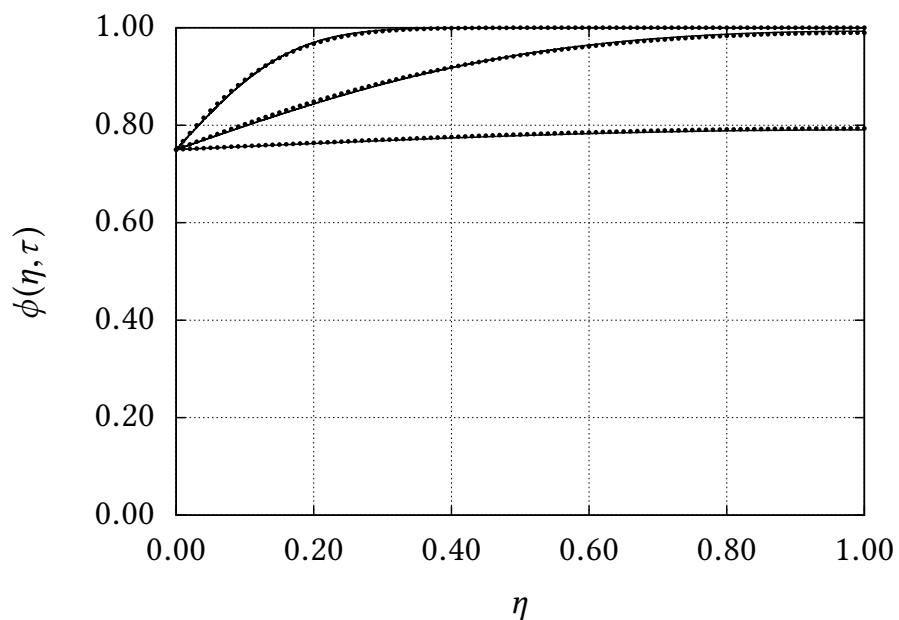


Figura 3.23: Comparação da solução numérica completa de (3.134)–(3.135) com a solução linearizada (3.143) para  $\Phi_0 = 0,75$ .

### 3.5 – Successive over-relaxation

Given Laplace's equation in two dimensions,

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0, \quad (3.161)$$

a standard discretization is

$$\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y^2} = 0. \quad (3.162)$$

With  $\Delta x = \Delta y$ , we obtain

$$\phi_{i,j} = \frac{1}{4} [\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}]. \quad (3.163)$$

Equation 3.163 is the basis for an over-relaxation method to solve (3.161) numerically. For any iterative method of the type

$$\mathbf{u}^{k+1} = f(\mathbf{u}^k), \quad (3.164)$$

where  $f$  defines the iterative method, a relaxation method is

$$\mathbf{u}^{k+1} = \omega f(\mathbf{u}^k) + (1 - \omega) \mathbf{u}^k. \quad (3.165)$$

where  $\omega$  is the *relaxation factor*. When  $1 < \omega < 2$ , convergence is accelerated, and the method is called *successive over-relaxation* (SOR) (? , section 19.5). A somewhat more convenient form for computation is

$$\begin{aligned} \mathbf{u}^{k+1} &= \omega [f(\mathbf{u}^k) - \mathbf{u}^k] + \mathbf{u}^k; \\ \delta \mathbf{u}^{k+1} &\equiv \mathbf{u}^{k+1} - \mathbf{u}^k = \omega [f(\mathbf{u}^k) - \mathbf{u}^k]. \end{aligned} \quad (3.166)$$

The term in brackets in (3.166) above is the original increment of the iterative method (3.164); in this sense, over-relaxation is simply to multiply each step of the iterative method by  $\omega$ ; for  $\omega > 1$ , this accelerates convergence towards the correct value. By calculating  $\delta \mathbf{u}^k$  in every step, we have a measure of how close we are to convergence, and can adopt a convenient norm  $\|\delta \mathbf{u}^{k+1}\|$  as a stopping criterion. With  $\delta \mathbf{u}^{k+1}$  in hand, the next value is calculated as

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \delta \mathbf{u}^{k+1}. \quad (3.167)$$

Consider then (3.161) with boundary conditions

$$\phi(x, 0) = x, \quad (3.168)$$

$$\phi(x, 1) = 1. \quad (3.169)$$

It is straightforward to verify that the analytical solution is

$$\phi(x, y) = x + y - xy. \quad (3.170)$$

It is now relatively simple to devise an iterative solution on the basis of (3.163), by writing

$$\phi_{i,j}^{k+1} = \frac{1}{4} [\phi_{i+1,j}^k + \phi_{i-1,j}^k + \phi_{i,j+1}^k + \phi_{i,j-1}^k], \quad (3.171)$$

where each iteration  $k$  updates a single node value; this is actually an implementation of the Gauss-Seidel method for solving linear systems of equations; we can recognize that  $f$  in this case is the average of the four neighbor values in the grid.

The serial implementation of the SOR method for Laplace's equation is fairly straightforward: we can see it in program `laplace-sor`:

---

Listagem 3.20: *laplace-sor.chpl* – Solution of Laplace’s equation with successive over relaxation.

---

```

1 // =====
2 // ==> laplace-sor: over-relaxation
3 // =====
4 use Time only stopwatch;
5 use unitTime;
6 const ut = utime();
7 config const epsilon = 1.0e-8;           // accuracy
8 config const omega = 1.95;             // relaxation factor
9 config const Nn = 128;                // number of points
10 const dl = 1.0/Nn;                  // delta l
11 var phi: [0..Nn,0..Nn] real = 0.0;    // iterative solution
12 for i in 0..Nn do {                 // the initial guess and BCs!!!
13     for j in 0..Nn do {
14         phi[i,j] = IG(i,j);
15     }
16 }
17 var runtime: stopwatch;
18 var deltam = epsilon;                  // the norm
19 var nc = 0;                          // # of iterations till convergence
20 runtime.start();
21 //
22 while deltam >= epsilon do {
23     deltam = 0.0;
24     for i in 1..Nn-1 do {
25         for j in 1..Nn-1 do {
26             var phiavg = (phi[i+1,j]+phi[i-1,j]+phi[i,j-1]+phi[i,j+1])/4.0;
27             var deltau = omega*(phiavg - phi[i,j]);
28             phi[i,j] += deltau;
29             deltau = abs(deltau);
30             deltam += deltau;
31         }
32     }
33     deltam /= ((Nn-1)**2) ;
34     nc += 1;
35 }
36 // done
37 runtime.stop();
38 var sum = 0.0;
39 var mad = 0.0;
40 for i in 0..Nn do {
41     for j in 0..Nn do {
42         sum += phi[i,j];
43         var del = abs(phi[i,j] - ana(i,j));
44         mad += del;
45     }
46 }
47 mad /= ((Nn+1)**2);
48 writelf( "# Nn = %9i ", Nn);
49 writelf( "nc = %6i ",nc);
50 writelf( "phi_avg = %8.4dr ",sum/((Nn+1)**2));
51 writelf( "mad =      %8.4er ",mad);
52 writelf( "rtime =    %8.4dr\n",runtime.elapsed()/ut);
53 //
54 // --> initial guess
55 //
56 private inline proc IG(i,j: int): real {
57     const x = i*dl;
58     const y = j*dl;
59     if i == 0 then {
60         return y;
61     }
62     else if i == Nn then {

```

```

63     return 1.0;
64 }
65 else if j == 0 then {
66     return x;
67 }
68 else if j == Nn then {
69     return 1.0;
70 }
71 else {
72     return 0.50;
73 }
74 }
75 // -----
76 // --> the analytical solution
77 // -----
78 private inline proc ana(i,j): real {
79     const x = i*dl;
80     const y = j*dl;
81     return x + y - x*y;
82 }
```

Some general remarks are in order: for the sake of simplicity, we are comparing the numerical solution to the analytical solution in the same program, and printing an overall error statistic, the mean absolute deviation (MAD) between the two. The stated “accuracy” for convergence of the iterative method,  $10^{-8}$ , actually produces a  $\text{MAD} = 2.56 \times 10^{-7}$  (with the default **config** constants). In other words, the numerical solution seems to be slightly biased with respect to the analytical solution. We are also calculating numerically the mean value of  $\phi$  over the domain, whose analytical value is

$$\bar{\phi} = \int_{x=0}^1 \int_{y=0}^1 (x + y - xy) \, dy \, dx = \frac{3}{4}. \quad (3.172)$$

The number of required iterations depends on the initial guess, provided in the local procedure **IG**. It returns the known boundary conditions of the analytical solution, and an intermediate value (0.5) between 0 and 1 for the interior points. The default relaxation factor,  $\omega = 1.95$ , has nothing magical to it: it is close to the optimal in this case, and was found by trial-and-error (note that, depending on the problem, there are analytical values that have been obtained for  $\omega$  (? and ?, section 19.5)).

With Chapel, in principle it is easy to obtain quickly a parallel version of **laplace-sor** by replacing the two **for** loops in **i** and **j** with **forall**s. However, in practice this produces a *substantially* different algorithm: the order in which the updates given by (3.171) are calculated is now essentially random, effectively generating race conditions. Aggressively changing both **fors** with **forall**s may lead to wrong results. In order to mitigate this, therefore, we only convert the outer loop into a **forall**.

The next problem is how to calculate the convergence parameter **deltam**, because Chapel does not allow by default to change a basic variable inside a **forall** that was declared outside of it. The best thing to do is to declare two auxiliary arrays, **deltaxm** and **deltaym** outside of each **forall/for** loop respectively, and use them to accumulate partial sums. The modified program is **laplace-sor-p**, and we only list the changed code.

Listagem 3.21: **laplace-sor-p** — Parallel solution of Laplace’s equation with successive over relaxation.

```

23 while deltam >= epsilon do {           // change to parallel
24     deltam = 0.0;
25     var deltaxm: [1..Nn-1] real;
26     forall i in 1..Nn-1 do {
27         var deltaym: [1..Nn-1] real;
28         for j in 1..Nn-1 do {
29             var phiavg = (phi[i+1,j]+phi[i-1,j]+phi[i,j-1]+phi[i,j+1])/4.0;
30             var deltaphi = omega*(phiavg - phi[i,j]);
```

```

31     phi[i,j] += deltaphi;
32     deltaphi = abs(deltaphi);
33     deltaym[j] = deltaphi;
34   }
35   deltaxm[i] = (+ reduce deltaym)/(Nn-1);
36 }
37 deltam = (+ reduce deltaxm)/(Nn-1);
38 nc += 1;
39 }                                // done

```

We can now run both the serial and the parallel versions for successively denser grids,  $N_n = 128, 256, 512, 1024$  and  $2048$ , measuring the number of steps needed for convergence  $n_c$ , their estimated values of  $\bar{u}$ , MADs and run times  $t_r$ . The result is given in table 3.2.

Tabela 3.2: Grid size  $N_n$ , number of iterations to convergence  $n_c$ , estimated  $\bar{u}$ , MAD and relative runtime  $t_r$  for the serial and parallel versions of the solution of Laplace's equation with SOR.

$N_n$	serial				parallel			
	$n_c$	$\bar{u}$	MAD	$t_r$	$n_c$	$\bar{\phi}$	MAD	$t_r$ (s)
128	431	0.7500	$2.5625 \times 10^{-7}$	0.0174	428	0.7500	$2.5438 \times 10^{-7}$	0.0051
256	1934	0.7500	$1.5653 \times 10^{-6}$	0.3223	1930	0.7500	$1.5606 \times 10^{-6}$	0.0425
512	6947	0.7500	$6.6456 \times 10^{-6}$	4.8341	6942	0.7500	$6.6515 \times 10^{-6}$	0.4589
1024	23955	0.7500	$2.7032 \times 10^{-5}$	67.6316	23951	0.7500	$2.7027 \times 10^{-5}$	5.7253
2048	80310	0.7499	$1.0864 \times 10^{-4}$	916.1190	80305	0.7499	$1.0865 \times 10^{-4}$	73.0319

Note that the parallel version has essentially the same statistics as the serial for  $n_c$ ,  $\bar{\phi}$  and MAD. The ratio of the running times  $t_r$  deserves a special attention, in table 3.3. Table 3.3 shows that the very simple parallelization achieved by replacing **for** with **forall** is able to produce a speed up factor of (approximately) 4–12, out of a theoretical maximum of 16 (when run in a computer with 16 performance cores).

Tabela 3.3: Ratio of serial to parallel runtimes.

$N_n$	128	256	512	1024	2048
Ratio	3.4118	7.5835	10.5341	11.8128	12.5441

The SOR method is more efficient than, say, Gaussian elimination. The latter is an  $O(n^3)$  (order of  $n^3$ ) algorithm (?), section 2.2), whereas SOR is  $O(n^2)$ , where  $n = N_n^2$  is the number of unknowns. Figure 3.24, obtained from table 3.2 above by plotting  $N_n^2/1000 \times t_r$ , shows that the running time is well represented by a function of the type  $t_r = an^2$ .

### 3.6 – Really efficient Successive Over-Relaxation

So far, we have chosen simplicity over efficiency, to illustrate a few ideas on parallelization of classical algorithms. But SOR methods can still be made much more efficient, while avoiding race conditions in the parallel versions at the same time. This consists of two ideas, clearly explained in section 19.5 of ? and in ?. The first is to apply (3.171) alternately on the “black” and “white” grid points shown in figure 3.25: we update the  $\phi_{i,j}$  values first on the black grid points and then on the white (or vice-versa) in two half-sweeps. In each half-sweep, the new values only depend on the old ones, and race conditions never appear. The second is to use *Chebyshev acceleration* to update the relaxation parameter  $\omega$  continuously through the algorithm. We will not delve into the details, and refer the reader to ?.

Putting everything together, we now have the program `laplace-asor`, in listing 3.22.

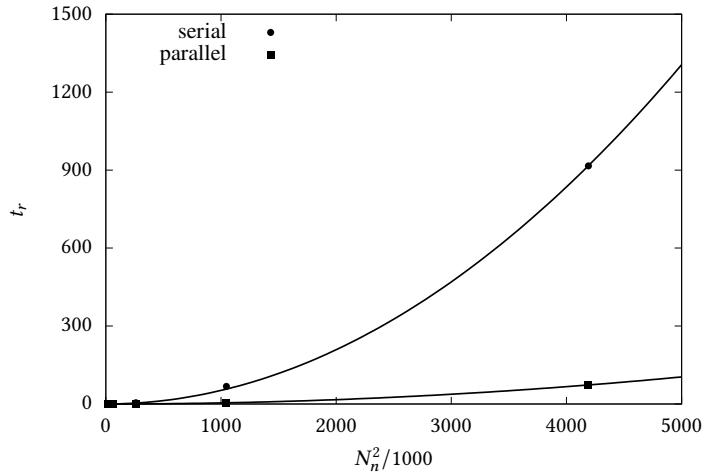


Figura 3.24: Running times  $t_r$  as a function of  $n = N_n^2$  for the SOR method, serial and parallel implementations. The lines are parabolas of the type  $t_r = an^2$ .

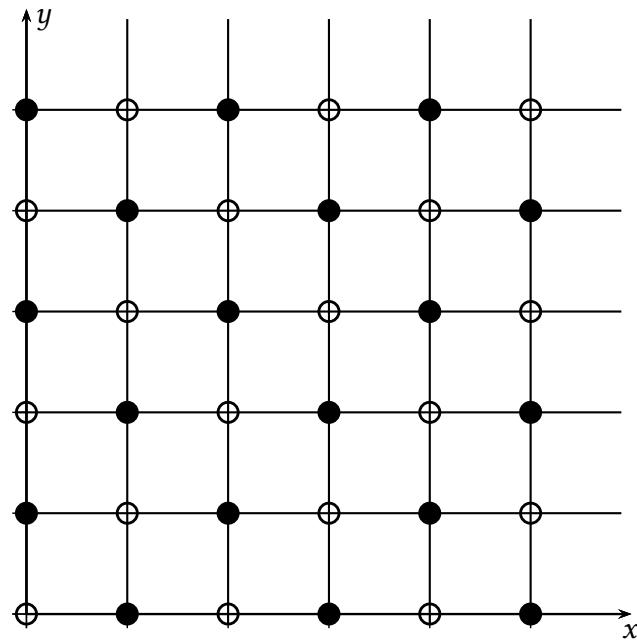


Figura 3.25: Alternating grids in 2D for the SOR methods. Note that when we update the black points using (3.171), only the white points are used, and vice-versa. In this way, we can update the whole grid in 2 half-sweeps, with the blacks depending only the whites (and vice-versa), so that race conditions never arise.

Listagem 3.22: `laplace-asor` – Solution of Laplace's equation in 2D over staggered grids using Chebyshev acceleration.

---

```

1 // =====
2 // ==> laplace-asor: adaptive successive over-relaxation with Laplace's equation
3 //
4 // For details, see NR and Hansen, P. B.
5 //
6 // NR == Press, W. H.; Teukolsky, S. A.; Vetterling, W. T. & Flannery,
7 // B. P. Numerical Recipes in C; The Art of Scientific Computing
8 // Cambridge University Press, 1992
9 //
10 // Hansen, P. B. Numerical solution of Laplace's equation Syracuse
11 // University, College of Engineering and Computer Science, Syracuse
12 // University, College of Engineering and Computer Science, 1992
13 // =====
14 use Time only stopwatch;
15 use unitTime;
16 use Math only pi,cos;
17 const ut = utime();
18 config const epsilon = 1.0e-8;           // accuracy
19 config const Nn = 128;                  // number of intervals
20 const dl = 1.0/Nn;                     // delta l
21 const rhoJ = cos(pi/(Nn+1));          // Jacobi spectral radius (see NR
22                                // p. 865--867)
23 const omega_f = 2.0/(1.0 + sqrt(1.0 - rhoJ**2));
24 writef("# Nn = %9i\n", Nn);
25 writef("# dl = %9.4dr\n", dl);
26 writef("# rhoJ = %9.4dr\n", rhoJ);
27 writef("# omega_f = %9.4dr\n", omega_f);
28 var phi: [0..Nn,0..Nn] real = 0.0;      // iterative solution
29 for i in 0..Nn do {                   // the initial guess
30     for j in 0..Nn do {
31         phi[i,j] = IG(i,j);
32     }
33 }
34 var omega: real;                      // at the start of Chebyshev acc.
35 var runtime: stopwatch;
36 var deltam = epsilon;                 // the norm
37 var nc = 0;                          // # of iterations till convergence
38 runtime.start();
39 var niter = 0;
40 var (js,jt) = (2,1);                // starting indices
41 while deltam >= epsilon do {        // check convergence
42     deltam = 0.0;
43     for b in 0..1 do {               // half-sweeps
44         if niter > 1 then {          // third or greater half-sweep
45             omega = 1.0/(1.0 - (rhoJ**2)*omega/4.0);
46         }
47         else if niter == 1 then {    // second half-step
48             omega = 1/(1.0 - (rhoJ**2)/2.0);
49         }
50         else {                     // first half-step
51             omega = 1.0;
52         }
53         halfsw(js);
54         js <=> jt;                // swap indices
55         niter += 1;
56     }
57     nc += 1;
58     deltam /= (Nn**2);
59 }
60 writeln("omega_f = ",omega_f);
61 writeln("omega =   ",omega);

```

```

62 runtime.stop();
63 writeln("nc = ",nc);
64 var sum = 0.0;
65 var mad = 0.0;
66 for i in 0..Nn do {
67     for j in 0..Nn do {
68         sum += phi[i,j];
69         var del = abs(phi[i,j] - ana(i,j));
70         mad += del;
71     }
72 }
73 mad /= ((Nn+1)**2);
74 writef("deltam = %8.5r\n",deltam);
75 writef("phiavg = %8.4dr\n",sum/((Nn+1)**2));
76 writef("mad    = %8.5r\n",mad);
77 writef("rtime   = %8.4dr\n",runtime.elapsed()/ut);
78 // -----
79 // --> initial guess
80 // -----
81 private inline proc IG(i,j: int): real {
82     const x = i*dl;
83     const y = j*dl;
84     if i == 0 then {
85         return y;
86     }
87     else if i == Nn then {
88         return 1.0;
89     }
90     else if j == 0 then {
91         return x;
92     }
93     else if j == Nn then {
94         return 1.0;
95     }
96     else {
97         return 0.50;
98     }
99 }
100 // -----
101 // --> the analytical solution
102 // -----
103 private inline proc ana(i,j): real {
104     const x = i*dl;
105     const y = j*dl;
106     return x + y - x*y;
107 }
108 // -----
109 // -->halfsw: a half-sweep
110 // -----
111 inline proc halfsw(
112     const in js: int           // starting j depends on js
113 ) {
114     for i in 1..Nn-1 do {
115         for j in js..Nn-1 by 2 do {
116             var phiavg = (phi[i+1,j]+phi[i-1,j]+phi[i,j-1]+phi[i,j+1])/4.0;
117             var deltaphi = omega*(phiavg - phi[i,j]);
118             phi[i,j] += deltaphi;
119             deltaphi = abs(deltaphi);
120             deltam += deltaphi;
121         }
122     }
123 }

```

In each iteration (counted by  $n_c$ )  $\omega$  is updated differently for the first and second half-steps (steps between consecutive half-sweeps), which is decided in lines 44–52. These two half-sweeps occur inside the **for** in line 43. The heart of the “staggering” update is programmed in procedure `halfsw` in line 112: it boils down to starting the  $j$  index either on 1 or 2, and incrementing it by 2. These indices are declared in line 40, and swapped in line 54.

The new approach is short of miraculous; here we compare in table 3.4 the performance of the serial programs `laplace-sor.chpl` and `laplace-asor.chpl`.

Tabela 3.4: Grid size  $N_n$ , number of iterations to convergence  $n_c$ , estimated  $\bar{u}$ , MAD and relative runtime  $t_r$  for the serial (`laplace-sor`) and accelerated serial (`laplace-asor`) versions of the solution of Laplace’s equation with SOR.

$N_n$	serial				accelerated serial			
	$n_c$	$\bar{u}$	MAD	$t_r$	$n_c$	$\bar{\phi}$	MAD	$t_r$ (s)
128	431	0.7500	$2.5625 \times 10^{-7}$	0.0174	364	0.7500	$3.5163 \times 10^{-8}$	0.0041
256	1934	0.7500	$1.5653 \times 10^{-6}$	0.3223	714	0.7500	$9.3864 \times 10^{-8}$	0.0267
512	6947	0.7500	$6.6456 \times 10^{-6}$	4.8341	1404	0.7500	$1.5544 \times 10^{-7}$	0.2587
1024	23955	0.7500	$2.7032 \times 10^{-5}$	67.6316	2759	0.7500	$2.1904 \times 10^{-7}$	2.0131
2048	80310	0.7499	$1.0864 \times 10^{-4}$	916.1190	5208	0.7500	$7.7417 \times 10^{-7}$	18.8378

What about parallelization? This is actually straightforward: comparing with the code from `laplace-sor-p.chpl`, we only need to change the procedure `halfsw`; therefore, we show only the modified part of `laplace-asor-p.chpl`:

Listagem 3.23: `laplace-asor-p` – Parallel solution of Laplace’s equation with accelerated successive over relaxation.

```

111 inline proc halfsw(                                // beginning of halfsw
112   const in js: int
113   )
114   var deltaxm: [1..Nn-1] real;
115   forall i in 1..Nn-1 do {
116     var dj = {js..Nn-1 by 2};
117     var deltaym: [dj] real;
118     foreach j in dj do {
119       var phiavg = (phi[i+1,j]+phi[i-1,j]+phi[i,j-1]+phi[i,j+1])/4.0;
120       var deltaphi = omega*(phiavg - phi[i,j]);
121       phi[i,j] += deltaphi;
122       deltaphi = abs(deltaphi);
123       deltaym[j] += deltaphi;
124     }
125     deltaxm[i] = (+ reduce deltaym);
126   }
127   deltam += (+ reduce deltaxm);
128 }                                                 // end of halfsw

```

And now we compare the serial and parallel implementations of the accelerated version in table 3.5.

Tabela 3.5: Grid size  $N_n$ , number of iterations to convergence  $n_c$ , estimated  $\bar{u}$ , MAD and relative runtime  $t_r$  for the serial (laplace-sor) and accelerated serial (laplace-asor) versions of the solution of Laplace's equation with SOR.

$N_n$	serial				accelerated serial			
	$n_c$	$\bar{u}$	MAD	$t_r$	$n_c$	$\bar{\phi}$	MAD	$t_r$ (s)
128	364	0.7500	$3.5163 \times 10^{-8}$	0.0041	359	0.7500	$3.3762 \times 10^{-8}$	0.0066
256	714	0.7500	$9.3864 \times 10^{-8}$	0.0267	709	0.7500	$9.0555 \times 10^{-8}$	0.0282
512	1404	0.7500	$1.5544 \times 10^{-7}$	0.2587	1399	0.7500	$1.5488 \times 10^{-7}$	0.1645
1024	2759	0.7500	$2.1904 \times 10^{-7}$	2.0131	2752	0.7500	$2.2240 \times 10^{-7}$	1.1350
2048	5208	0.7500	$7.7417 \times 10^{-7}$	18.8378	5209	0.7500	$7.5647 \times 10^{-7}$	8.4008