

EAMB7024 MÉTODOS NUMÉRICOS EM ENGENHARIA AMBIENTAL
— NOTAS DE AULA

Nelson Luís da Costa Dias

November 6, 2025

Versão: November 6, 2025T16:37:38

©Nelson Luís da Costa Dias, 2025. 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.

Contents

1 Linguagens de Programação	10
1.1 Antes de começar a trabalhar,	10
1.2 O 1º exemplo: a Conjectura de Collatz	11
1.2.1 Collatz em Fortran	11
1.2.2 Collatz em C	11
1.2.3 Collatz em Chapel	14
1.3 A quick and dirty random number generator	14
1.3.1 Random number generator in C	14
1.3.2 Random number generator in Fortran 90	16
1.3.3 Random number generator in Chapel	16
1.4 Newton-Raphson	18
1.4.1 Newton-Raphson in Fortran 90	18
1.4.2 Newton-Raphson in C	20
1.4.3 Newton-Raphson in Chapel	20
1.5 Integer and floating point binary representations	20
1.5.1 Two's complement	20
1.6 Floating point representation	23
1.7 The cubic root and Newton's fractal	24
1.8 A first look at parallelization	28
2 Métodos numéricos para o Cálculo de uma variável	30
2.1 Aproximações de diferenças finitas e suas ordens	30
2.2 Solução numérica de equações diferenciais ordinárias	31
2.3 Solução numérica; o método de Euler de ordem1	32
2.4 Um esquema de diferenças centradas, com tratamento analítico	34
2.5 A forma padrão $dy/dx = f(x, y)$	39
2.6 Um método de passos múltiplos: Adams-Bashforth	43
2.7 Interlúdio sobre arrays	47
2.8 O método de Runge-Kutta multidimensional	51
2.9 Problemas de valor de contorno em 1D	60
2.10 Trabalhos computacionais	71
3 Iterative methods and multigrid in one dimension	86
3.1 Introduction: a linear 1D problem	86
3.2 Multigrid	92
4 Solução numérica de equações diferenciais parciais	103
4.1 Advecção pura: a onda cinemática	103
4.2 Difusão pura	124
4.3 Difusão em 2 Dimensões: ADI, e equações elíticas	143
4.4 Trabalhos Computacionais	151

4.5 Successive over-relaxation	164
4.6 Really efficient Successive Over-Relaxation	167
5 Navier-Stokes	172
5.1 A grade escalonada e os balanços materiais de volumes finitos	172
5.2 Condições de contorno para u	183
5.3 Progressão no tempo	184
5.4 Solução iterativa da equação de Poisson com uma malha geral $\Delta x \neq \Delta y$	185
5.5 Poisson equation with Neumann BCs	190
5.6 Poisson with FFTs?	191
5.7 Índices	193
5.8 Marcha no tempo	194
Index	195
Referências Bibliográficas	195

List of Tables

2.1	Curva de remanso em canal trapezoidal – solução de Ven Te Chow	79
2.2	Parâmetros do modelo de Ludwig et al. (1978)	83
3.1	Number of intervals $n_g = 2^g$ at each resolution g	94
4.1	First values of m , n , their sum, and $(2m + 1)$ and $(2n + 1)$ in (4.77)–(4.78).	144
4.2	Grid size N_n , optimal over-relaxation parameter ω , number of iterations to convergence n_c , estimated \bar{u} , MAD and relative runtime t_r for the serial version of the solution of Laplace's equation with SOR.	167
4.3	Grid size N_n , optimal over-relaxation parameter ω , number of iterations to convergence n_c , estimated \bar{u} , MAD and relative runtime t_r for the serial (<code>Laplace-sor</code>) and accelerated serial (<code>laplace-asor</code>) versions of the solution of Laplace's equation with SOR. All cases were run with the optimum ω	170
4.4	Grid size N_n , optimal over-relaxation parameter ω , number of iterations to convergence n_c , estimated $\bar{\phi}$, MAD and relative runtime t_r for the accelerated serial (<code>laplace-asor</code>) and accelerated parallel (<code>laplace-asor-p</code>) versions of the solution of Laplace's equation with SOR.	171

List of Figures

1.1	Valores das iterações $f_k(n)$ da Conjectura de Collatz a partir de $n = 44371$	12
1.2	The 3 rd degree polynomial function $f(x) = x^3 - x^2 + 16x - 16$	20
1.3	The IEEE 754 single-precision floating point format.	23
1.4	Color codes ($1 + 0i = \text{red}$, $-1/2 + i\sqrt{3}/2 = \text{green}$, $-1/2 - i\sqrt{3}/2 = \text{blue}$) of each of the three roots of $z^3 - 1 = 0$ to which Newton-Raphson's method converges depending on the initial guess. The three roots are shown as small open circles on the unit circle $ z = 1$	27
2.1	Solução da equação (2.7) para $y(0) = 0$	32
2.2	Comparação da solução analítica da equação (2.7) com a saída de <code>sucesso.chpl</code> , para $\Delta x = 0,01$	35
2.3	Comparação da solução analítica da equação (2.7) com a saída de <code>sucesso.chpl</code> , para $\Delta x = 0,5$	36
2.4	Comparação da solução analítica da equação (2.7) com a saída de <code>succent.chpl</code> , para $\Delta x = 0,5$	38
2.5	Os métodos de Euler de ordens 1 e 2.	40
2.6	Comparação da solução analítica da equação (2.7) com a saída de <code>euler2.chpl</code> , para $\Delta x = 0,5$	44
2.7	Comparação da solução analítica da equação (2.7) com a saída de <code>rungk4.chpl</code> , para $\Delta x = 0,5$	44
2.8	Comparação da solução analítica da equação (2.7) com a saída de <code>adbash.chpl</code> , para $\Delta x = 0,5$	46
2.9	Solução numérica pelo Método de Runge-Kutta de um sistema de 2 equações diferenciais ordinárias	54
2.10	Solução numérica de (2.18)–(2.20).	64
2.11	Erro da solução numérica de (2.18)–(2.20) em função do número de pontos N da grade.	64
2.12	Solução numérica de (2.27)–(2.29) com uma discretização de ordem 1 para a condição de contorno direita.	67
2.13	Erro da solução numérica de (2.27)–(2.29) em função do número de pontos N da grade. O coeficiente angular da reta é ~ -1	67
2.14	Solução numérica de (2.27)–(2.29) com uma discretização de ordem 2 para a condição de contorno direita.	68
2.15	Erro da solução numérica de (2.27)–(2.29) em função do número de pontos N da grade. O coeficiente angular da reta é ~ -2	70
2.16	Um pêndulo (possivelmente) não-linear.	71
2.17	O período de um pêndulo não linear em função da amplitude inicial Θ_0 . A linha tracejada é o valor teórico $f(0) = 2\pi$ para oscilações de pequena amplitude.	74
2.18	Características geométricas de um canal.	78
2.19	Área molhada em função da distância de jusante do exemplo 10.1 de Chow (1959), calculados por Chow (círculos) <i>versus</i> resultados do método de Runge-Kutta (linha contínua).	80

2.20	Perda de carga em função da distância de jusante do exemplo 10.1 de Chow (1959), calculados por Chow (círculos) <i>versus</i> resultados do método de Runge-Kutta (linha contínua).	81
2.21	Velocidade em função da distância de jusante do exemplo 10.1 de Chow (1959), calculados por Chow (círculos) <i>versus</i> resultados do método de Runge-Kutta (linha contínua).	81
2.22	Cota em função da distância de jusante do exemplo 10.1 de Chow (1959), calculados por Chow (círculos) <i>versus</i> resultados do método de Runge-Kutta (linha contínua).	82
2.23	A traça (<i>Spruce budworm</i>) <i>Choristoneura orae</i> . Fonte: Wikipedia. ©entomart (http://www.entomart.be).	83
2.24	Simulação de eclosão de uma praga de traças, utilizando o modelo de Ludwig et al. (1978).	84
2.25	O oscilador não-linear de van der Pol ($\mu = 5$).	85
3.1	Solution of Helmholtz Equation in 1D.	88
3.2	Jacobi method: initial guess, analytical, and numerical solution.	91
3.3	Error of the Jacobi method as the iterations progress. Note the leveling of the error.	91
3.4	Pointwise error of the Jacobi method for the 10th, 100th and 250th iterations.	92
3.5	Multigrid static storage strategy.	95
3.6	The multigrid method for the Helmholtz 1D equation for a grid of 17 points ($N = 16$). The solid line is the analytical solution. All datapoints (except for (e)) are shown after 4 relaxations. (a): initial guess (note the high-frequency oscillations around the analytical solution). (b): downward solution for grid 0. (c): downward error corrections for grid 1. (d): downward error corrections for grid 2. (e): algebraic solution for the error in grid 3 using (3.7). (f): upward error correction for grid 2. (g): upward error correction for grid 1. (h): upward solution for grid 0 at the end of one V-cycle.	97
3.7	Solution of the Helmholtz 1D equation with the multigrid method after 4 V-cycles.	98
4.1	Solução analítica das equações 4.3–4.4.	104
4.2	Solução numérica produzida por <code>onda1d-ins.chpl</code> , para $t = 250\Delta t, 500\Delta t$ e $750\Delta t$	107
4.3	Solução numérica produzida por <code>onda1d_lax.chpl</code> , para $t = 500\Delta t, 1000\Delta t$ e $1500\Delta t$	111
4.4	Solução numérica produzida pelo esquema <i>upwind</i> , para $t = 250, 500$ e 750	113
4.5	Quick's interpolation scheme.	114
4.6	Solução numérica produzida pelo esquema QUICK, para $t = 250, 500$ e 750	118
4.7	As funções $\sin^2(\theta)$ e $\cos^2(\theta)$	122
4.8	A função definida em 4.44 (para $Fo = 1$)	122
4.9	Exemplo 4.1: interseção (em cinza escuro) de 32 regiões (em cinza-claro; incrementos de $\pi/32$) na equação (4.44) A parábola em preto representa o caso degenerado $\theta = 0$	123
4.10	Solução analítica da equação de difusão para $t = 0, t = 0,05, t = 0,10$ e $t = 0,15$	127
4.11	Solução numérica com o método explícito (4.54) (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.	131
4.12	Solução numérica com o método implícito (4.58) (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.	134
4.13	Solução numérica com o método de Crank-Nicholson ((4.64)) (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.	136
4.14	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 $t = 0,999$	141

4.15 Comparison between the analytical (mesh) and numerical (points; ADI) solutions of the two-dimensional diffusion equation.	150
4.16 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.	155
4.17 Parte real de $\phi(\zeta, \tau)$, $\tau = 5, 25, 50, 100$, e solução analítica.	157
4.18 Parte imaginária de $\phi(\zeta, \tau)$, $\tau = 5, 25, 50, 100$, e solução analítica.	157
4.19 Comparação do esquema numérico com a solução analítica de Boussinesq (1904). De cima para baixo, $\tau = 0,01, \tau = 0,1$, e $\tau = 1,0$	161
4.20 Comparação da solução numérica completa de (4.134)–(4.135) com a solução linearizada (4.143) para $\Phi_0 = 0$. De cima para baixo, $\tau = 0,01, \tau = 0,1$, e $\tau = 1,0$	162
4.21 Comparação da solução numérica completa de (4.134)–(4.135) com a solução linearizada (4.143) para $\Phi_0 = 0,25$. De cima para baixo, $\tau = 0,01, \tau = 0,1$, e $\tau = 1,0$	162
4.22 Comparação da solução numérica completa de (4.134)–(4.135) com a solução linearizada (4.143) para $\Phi_0 = 0,50$. De cima para baixo, $\tau = 0,01, \tau = 0,1$, e $\tau = 1,0$	163
4.23 Comparação da solução numérica completa de (4.134)–(4.135) com a solução linearizada (4.143) para $\Phi_0 = 0,75$	163
4.24 Running times t_r as a function of $n = N_n^2$ for the SOR method. The line is a parabola of the type $t_r = an^2$	168
4.25 Alternating grids in 2D for the SOR methods. Note that when we update the black points using (4.171), 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.	168
5.1 Balanços materiais de um escalar em uma grade escalonada	173
5.2 Balanço de quantidade de movimento em x em uma grade escalonada	176
5.3 Balanço de quantidade de movimento em y em uma grade escalonada	180

Listings

1.1	collatz.for — A Conjectura de Collatz em Fortran.	12
1.2	collatzf.out — Arquivo de saída de collatz.for (5 primeiras linhas)	12
1.3	collatz2.for — A Conjectura de Collatz em Fortran utilizando integer*2.	13
1.4	collatz2f.out — Arquivo de saída de collatz2.for	13
1.5	collatz.c — A Conjectura de Collatz em C.	13
1.6	collatz.chpl — A Conjectura de Collatz em Chapel.	14
1.7	qdran.h — Quick and Dirty Random Number Generator prototypes.	15
1.8	qdran.c — Quick and Dirty Random Number Generator functions.	15
1.9	ctestran.c — Test of the Quick and Dirty Random Number Generator.	16
1.10	ftestran.f90 — Test of uranqd using the C library qdran.c.	17
1.11	qdran.chpl — Random number generator in Chapel	18
1.12	qdran.chpl — Testing program for the random number generator in Chapel	19
1.13	fnewton.f90 — The Newton-Raphson method in Fortran 90.	19
1.14	cnewton.c — The Newton-Raphson method in C.	21
1.15	chplnewton.chpl — The Newton-Raphson method in Chapel.	21
1.16	binaryint.chpl — Binary representations of 14 and -14.	22
1.17	binaryint.out — Arquivo de saída de binaryint.chpl	22
1.18	tableint.chpl — Binary representations of 14 and -14.	23
1.19	znewton.chpl — Obtain a single root of $z^3 - 1 = 0$ from an initial guess zinit.	25
1.20	zxynewton.plt — script to plot the root of $z^3 - 1 = 0$ found by Newton-Raphson's method in the complex plane.	25
1.21	zxynewton.chpl — Obtain the root of $z^3 - 1 = 0$ for each pixel of a 4×4 square in the complex plane and color-code it according to the root found: $1 + 0i = \text{red}, -1/2 + i\sqrt{3}/2 = \text{green}, -1/2 - i\sqrt{3}/2 = \text{blue}$.	26
1.22	zxynewton-p.chpl — Parallel version to obtain the root of $z^3 - 1 = 0$ for each pixel of a 4×4 square in the complex plane and color-code it according to the root found: $1 + 0i = \text{red}, -1/2 + i\sqrt{3}/2 = \text{green}, -1/2 - i\sqrt{3}/2 = \text{blue}$.	29
2.1	fracasso.chpl — Um programa com o método de Euler que não funciona	33
2.2	fracasso.out — Saída do programa fracasso.	33
2.3	succeso.chpl — Um programa com o método de Euler que funciona	35
2.4	succent.chpl — Método de Euler implícito	37
2.5	euler2 — Um método explícito de ordem 2	41
2.6	rungek4 — Método de Runge-Kutta, ordem 4	42
2.7	adbash — Solução de (2.7) com Adams-Bashforth	45
2.8	learnarray — Alguns aspectos de arrays em Chapel	47
2.9	ada — Implementa um vetor (vec) que contorna a limitação de arrays genéricos em argumentos de rotinas.	48
2.10	runkutr — Implementa o método de Runge-Kutta multidimensional.	52
2.11	rktestr — Resolve um sistema de EDOs com o método de Runge-Kutta multidimensional.	53
2.12	oncin.chpl — Propagação de cheia com o método de Runge-Kutta.	57
2.13	tridiag.chpl — Exporta uma rotina que resolve um sistema tridiagonal.	62

2.14	<code>edo-l1.chpl</code> — Solução do problema de valor de contorno (2.18)–(2.20).	63
2.15	<code>edo-l2.chpl</code> — Solução do problema de valor de contorno (2.27)–(2.29).	66
2.16	<code>edo-l2b.chpl</code> — Solução do problema de valor de contorno (2.27)–(2.29) com um esquema de ordem 2 para a condição de contorno direita.	69
3.1	<code>helm1d.chpl</code> — Solution of Helmholtz equation in 1D.	87
3.2	<code>jacob1d.chpl</code> — Solution of Helmholtz equation with Jacobi Method in 1D.	89
3.3	<code>vcyle.chpl</code> — Solution of the 1D Helmholtz equation using V-cycles.	96
4.1	<code>onda1d_ins.chpl</code> — Solução de uma onda cinemática 1D com um método explícito instável	105
4.2	<code>surf1d_ins.chpl</code> — Seleciona alguns intervalos de tempo da solução numérica para plotagem	106
4.3	<code>onda1d_lax.chpl</code> — Solução de uma onda cinemática 1D com o método de Lax	110
4.4	<code>surf1d_lax.chpl</code> — Seleciona alguns intervalos de tempo da solução numérica para plotagem	112
4.5	<code>quick_grid.chpl</code> — A stable grid for QUICK's solution of (4.1)	115
4.6	<code>onda1d_quick.chpl</code> — Solution of a 1D kinematic wave using QUICK	116
4.7	<code>surf1d_quick.chpl</code> — Seleciona alguns intervalos de tempo da solução numérica com QUICK para plotagem	117
4.8	<code>difusao1d-ana.chpl</code> — Solução analítica da equação da difusão	125
4.9	<code>divisao1d-ana.py</code> — Seleciona alguns instantes de tempo da solução analítica para visualização	126
4.10	<code>difusao1d-exp.py</code> — Solução numérica da equação da difusão: método explícito.	129
4.11	<code>divisao1d-exp.chpl</code> — Seleciona alguns instantes de tempo da solução analítica para visualização	130
4.12	<code>difusao1d-imp.chpl</code> — Solução numérica da equação da difusão: método implícito.	133
4.13	<code>difusao1d-ckn.chpl</code> — Solução numérica da equação da difusão: esquema de Crank-Nicholson.	135
	<code>sana.max</code>	139
4.14	Implementação de um esquema numérico implícito para a equação da difusão-advecção.	140
4.15	<code>diffgrid2d.chpl</code> — Grid constants for solutions of the two-dimensional diffusion equation.	144
4.16	<code>diffana2d.chpl</code> — Analytical solution of the two-dimensional diffusion equation.	145
4.17	<code>diffadi2d.chpl</code> — Numerical (ADI) solution of the two-dimensional diffusion equation.	147
4.18	<code>diffadi2d-p</code> — Parallel implementation of the ADI method: sweep in the x direction.	149
4.19	<code>diffview2d.chpl</code> — Read a chosen timestep from a binary file and print it in a text file.	149
4.20	<code>laplace-sor.chpl</code> — Solution of Laplace's equation with successive over relaxation.	165
4.21	<code>laplace-asor-s</code> — Solution of Laplace's equation in 2D over staggered grids using Chebyshev acceleration.	167
4.22	<code>laplace-asor-p</code> — Parallel solution of Laplace's equation with accelerated successive over relaxation.	170

1

Linguagens de Programação

Neste curso quase 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 compilada e com tipagem estática 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
Fortran	Linux, MacOs, Windows	https://gcc.gnu.org/wiki/GFortran
C	Linux, MacOs, Windows	(Variável: parta de https://gcc.gnu.org/)
Chapel	Linux, MacOs, Windows(?)	https://chapel-lang.org/

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 instalada (pelo menos) uma das 3 linguagens de programação acima.

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.

2025-07-24T14:33:29. Um editor que tem tido ampla aplicação é o VS-code da Microsoft. Você pode encontrá-lo em <https://code.visualstudio.com/>, e ele pode ser instalado tanto em Windows

quanto em Linux, com facilidade. Uma grande vantagem é que ele tem coloração de sintaxe para todas as 3 linguagens acima.

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

- 1) Criar o arquivo com o programa em Fortran, C ou 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 – O 1º exemplo: a Conjectura de Collatz

Dado um número inteiro positivo n , considere a função

$$f(n) = \begin{cases} n/2, & n \bmod 2 = 0, \\ 3n + 1, & n \bmod 2 = 1, \end{cases} \quad (1.1)$$

onde $x \bmod y$ indica o resto da divisão inteira de x por y . A conjectura de Collatz é que em número finito de operações, $\forall n > 1$,

$$f(f(f \dots f(n))) = 1.$$

ou seja: a sequência sempre termina em 1. A conjectura nunca foi provada.

1.2.1 – Collatz em Fortran

Nós agora vamos programar a conjectura de Collatz em Fortran, C e Chapel, nesta ordem. A listagem 1.1 mostra um primeiro programa em Fortran-77.

O programa é quase auto-explicativo: ele abre (open) um arquivo de saída com o nome `collatz-f.out` e imprime duas colunas: o número da iteração, e o valor de n resultante.

A listagem 1.2 mostra as primeiras linhas de `collatz-f.out`.

O gráfico dos valores de cada iteração é bem interessante, como mostrado na figura 1.1.

Alguns resultados supreendentes podem ser encontrados modificando-se ligeiramente o programa `collatz.for`. Por exemplo, considere a versão `collatz2.for` na listagem 1.3.

A sua saída é mostrada na listagem 1.4. Agora, nós compilamos o programa *impedindo* a detecção de um certo tipo de erro, com

```
gfortran -fno-range-check collatz2.for -o collatz2
```

Consequentemente, a linha $n = 44371$ gerou um *overflow* silencioso, porque o maior inteiro de 2 bytes possível é $32767 < 44371$, e o resultado foi interpretado como -21165 .

1.2.2 – Collatz em C

A versão do mesmo algoritmo em C é mostrada na listagem 1.5.

Uma comparação das saídas `collatzf.out` e `collatzc.out` mostra que elas são iguais.

Listing 1.1: *collatz.for* — A Conjectura de Collatz em Fortran.

```

1 ! -----
2 ! collatz.for: the Collatz conjecture
3 !
4 ! Nelson Luis Dias
5 ! 2025-08-03T10:42:20
6 ! -----
7     program collatz
8     integer k,n
9     open(unit=6,form='formatted',file='collatzf.out')
10    n = 44371
11    k = 0
12    write(6,100) k, n
13    do while (n .gt. 1)
14        if (mod(n,2) .eq. 0) then
15            n = n/2
16        else
17            n = 3*n + 1
18        endif
19        k = k + 1
20        write(6,100) k, n
21    enddo
22 00100 format(i8,X,i8)
23  close(unit=6)
24  end

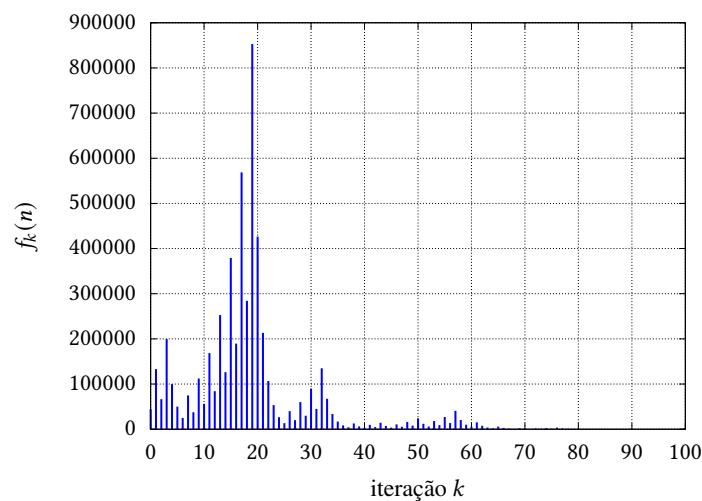
```

Listing 1.2: *collatzf.out* — Arquivo de saída de *collatz.for* (5 primeiras linhas)

```

.....0....44371
.....1....133114
.....2....66557
.....3....199672
.....4....99836

```

Figure 1.1: Valores das iterações $f_k(n)$ da Conjectura de Collatz a partir de $n = 44371$.

Listing 1.3: collatz2.for — A Conjectura de Collatz em Fortran utilizando integer*2.

```

1 ! -----
2 ! collatz2.for: the Collatz conjecture, but with integer*2
3 !
4 ! Nelson Luís Dias
5 ! 2025-07-24
6 !
7     program collatz2
8     integer*2 k,n
9     open(unit=6,form='formatted',file='collatz2f.out')
10    n = 44371
11    k = 0
12    write(6,100) k, n
13    do while (n .gt. 1)
14        if (mod(n,2) .eq. 0) then
15            n = n/2
16        else
17            n = 3*n + 1
18        endif
19        k = k + 1
20        write(6,100) k, n
21    enddo
22 00100 format(i8,X,i8)
23  close(unit=6)
24  end

```

Listing 1.4: collatz2f.out — Arquivo de saída de collatz2.for

```
.....0...-21165
```

Listing 1.5: collatz.c — A Conjectura de Collatz em C.

```

1 // -----
2 // collatz.c: the Collatz conjecture
3 //
4 // Nelson Luís Dias
5 // 2025-07-23
6 //
7 #include <stdio.h>
8 int main(void) {
9     int k, n;
10    FILE *fou;
11    fou = fopen("collatzc.out","wt");
12    n = 44371;
13    k = 0;
14    fprintf(fou,"%8d.%8d\n",k,n);
15    while (n > 1) {
16        if (n % 2 == 0) {
17            n /= 2;
18        }
19        else {
20            n = 3*n + 1;
21        }
22        k += 1;
23        fprintf(fou,"%8d.%8d\n",k,n);
24    }
25    fclose(fou);
26    return 0;
27 }
```

Listing 1.6: `collatz.chpl` — A Conjectura de Collatz em Chapel.

```

1 // -----
2 // collatz.chpl: the Collatz conjecture
3 //
4 // Nelson Luís Dias
5 // 2025-07-23
6 // -----
7 use IO only openWriter;
8 var k, n: int ;
9 const fou = openWriter("collatzchpl.out");
10 n = 44371;
11 k = 0;
12 fou.writef("%8i.%8i\n",k,n);
13 while n > 1 do {
14     if n % 2 == 0 then {
15         n /= 2;
16     }
17     else {
18         n = 3*n + 1;
19     }
20     k += 1;
21     fou.writef("%8i.%8i\n",k,n);
22 }
23 fou.close();

```

1.2.3 – Collatz em Chapel

A versão do mesmo algoritmo em C é mostrada na listagem 1.6.

Uma comparação das saídas `collatzf.out` e `collatzchpl.out` novamente mostra que elas são iguais.

1.3 – A quick and dirty random number generator

This section follows the “Quick and Dirty” Random Number Generator presented by Press et al. (1992, p. 284, Eq. 7.1.6) — see “An Even Quicker Generator”. It consists simply of

$$I_{j+1} = aI_j + c \quad (1.2)$$

with *unsigned 32-bit integers*, $a = 1664525$, $c = 1013904223$, and letting the operations overflow silently. The language that overflows and underflows silently by excellence is C, so we will start with it.

In modern computers and compilers (as of 2025), an *unsigned int* in C uses 4 bytes, but in 1992 it usually occupied only 2. Our implementation, therefore, will use a specific 32-bit *unsigned int* (`uint32_t` from `<stdint.h>`) whereas Press et al. (1992) used *unsigned long*.

The sequence of numbers is not really random: it is obviously deterministic starting with a *seed* I_0 . We need three functions: the seeding function (`seedqd`), the function `ranqd` that iterates (1.2), and a function `uranqd` that converts the integer I_{j+1} into a floating point value, representing a random variable distributed uniformly between 0 and 1 (hence the *u* in `uranqd`). The prototypes are shown in the file `qdran.h`, in listing 1.7.

1.3.1 – Random number generator in C

The implementation itself, in file `qdran.c`, is given in listing 1.8: it is our first example of a *library*.

How good is `uranqd`? It should at least come close to the theoretical mean and variance of a uniform (truly) random variable between 0 and 1:

$$\mu_u = \int_0^1 u du = \frac{1}{2}, \quad (1.3)$$

Listing 1.7: qdran.h — Quick and Dirty Random Number Generator prototypes.

```

1 #include <stdint.h>
2 uint32_t ranqd(void);
3 double uranqd(void);
4 void seedqd(uint32_t seed);

```

Listing 1.8: qdran.c — Quick and Dirty Random Number Generator functions.

```

1 #include <stdint.h>
2 #include "qdran.h"
3 // -----
4 // ==> qdran: a library for random number generation
5 // -----
6 static uint32_t next = 1;    // next exists between calls
7 //
8 // --> ranqd: generates the next pseudorandom number from current
9 // -----
10 uint32_t ranqd(void) {
11     next = next*1664525 + 1013904223;
12     return next;
13 }
14 //
15 // --> uranqd: normalizes to real between 0.0 and 1.0 (generates a
16 //      "uniform" variable ~U(0,1) )
17 //
18 double uranqd(void) {
19     next = next*1664525 + 1013904223;
20     return ((double) (next))/UINT32_MAX ;
21 }
22 //
23 // --> seedqd: seeds the sequence, changing global variable next
24 //
25 void seedqd(uint32_t seed) {
26     next = seed;
27 }

```

Listing 1.9: ctestran.c — Test of the Quick and Dirty Random Number Generator.

```

1 #include <stdio.h>
2 #include "qdran.h"
3 #define N 1000000
4 int main(void) {
5     double u[N];
6     double mu, sig2;
7     double s1 = 0.0;
8     double s2 = 0.0;
9     seedqd(0);
10    for (long i = 0; i < N; i++) {
11        u[i] = uranqd();
12        s1 += u[i];
13    }
14    mu = s1/N;
15    for (long i = 0; i < N; i++) {
16        s2 += (u[i] - mu)*(u[i]-mu);      // alas, no exponential operator in C
17    }
18    sig2 = s2/N;
19    printf("sample mean = %8.4lf\n",mu);
20    printf("sample var = %8.4lf\n",sig2);
21    return 0;
22 }
```

$$\sigma_u^2 = \int_0^1 (u - \mu_u)^2 du = \frac{1}{12}. \quad (1.4)$$

We test this in program `ctestran.c`, shown in listing 1.9.

The result is close to the theoretical values, but it takes a long sequence (1,000,000 pseudorandom numbers).

```
$ gcc ctestran.c qdran.o -o ctestran
$ ./ctestran
sample mean = 0.5005
sample var = 0.0834
```

1.3.2 – Random number generator in Fortran 90

We can now write a test program in Fortran. However, we will not be able to write the random number generator in Fortran itself, because it does not have unsigned integers (still; one day Fortran will have everything). The solution is to link the Fortran program to the `qdran` C library. The way to do it is a little bit complicated. The best route in 2025 is to use the modern Fortran standards (Fortran 90, 95, 2003, 2008, 2018; from here on, we will call it “Fortran 90”, but we will mean the latest standard (currently 2018)). The program is shown in listing 1.10.

The “new” Fortran is verbose; the need to link the program with a C library increases the verbosity even further. Although Fortran is still very much alive and well, the price paid has been a more complicated syntax.

To compile and run the program, we need:

```
$ gfortran ftestran.f90 qdran.o -o ftestran
$ ./ftestran
```

The output is identical to that from `ctestran.c`.

1.3.3 – Random number generator in Chapel

It is possible to use the C library `qdran.c` in Chapel as well, but it is simpler to re-implement it in Chapel. We do it in listing 1.11. The testing program, `chpltestran.chpl`, is shown in listing 1.12.

Listing 1.10: ftestran.f90 — Test of uranqd using the C library qdran.c.

```
1 program ftestran
2   use iso_c_binding
3   implicit none
4   interface
5     subroutine seedqd(seed) bind(c,name="seedqd")
6       use iso_c_binding
7       integer(c_int32_t), value :: seed
8     end subroutine seedqd
9     function uranqd() result(back) bind(c,name="uranqd")
10       real*8 back
11     end function uranqd
12   end interface
13   integer*4, parameter :: N = 1000000
14   integer*4 :: i
15   real*8 :: mu, sig2
16   real*8 :: s1 = 0.0
17   real*8 :: s2 = 0.0
18   real*8 :: u(1:N)
19   call seedqd(0)
20   do i = 1,N
21     u(i) = uranqd()
22     s1 = s1 + u(i)
23   end do
24   mu = s1/N
25   do i = 1,N
26     s2 = s2 + (u(i)-mu)**2
27   enddo
28   sig2 = s2/N
29   write(6,100) mu
30   write(6,200) sig2
31 100 format("sample.mean_=_",f8.4)
32 200 format("sample.var_=_",f8.4)
33 end program ftestran
```

Listing 1.11: qdran.chpl — Random number generator in Chapel

```

1 // =====
2 // ==> qdran: a library for random number generation
3 // =====
4 private var next: uint(32) = 1;      // next exists between calls
5 private param maxx = max(uint(32)); // to normalize between 0 and 1
6 //
7 // --> ranqd: generates the next pseudorandom number from current
8 //
9 proc ranqd(): uint(32) {
10    next = next*1664525 + 1013904223;
11    return next;
12 }
13 //
14 // --> uranqd: normalizes to real between 0.0 and 1.0 (generates a
15 //   "uniform" variable ~U(0,1) )
16 //
17 proc uranqd(): real {
18    next = next*1664525 + 1013904223;
19    return (next:real)/maxx ;
20 }
21 //
22 // --> seedq: seeds the sequence, changing global variable next
23 //
24 proc seedq(in seed: uint(32)) {
25    next = seed;
26 }
```

These listings show some advantages of Chapel, purely from the point of view of style: it is arguably the smallest in terms of number of lines, and one can also make the argument that its syntax is more elegant (although this is somewhat subjective, and the author must confess his bias towards Chapel).

1.4 – Newton-Raphson

Consider the function

$$f(x) = (x^2 + 16)(x - 1) = x^3 - x^2 + 16x - 16. \quad (1.5)$$

Although there are analytical equations for the roots of $f(x) = 0$, it is often much more expedient to find an approximate root numerically. Note that there are 3 roots, but 2 of them are complex. Therefore, we search for the real root with the Newton-Raphson method, whose basis is a simple Taylor expansion, as follows:

$$\begin{aligned} f(x_{n+1}) &\approx f(x_n) + f'(x_n)(x_{n+1} - x_n) = 0, \\ x_{n+1} - x_n &= -\frac{f(x_n)}{f'(x_n)}, \\ x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)}. \end{aligned} \quad (1.6)$$

We need to iterate 1.6 until two successive estimates are close together (or until $f(x_{n+1})$ is close enough to 0). Let us do it (again) in Fortran 90, C and Chapel.

1.4.1 – Newton-Raphson in Fortran 90

The Fortran 90 implementation is shown in listing 1.13, explicitly for $f(x)$ given by (1.5). The geometric interpretation of the method is given by the red lines in Figure 1.2.

Listing 1.12: qdran.chpl — Testing program for the random number generator in Chapel

```

1 use qdran;
2 const N = 1000000;
3 var u: [1..N] real;
4 var mu, sig2: real;
5 var s1 = 0.0;
6 var s2 = 0.0;
7 seedqd(0);
8 for i in 1..N do {
9     u[i] = uranqd();
10    s1 += u[i];
11 }
12 mu = s1/N;
13 for i in 1..N do {
14     s2 += (u[i] - mu)**2;
15 }
16 sig2 = s2/N;
17 writeln("sample_mean=%8.4dr\n",mu);
18 writeln("sample_var=%8.4dr\n",sig2);

```

Listing 1.13: fnewton.f90 — The Newton-Raphson method in Fortran 90.

```

1 program fnewton
2   implicit none
3   real*8, parameter :: eps = 1.0e-6
4   real*8 :: x,x0
5   real*8 :: delta = eps
6   real*8 :: fx0
7   open(unit=1,form='formatted',file='fnewton.out')
8   x0 = -4.0
9   do while (delta .ge. eps)
10    fx0 = f(x0)
11    x = x0 - fx0/fl(x0)
12    delta = abs(x - x0)
13    write(1,100) x0,0.0,0.0
14    write(1,100) x0,fx0,delta
15    x0 = x
16  enddo
17 100 format(3(f12.6,2x))
18  write(6,200) x
19 200 format("root=%",f8.4)
20  close(1)
21 contains
22   function f(xin) result(back)
23     real*8, intent(in) :: xin
24     real*8 :: back
25     back = (xin**2 + 16)*(xin-1)
26   end function f
27   function fl(xin) result(back)
28     real*8, intent(in) :: xin
29     real*8 :: back
30     back = 3*xin**2 - 2*xin + 16
31   end function fl
32 end program fnewton

```

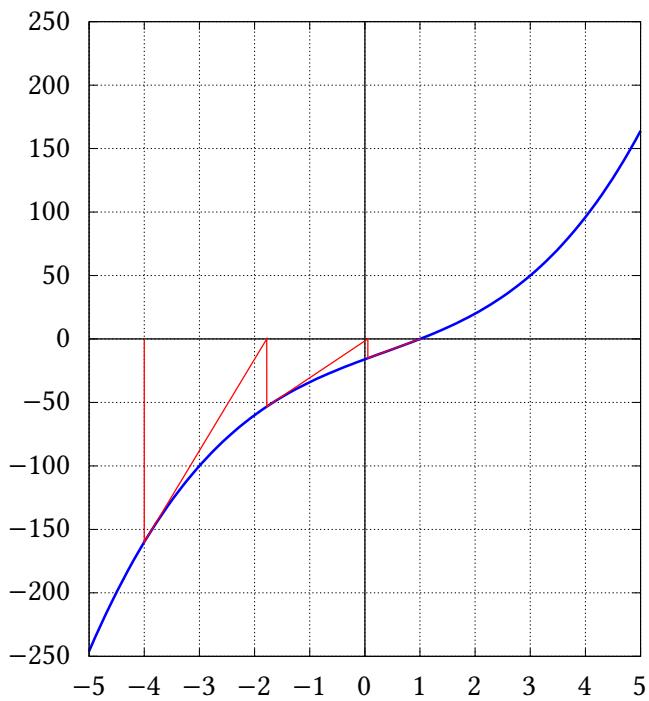


Figure 1.2: The 3rd degree polynomial function $f(x) = x^3 - x^2 + 16x - 16$.

1.4.2 – Newton-Raphson in C

The same program in C is shown in listing 1.14

1.4.3 – Newton-Raphson in Chapel

Finally, we do it in Chapel, in listing 1.15.

We will now proceed *using Chapel only*: it is one of the most modern, and one of the few that allows for parallel processing.

1.5 – Integer and floating point binary representations

In the examples above, we skimmed an important question: in practice, numbers in a computer have a concrete representation, and we need to understand it in order to avoid mistakes and problems.

There are really only two important formats for us: *integer* and *floating-point*. Let us start with the simplest: integer. Consider what is now universally the smallest amount of memory that a computer can address: the *byte*. It consists of 8 bits, and therefore there are $2^8 = 256$ states that it can represent. What numbers can a byte represent? If we elect to represent only positive integers, then the answer is straightforward: a byte can represent the positive integers 0 through 255.

1.5.1 – Two’s complement

What about negative integer numbers? In this case, we need to reserve a bit to represent the sign. A naive choice then reserves 1 bit for the sign, and the remaining 7 bits to represent 0–127 (2^7). However, in this case clearly we will have *two* representations for zero: +0 and −0. Therefore, with a little ingenuity, we can squeeze in one more value, and represent the numbers −128 to +127. This is called “Two’s complement”, and here is how it is done.

First, positive values will be represented with up to the 7 rightmost bits. For example, to represent 14 we use

[00001110].

Listing 1.14: cnewton.c — The Newton-Raphson method in C.

```

1 #include <stdio.h>
2 #include <math.h>
3 #define eps 1.0e-6
4 double f(double xin) {
5     return (pow(xin,2) + 16)*(xin-1);
6 }
7 double fl(double xin) {
8     return 3*pow(xin,2) - 2*xin + 16;
9 }
10 int main(void) {
11     FILE *fou;
12     double x,x0;
13     double delta = eps;
14     double fx0;
15     fou = fopen("cnewton.out","wt");
16     x0 = -4.0;
17     while (delta >= eps) {
18         fx0 = f(x0);
19         x = x0 - fx0/fl(x0);
20         delta = fabs(x - x0);
21         fprintf(fou,"%12.6lf..%12.6lf..%12.6lf\n",x0,0.0,0.0);
22         fprintf(fou,"%12.6lf..%12.6lf..%12.6lf\n",x0,fx0,delta);
23         x0 = x;
24     }
25     printf("root = %8.4lf\n",x);
26     fclose(fou);
27 }
```

Listing 1.15: chplnewton.chpl — The Newton-Raphson method in Chapel.

```

1 use IO only openWriter;
2 const eps = 1.0e-6;
3 const fou = openWriter("chplnewton.out");
4 var x,x0: real;
5 var delta = eps;
6 var fx0: real;
7 x0 = -4.0;
8 while (delta >= eps) {
9     fx0 = f(x0);
10    x = x0 - fx0/fl(x0);
11    delta = abs(x - x0);
12    fou.writef("%12.6dr..%12.6dr..%12.6dr\n",x0,0.0,0.0);
13    fou.writef("%12.6dr..%12.6dr..%12.6dr\n",x0,fx0,delta);
14    x0 = x;
15 }
16 writef("root = %8.4dr\n",x);
17 fou.close();
18 proc f(const in xin: real): real {
19     return (xin**2 + 16)*(xin-1);
20 }
21 proc fl(const in xin:real): real {
22     return 3*xin**2 - 2*xin + 16;
23 }
```

Listing 1.16: `binaryint.chpl` — Binary representations of 14 and -14 .

```

1 var i : int(8) = 14;
2 var u : uint(8) = 14;
3 writeln("-" * 20);
4 writef("%08bi\n", i);
5 writef("%08bu\n", u);
6 writeln("-" * 20);
7 i *= -1;
8 writef("%08bi\n", i);
9 u = i;
10 writef("%08bu\n", u);

```

Listing 1.17: `binaryint.out` — Arquivo de saída de `binaryint.chpl`

```

00001110
00001110
-----
-0001110
11110010

```

We use two's complement to represent negative integers. Consider -14 . First we “flip” the bits (we perform a bitwise not):

$[11110001]$.

Next, we add 1:

$$\begin{array}{r}
 11110001 \\
 + 00000001 \\
 \hline
 = 11110010
 \end{array}$$

Hence, the bitwise representation of -14 (for an 8-bit signed integer!) is

$[11110010]$.

Binary representations of 14 and -14 are shown in program `binaryint.chpl`, shown in listing 1.16. The output of `binaryint.chpl` is shown in listing 1.17.

The beauty of two's complement is that subtracting two quantities is turned into a regular sum; for example, $30 - 14 = 16$:

$$\begin{array}{r}
 00011110 \\
 + 11110010 \\
 \hline
 = 100010000
 \end{array}$$

But this sum overflows (the result has 9 bits) and the leftmost 1 is lost. What is left is the binary representation of 16:

$[00010000] \blacksquare$

More generally, if $b_{N-1}b_{N-2}\dots b_1b_0$ is the binary representation of a signed integer of width N bits (above, $N = 8$), then the value of the integer is

$$I = -b_{N-1} \times 2^{N-1} + b_{N-2} \times 2^{N-2} + \dots + b_1 \times 2^1 + b_0 \times 2^0. \quad (1.7)$$

The smallest representable integer is -2^{N-1} , and the largest is $2^{N-1} - 1$.

Finally, we present a small program that generates a table with the binary representation of all signed 8-bit integers from -128 to $+127$. Note how the program silently overflows the variable `i` and how this

Listing 1.18: `tableint.chpl` — Binary representations of 14 and -14.

```

1 use IO only openWriter;
2 const fou = openWriter("tableint.out");
3 var i : int(8) = 14;
4 var u : uint(8) = 14;
5 i = -128;
6 // -----
7 // note that addition of int(8) is cyclic (no integer overflow)!
8 // -----
9 do {
10     u = i;
11     fou.writef("%4i.%08bu\n", i, u);
12     i += 1;
13 } while i != -128;

```

can be used to end the **do ... while** loop. The table is too long to present here: the reader can run the program (`tableint.chpl`) given in listing 1.18 and edit the output `tableint.out` to inspect the table.

1.6 – Floating point representation

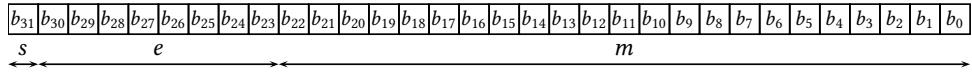


Figure 1.3: The IEEE 754 single-precision floating point format.

The IEEE 754 single-precision floating point representation of 32 bits consists of 1 sign bit, 8 bits for the exponent, and 23 bits for the *mantissa* such that $1 + 8 + 23 = 32$ (see figure 1.3). These bits are stored as $b_{31}b_{30}b_{29}\dots b_2b_1b_0$, $b_i = 0$ or 1 , interpreted as follows:

b_{31} is the sign bit s .

$b_{30}b_{29}\dots b_{23}$ is the exponent e .

$b_{22}b_{21}\dots b_0$ is the mantissa m .

Their values as calculated as

$$s = (-1)^{b_{31}}, \quad (1.8)$$

$$e = b_{23} \times 2^0 + b_{24} \times 2^1 + \dots + b_{30} \times 2^7, \quad (1.9)$$

$$m = 1 + \sum_{k=1}^{23} b_{23-k} \times 2^{-k} \quad (1.10)$$

Moreover, the exponent is limited to the range $1 \leq e \leq 254$; the values 0 and 255 have a special meaning. 255 means either NAN ($e = 255$ and non-zero mantissa) or INF ($e = 255$ and zero mantissa), whereas 0 indicates *subnormal* numbers associated with *underflow*. The value of a normal floating point *number* x then is

$$x = (-1)^s \times 2^{e-127} \times m. \quad (1.11)$$

We can now calculate the minimum, maximum and smallest 32-bit floating point number that is representable:

$$f_{min} = - \left[1 + \sum_{k=1}^{23} 2^{-k} \right] \times 2^{254-127} = -3.4028235 \times 10^{+38}, \quad (1.12)$$

$$f_{max} = + \left[1 + \sum_{k=1}^{23} 2^{-k} \right] \times 2^{254-127} = +3.4028235 \times 10^{+38}, \quad (1.13)$$

$$f_\delta = +2^{1-127} = +1.1754944 \times 10^{-38}. \quad (1.14)$$

1.7 – The cubic root and Newton’s fractal

The previous sections made a case: it is not too different to express an algorithm in different programming languages, and in this sense learning a new language, once another one is already known, is not too difficult either. In fact, in this section we will meet a problem that lends itself to very strong parallelizing.

Consider then the cubic roots of 1. Of course, $1^3 = 1$, so that 1 is a cubic root. But there are other two, complex roots. Put

$$\begin{aligned} w &= 1 + 0i = e^{2k\pi i}, \quad k = 0, 1, 2, \\ z &= re^{\theta i}, \\ z^3 &= w, \\ r^3 e^{3\theta i} &= 1 e^{2k\pi i}, \Rightarrow \\ r^3 &= 1, \quad r = 1, \\ \theta &= \frac{2k\pi}{3}; \\ \theta_0 &= 0, \\ \theta_1 &= \frac{2\pi}{3} = 120^\circ, \\ \theta_2 &= \frac{4\pi}{3} = 240^\circ. \end{aligned}$$

Hence, the three cubic roots of 1 are

$$\begin{aligned} z_1 &= 1, \\ z_2 &= e^{2\pi i/3}, \\ z_3 &= e^{4\pi i/3}. \end{aligned}$$

The interesting question is: can we find them with the Newton-Raphson method? Let us try. We want to find the roots of

$$\begin{aligned} f(z) &= 0, \\ f(z) &= z^3 - 1. \end{aligned}$$

We need the derivative,

$$f'(z) = 3z^2.$$

The iteration is

$$z_{n+1} = z_n - \frac{z_n^3 - 1}{3z_n^2}.$$

We write a first program, `znewton.chpl` (listing 1.19, where the initial guess `zinit` is a *configuration constant*, that can be changed in the command line. Running the program with three different initial guesses finds the three complex roots:

```
$ ./znewton --zinit="1.0+1i"
cubic root of 1 = 1.0000 - 0.0000i
$ ./znewton --zinit="-0.5+1i"
cubic root of 1 = -0.5000 + 0.8660i
$ ./znewton --zinit="-0.5-1i"
cubic root of 1 = -0.5000 - 0.8660i
```

Listing 1.19: znewton.chpl — Obtain a single root of $z^3 - 1 = 0$ from an initial guess zinit.

```

1 const eps = 1.0e-6;
2 config const zinit = 1.0 + 1.0i;
3 var fz0, z, z0: complex;
4 var delta = eps;
5 z0 = zinit;
6 while (delta >= eps) {
7     fz0 = f(z0);
8     z = z0 - fz0/fl(z0);
9     delta = abs(z - z0);
10    z0 = z;
11 }
12 writef("cubic_root_of_1=%8.4dz\n",z);
13 proc f(const in zin: complex): complex {
14     return (zin**3 - 1.0);
15 }
16 proc fl(const in zin:complex): complex {
17     return (3*zin**2);
18 }
```

But how exactly does this work? How does the initial guess determine the root that is found? To find an answer, we adopt a brute-force approach: we scan a 4×4 square around the center of the complex plane: we start with the initial guess at each of these points (within a given finite resolution!) and we assign the point a color: red if the root found is $1 + 0i$, green if it is $-1/2 + i\sqrt{3}/2$ and blue if it is $-1/2 - i\sqrt{3}/2$. This is program zxynewton.chpl, in listing 1.21. The output, zxynewton.out, is a very large file with three columns: the coordinates x, y of each point in the square, and a color code for the root that was found (0 = red, 1 = green, 2 = blue). The output file can be converted to a png image using (for example) the plotting program Gnuplot. Here is the gnuplot script:

Listing 1.20: zxynewton.plt — script to plot the root of $z^3 - 1 = 0$ found by Newton-Raphson’s method in the complex plane.

```

1 set encoding utf8
2 set terminal png enhanced font times 48 size 4096,4096
3 set size square
4 set minussign
5 set output 'zxynewton.png'
6 set palette model RGB
7 set palette defined ( 0 "red", 1 "green", 2 "blue" )
8 set notitle
9 set xrange [-2:2]
10 set yrange [-2:2]
11 set format x "%4.1f"
12 set format y "%4.1f"
13 set nokey
14 unset colorbox
15 set arrow from -2,0 to 2,0 front lt 1 lw 6 lc rgb "black"
16 set arrow from 0,-2 to -0,2 front lt 1 lw 6 lc rgb "black"
17 set object 1 circle front at 1,0 size 0.025 lc rgb "black" lw 6 # fillstyle solid
18 set object 2 circle front at -0.5,sqrt(3.0)/2.0 size 0.025 lc rgb "black" lw 6 # fillstyle solid
19 set object 3 circle front at -0.5,-sqrt(3.0)/2.0 size 0.025 lc rgb "black" lw 6 # fillstyle solid
20 set object 4 circle front at 0,0 size 1 lw 6 lc rgb "black"
21 plot 'zxynewton.out' using 1:2:3 with image lc palette
```

The result is shown in figure 1.4. The somewhat surprising result is that it is a *fractal*. We leave it to the reader to enjoy the view.

Listing 1.21: `zxynewton.chpl` — Obtain the root of $z^3 - 1 = 0$ for each pixel of a 4×4 square in the complex plane and color-code it according to the root found: $1+0i = \text{red}, -1/2+i\sqrt{3}/2 = \text{green}, -1/2-i\sqrt{3}/2 = \text{blue}$.

```

1  use IO only openWriter;
2  const colors = {0,1,2};           // an associative domain
3  var Ncol: [colors] int = 0;      // a "dictionary", or associative array
4  const eps = 1.0e-6;
5  const fou = openWriter("zxynewton.out");
6  config const N = 2048;
7  var dxy = 2.0/N;
8  var fz0: complex;
9  for i in -N..N do {
10    for j in -N..N do {
11      var delta = eps;
12      var zinit = i*dxy + (j*dxy)*1i;
13      var z0 = zinit;
14      var z: complex;
15      while (delta >= eps) {
16        fz0 = f(z0);
17        z = z0 - fz0/fl(z0);
18        delta = abs(z - z0);
19        z0 = z;
20      }
21      var color: int;
22      if isClose(z0.im,0.0,absTol=1.0e-5) then {
23        color = 0; // red
24      } else if isClose(z0.im,sqrt(3.0)/2.0,absTol=1.0e-5) then {
25        color = 1; // green
26      }
27      else {
28        color = 2; // blue
29      }
30      Ncol[color] += 1;
31      fou.writef("%+12.6dr.%+12.6dr.%2i\n",
32                  zinit.re,zinit.im,color);
33    }
34  }
35  writeln(Ncol);
36  fou.close();
37  proc f(const in zin: complex): complex {
38    return (zin**3 - 1.0);
39  }
40  proc fl(const in zin:complex): complex {
41    return (3*zin**2);
42 }
```

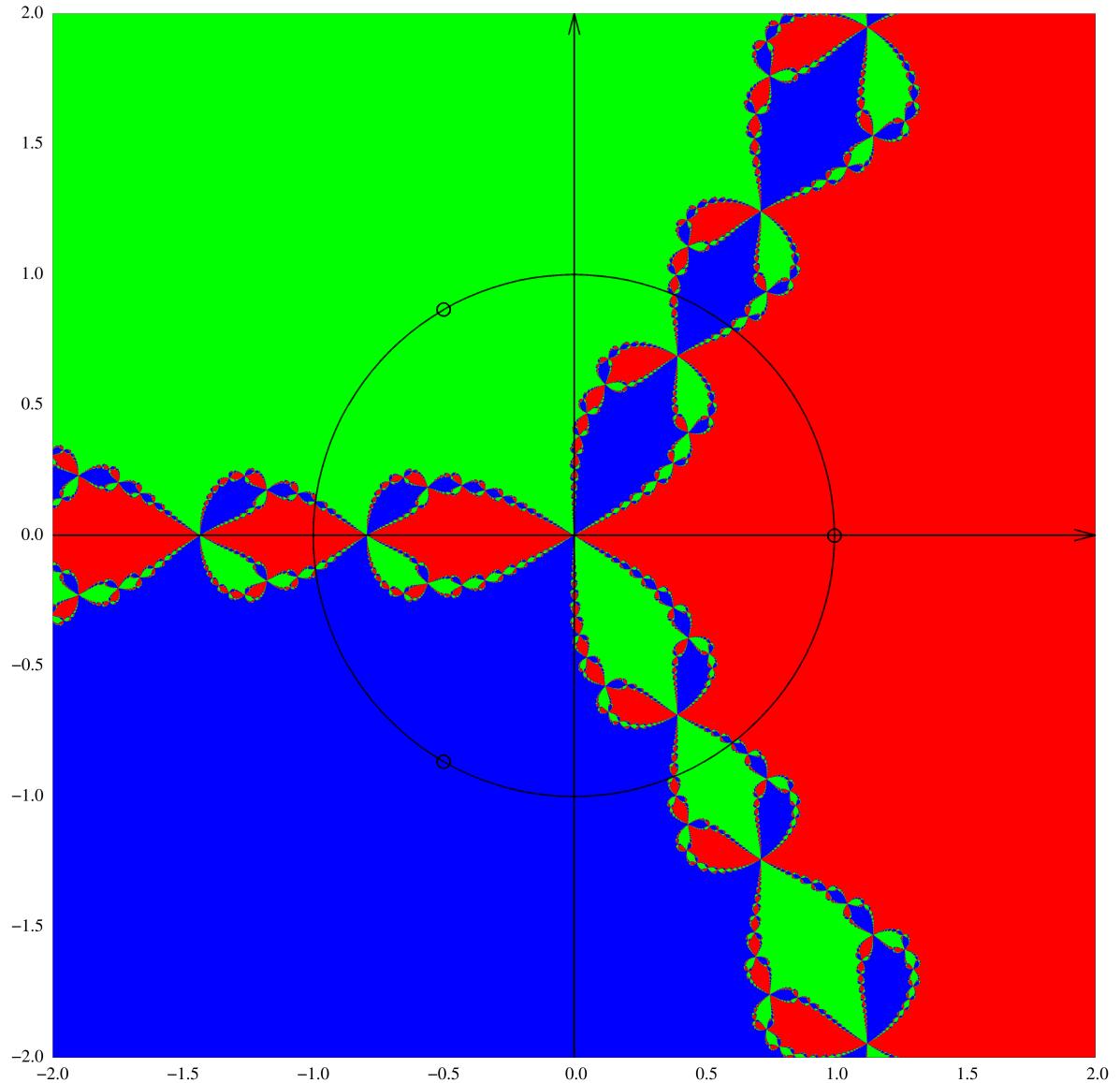


Figure 1.4: Color codes ($1 + 0i$ = red, $-1/2 + i\sqrt{3}/2$ = green, $-1/2 - i\sqrt{3}/2$ = blue) of each of the three roots of $z^3 - 1 = 0$ to which Newton-Raphson’s method converges depending on the initial guess. The three roots are shown as small open circles on the unit circle $|z| = 1$.

1.8 – A first look at parallelization

If you run `zxynewton.chpl` you may notice that it takes a few seconds: the configuration constant N makes for a very large number of calculations of the root z_0 . Speed of this calculation can be significantly improved if you have many cores in your computer (you probably do). The parallel version of it is `zxynewton-p.chpl`, shown in listing 1.22. We invite you to compile and run it, to “feel” the difference.

Listing 1.22: `zxynewton-p.chpl` — Parallel version to obtain the root of $z^3 - 1 = 0$ for each pixel of a 4×4 square in the complex plane and color-code it according to the root found: $1+0i = \text{red}$, $-1/2+i\sqrt{3}/2 = \text{green}$, $-1/2-i\sqrt{3}/2 = \text{blue}$.

```

1 use IO only openWriter;
2 config const N = 2048;
3 const colors = {0,1,2};           // an associative domain
4 var Ncol: [colors] int = 0;      // a "dictionary", or associative array
5 var xycolor: [-N..N,-N..N] int(8); // to save memory
6 var zinit: [-N..N,-N..N] complex;
7 const eps = 1.0e-6;
8 const fou = openWriter("zxynewton-p.out");
9 var dxy = 2.0/N;
10 for i in -N..N do {
11     forall j in -N..N do {
12         var delta = eps;
13         zinit[i,j] = i*dxy + (j*dxy)*1i;
14         var z0 = zinit[i,j];
15         var z, fz0: complex;
16         while (delta >= eps) {
17             fz0 = f(z0);
18             z = z0 - fz0/fl(z0);
19             delta = abs(z - z0);
20             z0 = z;
21         }
22         if isClose(z0.im,0.0,absTol=1.0e-5) then {
23             xycolor[i,j] = 0; // red
24         } else if isClose(z0.im,sqrt(3.0)/2.0,absTol=1.0e-5) then {
25             xycolor[i,j] = 1; // green
26         }
27         else {
28             xycolor[i,j] = 2; // blue
29         }
30         Ncol[xycolor[i,j]] += 1;
31     }
32 }
33 writeln(Ncol);
34 for i in -N..N do {
35     for j in -N..N do {
36         fou.writef("%+12.6dr.%+12.6dr.%2i\n",
37                     zinit[i,j].re,zinit[i,j].im,xycolor[i,j]);
38     }
39 }
40 fou.close();
41 proc f(const in zin: complex): complex {
42     return (zin**3 - 1.0);
43 }
44 proc fl(const in zin:complex): complex {
45     return (3*zin**2);
46 }
```

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 δ e M tais que para $\forall x$, $0 < |x - a| < \delta$,

$$|f(x)| \leq M g(x)$$

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 Δx , ou da “ordem de” Δx . Trocando-se Δx 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 2^a da 1^a,

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

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, Δx^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 1^a 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.

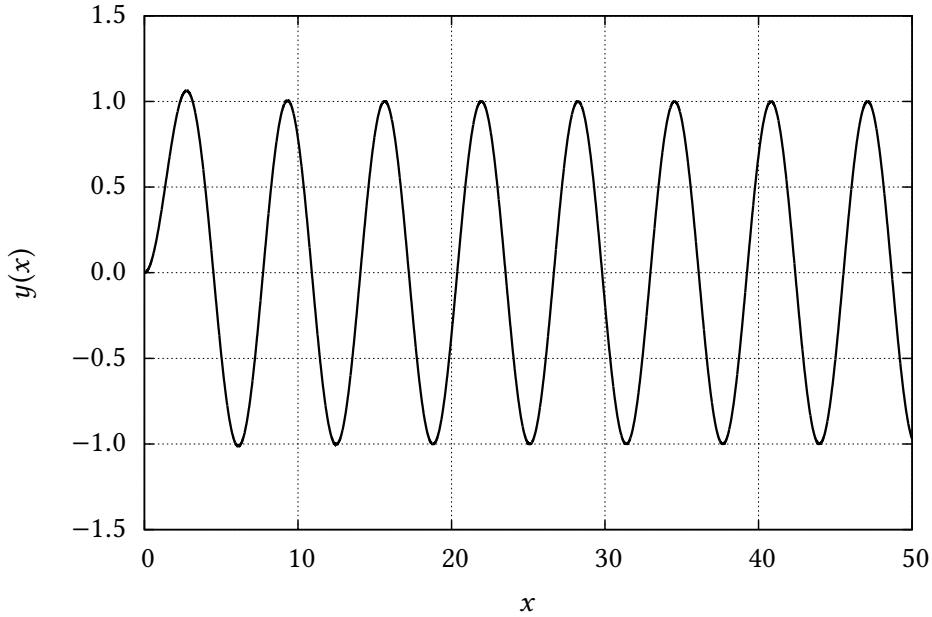


Figure 2.1: Solução da equação (2.7) para $y(0) = 0$.

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 Δx), e uma *lista* de y s correspondentes. Obviamente, como os y s devem aproximar a função, não podemos esperar deles que sejam igualmente espaçados!

Desejamos então:

$$x_0, x_1, \dots, x_N$$

onde

$$x_n = n\Delta x,$$

com os correspondentes

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

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

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

onde eu deixei, propositalmente,

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

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 :

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

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

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

Listing 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();

```

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

Listing 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 Δx bem pequeno, e de $50/0,01 = 5000$ pontos para gerar a solução. Será possível gerar uma solução tão boa com, digamos, 100 pontos?

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!

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 Δx grosso, da ordem de 0,5. Nossa abordagem será propor um esquema *centrado*:

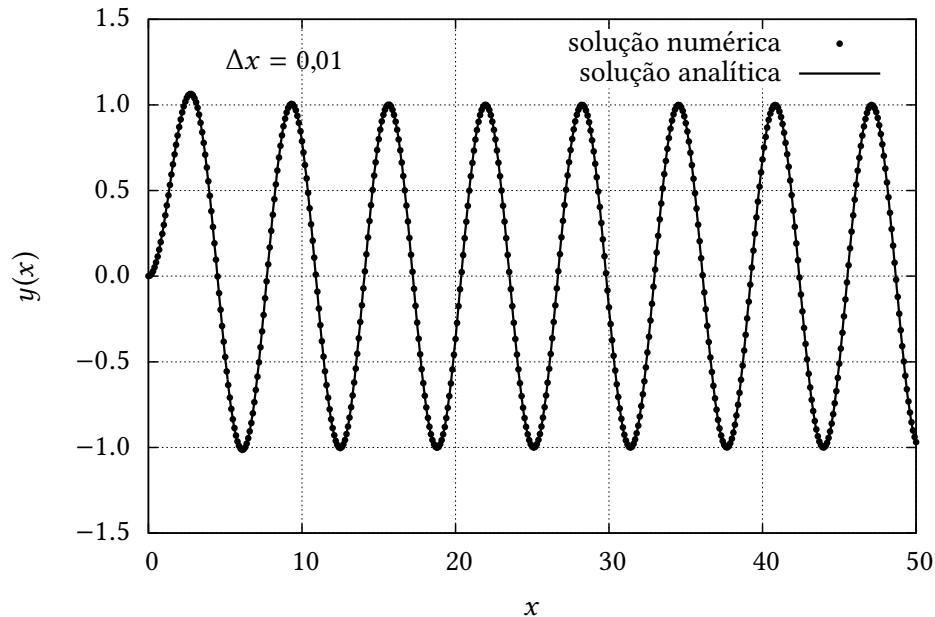
$$\frac{y_{n+1} - y_n}{\Delta x} + \frac{y_n + y_{n+1}}{x_n + x_{n+1}} = \sin\left(\frac{x_n + x_{n+1}}{2}\right).$$

Listing 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);

```

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

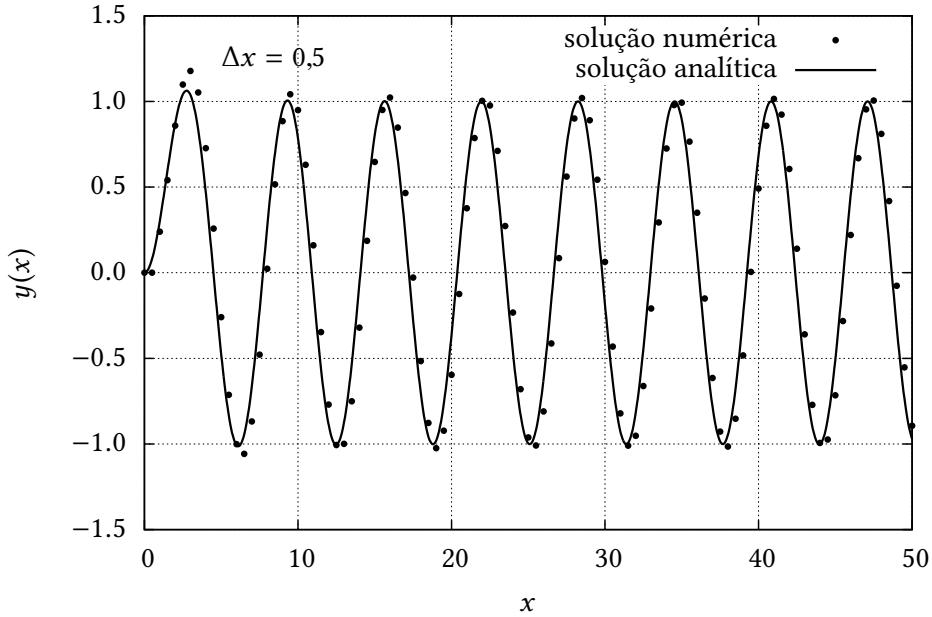


Figure 2.3: Comparação da solução analítica da equação (2.7) com a saída de `sucesso.chpl`, para $\Delta x = 0,5$.

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

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

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

Uma rápida rearrumação produz

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

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 Δx 50 vezes menor!

Listing 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();

```

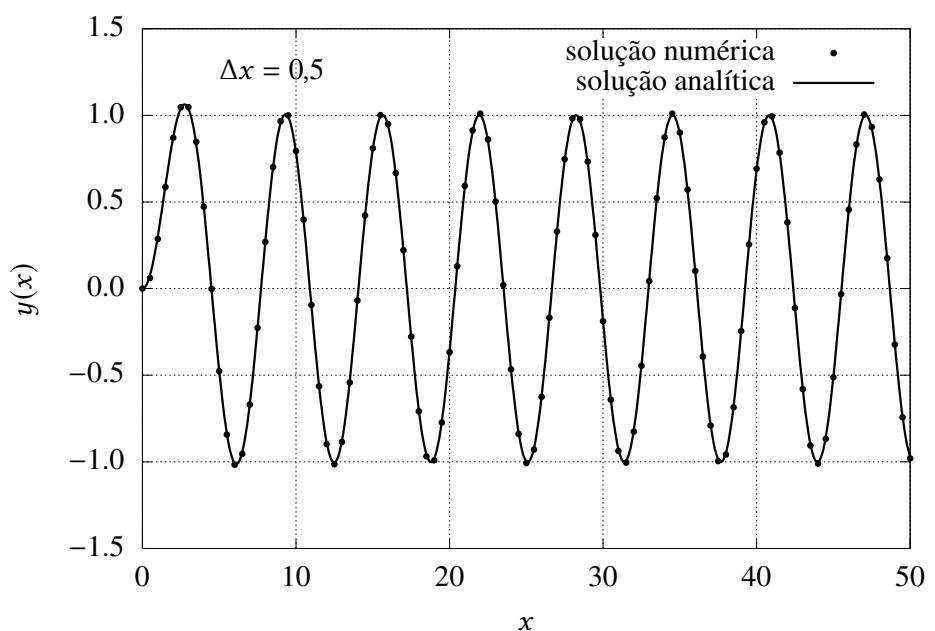


Figure 2.4: Comparação da solução analítica da equação (2.7) com a saída de `succent.chpl`, para $\Delta x = 0,5$.

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 partiríamos de P e chegaríamos, 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 $y(x_n + h/2)$:

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

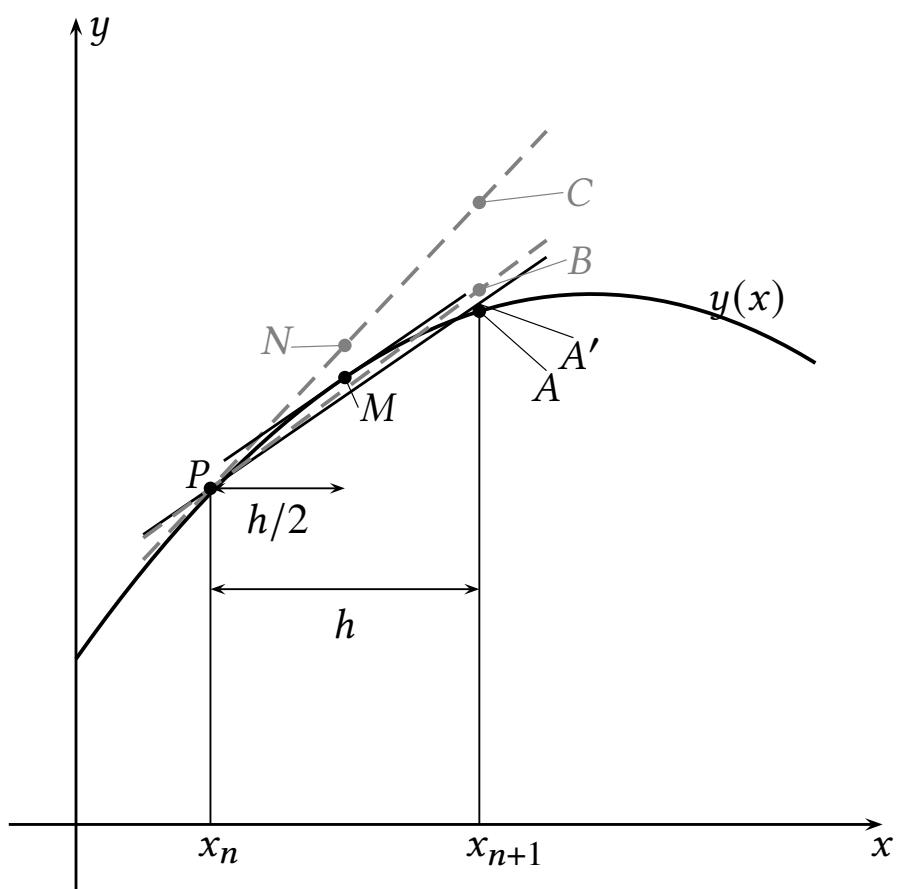


Figure 2.5: Os métodos de Euler de ordens 1 e 2.

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^a 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.

Listing 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 config const h = 0.1;           // 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 }
53 var erro = 0.0;              // calcula o erro relativo médio
54 for i in 1..n do {
55     var yana = sin(x[i])/x[i] - cos(x[i]);
56     erro += abs( (y[i] - yana)/yana );
57 }
58 erro /= n ;
59 writeln("erro_relativo_médio=%10.5dr", erro);
60 writeln();
61 const fou = openWriter("euler2.out", locking=false);
62 for i in 0..n do {          // imprime o arquivo de saída
63     fou.writef("%12.6dr%12.6dr\n",x[i],y[i]);
64 }
65 fou.close();

```

O resultado é muito bom, com um erro absoluto médio $\epsilon = 0,02529$.

Runge-Kutta ordem 4 Mas nós podemos fazer melhor, com o método de Runge-Kutta de 4^a 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.

Listing 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 config const h = 0.1;           // 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*af(x,y);
43     var k2 = h*af(x+h/2,y+k1/2);
44     var k3 = h*af(x+h/2,y+k2/2);
45     var k4 = h*af(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.

2.6 – Um método de passos múltiplos: Adams-Bashforth

Sem entrar nos detalhes do por quê, considere

$$y_{n+2} = y_{n+1} + h \left[\frac{3}{2}f(t_{n+1}, y_{n+1}) - \frac{1}{2}f(t_n, y_n) \right] \quad (2.17)$$

Inicialmente, entretanto, nós só temos y_0 , e precisamos de pelo menos um passo até y_1 . Esse primeiro passo pode ser feito da melhor forma possível. Por exemplo, nós podemos dar 10 passos com um valor

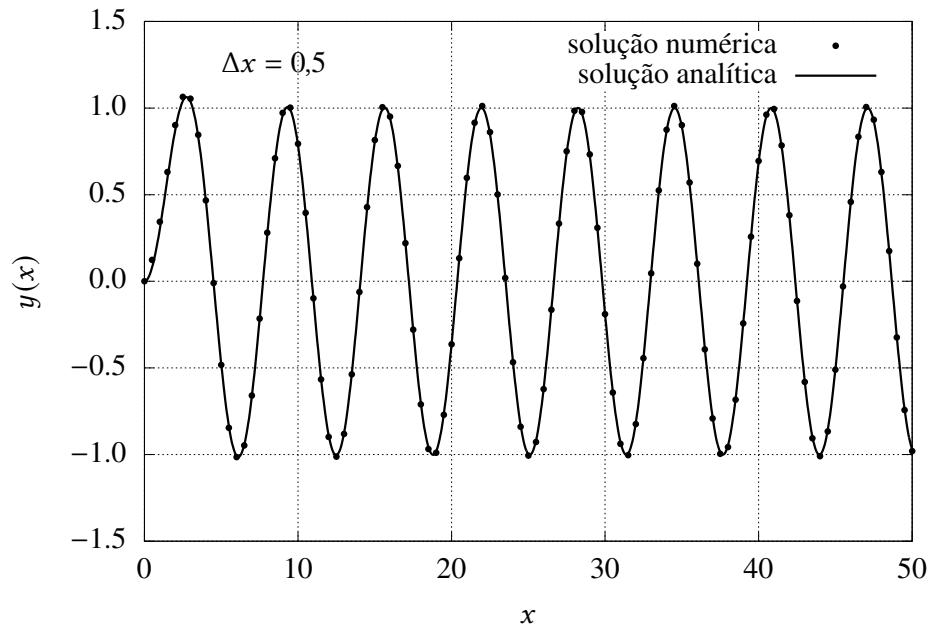


Figure 2.6: Comparaçāo da solução analítica da equaçāo (2.7) com a saída de `euler2.chpl`, para $\Delta x = 0,5$.

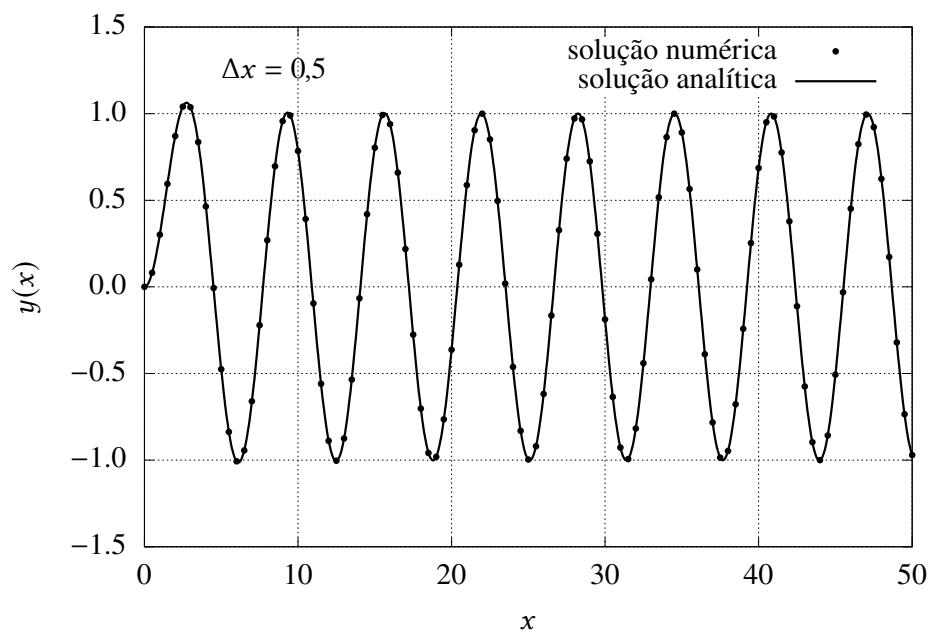


Figure 2.7: Comparaçāo da solução analítica da equaçāo (2.7) com a saída de `rungek4.chpl`, para $\Delta x = 0,5$.

de h bem pequeno, e em seguida começar a aplicar (2.17). Vamos a isso. O programa que resolve a equação diferencial (2.7) (novamente) chama-se `adbash.chpl`.

Listing 2.7: `adbash` — Solução de (2.7) com Adams-Bashforth

```

1 // -----
2 // adbash: resolve a equação diferencial  $dy/dx + y/x = \sin(x)$  usando
3 // Adams-Bashforth ordem 2. O primeiro passo utiliza RK4.
4 // -----
5 use Math only sin, cos;
6 use IO only openWriter;
7 config const h = 0.1;           // passo em x
8 const n = round(50/h):int;     // número de passos
9 var
10    x,                         // variável independente
11    y:                          // variável dependente
12    [0..n] real;
13 // -----
14 // função que define a EDO  $dy/dx = \sin(x) - y/x$ 
15 // -----
16 proc ff(
17    const in x: real,
18    const in y: real
19 ): real {
20    if x == 0.0 then {
21        return 0.0 ;
22    }
23    else {
24        return sin(x) - y/x ;
25    }
26 }
27 // -----
28 // adbash implementa adams-bashforth ordem 2
29 // -----
30 proc adbash(
31    const in xn: real,          //  $x_n$ 
32    const in yn: real,          //  $y_n$ 
33    const in xnm1: real,        //  $x_{n-1}$ 
34    const in ynm1: real,        //  $y_{n-1}$ 
35    const in h: real,           // size of step
36    const ref af: proc(         // function to integrate
37        const in ax: real,
38        const in ay: real
39    ): real
40 ): real {
41    var ynp1 = yn + (h/2.0)*(3*af(xn,yn) - af(xnm1,ynm1));
42    return ynp1;
43 }
44 // -----
45 // 10 passos com  $h$  quadraticamente menor
46 // -----
47 const hh = h*h ;
48 var xq: [0..10] real = [i in 0..10] i*hh ;
49 var yq: [0..10] real;
50 //
51 // primeiro passo com Euler
52 //
53 yq[0] = 0.0;
54 yq[1] = yq[0] + hh*ff(xq[0],yq[0]);
55 //
56 // próximos passinhos com Adams-Bashforth
57 //
58 for i in 2..10 do {
59    yq[i] =adbash(xq[i-1],yq[i-1],xq[i-2],yq[i-2],hh,ff);

```

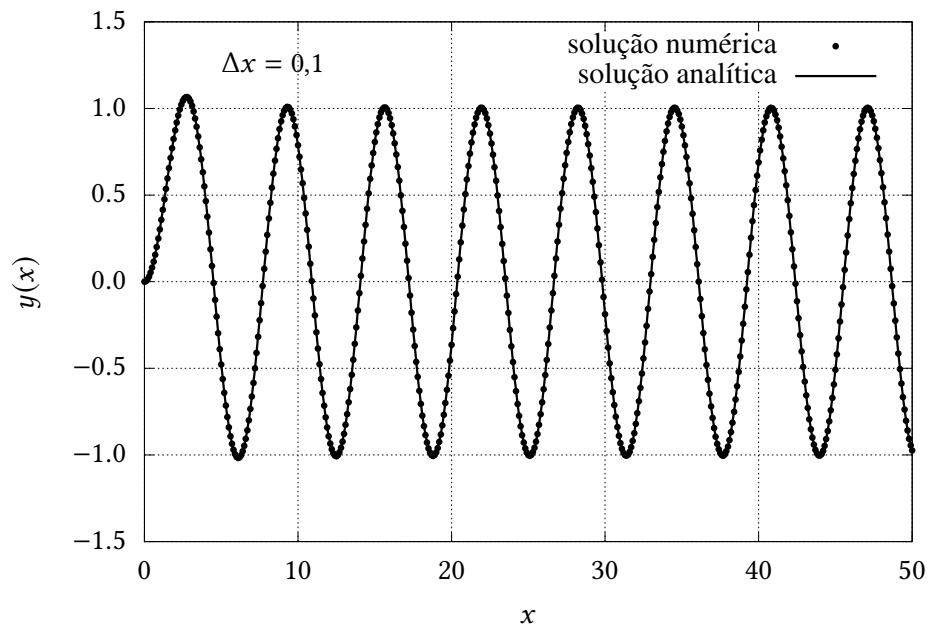


Figure 2.8: Comparação da solução analítica da equação (2.7) com a saída de `adbash.chpl`, para $\Delta x = 0,5$.

```

60 }
61 // -----
62 // agora a solução com passo h
63 // -----
64 x[0] = 0.0;           // x inicial
65 y[0] = 0.0;           // y inicial
66 x[1] = h;
67 y[1] = yq[10];
68 // -----
69 // os demais com Adams-Bashforth
70 // -----
71 for i in 2..n do {      // loop da solução numérica
72     var ynp1 = adbash(x[i-1],y[i-1],x[i-2],y[i-2],h,ff);
73     x[i] = i*h;
74     y[i] = ynp1;
75 }
76 var erro = 0.0;          // calcula o erro relativo médio
77     for i in 1..n do {
78         var yana = sin(x[i])/x[i] - cos(x[i]);
79         erro += abs( (y[i] - yana)/yana );
80 }
81 erro /= n ;
82 writef("erro_relativo_médio=%10.5dr",erro);
83 writeln();
84 const fou = openWriter("adbash.out",locking=false);
85 for i in 0..n do {        // imprime o arquivo de saída
86     fou.writef("%12.6dr.%12.6dr\n",x[i],y[i]);
87 }
88 fou.close();

```

O resultado de rodar `adbash.chpl` com $\Delta x = 0.1$ está mostrado na figura 2.8. Embora o resultado seja visualmente satisfatório, o esquema (2.17) é *pior* do que Euler de ordem 2. Naquele caso, com $\Delta x = 0.5$, o erro médio relativo foi 0.02529; para $\Delta x = 0.1$, `euler2` tem um erro de 0.00233, enquanto que o erro médio relativo de Adams-Bashforth ordem 2 com $\Delta x = 0.1$ é 0.00478 (essencialmente o dobro).

2.7 – Interlúdio sobre arrays

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

Listing 2.8: `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`.

Para contornar essa limitação, nós precisamos implementar um tipo especial chamado vec, no módulo `ada.chpl`, cujas primeiras linhas são mostradas da listagem 2.9.

Listing 2.9: ada — Implementa um vetor (vec) que contorna a limitação de arrays genéricos em argumentos de rotinas.

```

1 // =====
2 // ==> ada: 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 rv: range(int)
15    ) {
16        assert ( rv.size == dom.size );
17        vfirst = rv.first;
18        vlast = rv.last;
19        vdelta = vfirst - dom.first;
20    }
21    proc ref this(in k: int) ref { // access arr[k]
22        return arr[k-vdelta];
23    }
24    //
25    // operator overloading: *all* of the subsequent operators return vec. if you
26    // want to assign an operation between an array b and a vec c to an array a,
27    // just say: a = b + c.arr
28    //
29    // the domain of the vec returned is always the same as the domain of the
30    // lhs, except when the lhs is a scalar
31    //
32    operator +=(ref rhs: vec) { // this + vec
33        this.arr += rhs.arr;
34    }
35    operator -(rhs: vec) { // this - vec
36        this.arr -= rhs.arr ;
37    }
38
39    operator +(lhs: real, rhs: vec): vec { // real + vec
40        var r = new vec(rhs.dom);
41        r.arr = lhs + rhs.arr;
42        return r;
43    }
44    operator +(lhs: vec, rhs: real): vec { // vec + real
45        var r = new vec(lhs.dom);
46        r.arr = lhs.arr + rhs;
47        return r;
48    }
49
50    operator +(lhs: vec, rhs: vec): vec { // vec + vec
51        const n = lhs.size;
52        assert (n == rhs.size);
53        var r = new vec(lhs.dom);
54        r.arr = lhs.arr + rhs.arr ;
55        return r;
56    }
57    operator -(lhs: real, rhs: vec): vec { // real - vec

```

```
58     var r = new vec(rhs.dom);
59     r.arr = lhs - rhs.arr ;
60     return r;
61 }
62 operator -(lhs: vec, rhs: real): vec { // vec - real
63     var r = new vec(lhs.dom);
64     r.arr = lhs.arr - rhs;
65     return r;
66 }
67 operator -(lhs: vec, rhs: vec): vec { // vec - vec
68     const n = lhs.size;
69     assert (n == rhs.size);
70     var r = new vec(lhs.dom);
71     r.arr = lhs.arr - rhs.arr;
72     return r;
73 }
74 operator *(lhs: real, rhs: vec): vec { // real * vec
75     var r = new vec(rhs.dom);
76     r.arr = lhs*rhs.arr;
77     return r;
78 }
79 operator *(lhs: vec, rhs: real): vec { // vec * real
80     var r = new vec(lhs.dom);
81     r.arr = lhs.arr*rhs;
82     return r;
83 }
84 operator *(lhs: vec, rhs: vec): vec { // vec * vec
85     assert (lhs.size == rhs.size);
86     var r = new vec(lhs.dom);
87     r.arr = lhs.arr*rhs.arr;
88     return r;
89 }
90 operator /(lhs: real, rhs: vec): vec { // real / vec
91     var r = new vec(rhs.dom);
92     r.arr = lhs/rhs.arr;
93     return r;
94 }
95 operator /(lhs: vec, rhs: real): vec { // vec / real
96     var r = new vec(lhs.dom);
97     r.arr = lhs.arr/rhs;
98     return r;
99 }
100 operator /(lhs: vec, rhs: vec): vec { // vec / vec
101     const n = lhs.size;
102     assert (n == rhs.size);
103     var r = new vec(lhs.dom);
104     r.arr = lhs.arr/rhs.arr;
105     return r;
106 }
107 operator **(lhs: vec, rhs: int): vec { // vec**int
108     var r = new vec(lhs.dom);
109     r.arr = lhs.arr**rhs;
110     return r;
111 }
112 operator **(lhs: real, rhs: vec): vec { // real**vec
113     var r = new vec(rhs.dom);
114     r.arr = lhs**rhs.arr;
115     return r;
116 }
117 operator **(lhs: vec, rhs: real): vec { // vec**real
118     var r = new vec(lhs.dom);
119     r.arr = lhs.arr**rhs;
120     return r;
```

```

121     }
122     operator **(lhs: [] real, rhs: vec): vec
123         where lhs.rank == 1 {                      // [] real**vec
124             const n = lhs.size;
125             assert (n == rhs.size);
126             var r = new vec(rhs.dom);
127             r.arr = lhs**rhs.arr;
128             return r;
129     }
130     operator **(lhs: vec, rhs: vec): vec {      // exponentiation
131         const n = lhs.size;
132         assert (n == rhs.size);
133         var r = new vec(lhs.dom);
134         r.arr = lhs.arr**rhs.arr;
135         return r;
136     }
137     proc compare(i,j) {
138         return arr[i] - arr[j] ;
139     }
140 }
141
142 record mat {
143     var dom: domain(2);                         // the 2d domain
144     var arr: [dom] real;                        // the array
145     var v0first = dom.dim(0).first;            // the first index after reind
146     var v1first = dom.dim(1).first;
147     var v0delta = 0;                            // the array shift
148     var v1delta = 0;
149     proc size: int {                          // the size of a mat
150         return dom.size;
151     }
152     proc shape: 2*int {
153         return dom.shape;
154     }
155     proc ref reindex(
156         const in dv0: range(int),
157         const in dv1: range(int)
158     ) {
159         assert ( dom.shape == (dv0.size,dv1.size) );
160         v0first = dv0.first;
161         v0delta = v0first - dom.dim(0).first;
162         v1first = dv1.first;
163         v1delta = v1first - dom.dim(1).first;
164     }
165     proc ref this(in k: int, in l: int) ref {    // access arr[k]
166         return arr[k-v0delta,l-v1delta];
167     }
168 // -----
169 // operator overloading: *all* of the subsequent operators return
170 // mat. if you want to assign an operation between an array b and a
171 // mat c to an array a, just say: a = b + c.arr
172 //
173 // the domain of the mat returned is always the same as the domain of
174 // the lhs, except when the lhs is a scalar
175 //
176     operator +(lhs: real, rhs: mat): mat {    // real + mat
177         var r = new mat(rhs.dom);
178         r.arr = lhs + rhs.arr;
179         return r;
180     }
181     operator +(lhs: mat, rhs: real): mat {    // mat + real
182         var r = new mat(lhs.dom);
183         r.arr = lhs.arr + rhs;

```

```

184     return r;
185 }
186 operator +(lhs: mat, rhs: mat): mat {      // mat + mat
187     assert (lhs.shape== rhs.shape);
188     var r = new mat(lhs.dom);
189     r.arr = lhs.arr + rhs.arr ;
190     return r;
191 }
192 operator -(lhs: real, rhs: mat): mat {      // real - mat
193     var r = new mat(rhs.dom);
194     r.arr = lhs - rhs.arr ;
195     return r;
196 }
197 operator -(lhs: mat, rhs: real): mat {      // mat - real
198     var r = new mat(lhs.dom);
199     r.arr = lhs.arr - rhs;
200     return r;
201 }
202 operator -(lhs: mat, rhs: mat): mat {      // mat - mat
203     assert (lhs.shape == rhs.shape);
204     var r = new mat(lhs.dom);
205     r.arr = lhs.arr - rhs.arr;
206     return r;
207 }
208 }
209
210 proc tovec(x: [] real): vec where x.rank == 1 {
211     var v = new vec(x.domain);
212     v.arr = x;
213     return v;
214 }
215
216 proc tomat(x: [] real): mat where x.rank == 2 {
217     var m = new mat(x.domain);
218     m.arr = x;
219     return m;
220 }
```

2.8 – O método de Runge-Kutta multidimensional

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

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

Note que y e f são **vetores**, enquanto que x permanece sendo um escalar. Neste livro, **vetores** são escritos em **negrito**: 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

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

$$\begin{aligned}k_4 &= hf(x_n + h, \mathbf{y}_n + \mathbf{k}_3), \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{6}\mathbf{k}_1 + \frac{1}{3}\mathbf{k}_2 + \frac{1}{3}\mathbf{k}_3 + \frac{1}{6}\mathbf{k}_4.\end{aligned}$$

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

Listing 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 ada;                                // 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     const ref 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`:

Listing 2.11: rktestr — Resolve um sistema de EDOs com o método de Runge-Kutta multidimensional.

```

1 // =====
2 // ==> rktestr: solve
3 //
4 // dy1/dx = y2,
5 // dy2/dx = y1,
6 //
7 // using the Runge-Kutta method; uses runkutr, which calls all vecs by reference
8 // =====
9 use ada;                      // we need vecs!
10 use runkutr;                   // runge-kutta comes from here
11 const h = 0.1;                  // step size
12 const nt = round(10/h): int;    // number of steps
13 var x: [0..nt] real = 0.0;      // independent variable
14 var y: [0..nt] vec;            // dependent variable
15 proc ff()                      // specify integration function (b)
16   const in x: real,
17   ref y: vec,                  // intent is ref: vec fields cannot be const
18   ref dydx: vec                // intent is ref
19 {
20   dydx[1] = y[2];              // assume both domains are == {1..2}!
21   dydx[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 writef("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 2.9. Note que $\cosh(x) \approx \sinh(x)$ para $x \gtrsim 2.5$.

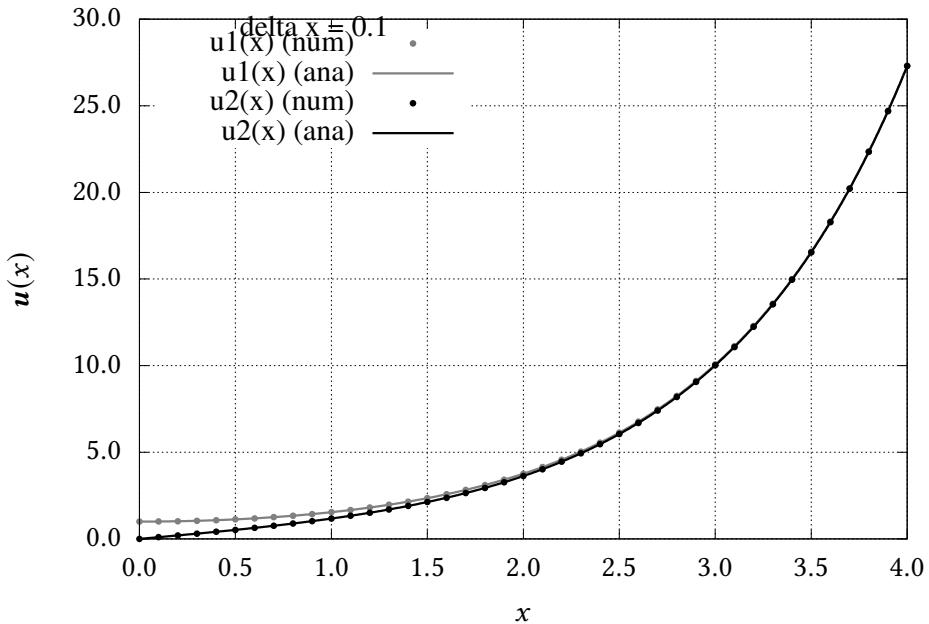


Figure 2.9: Solução numérica pelo Método de Runge-Kutta de um sistema de 2 equações diferenciais ordinárias

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:

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

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 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 $Q = Q(A)$:

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

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

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

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

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

$$\frac{d\mathbf{Q}}{dt} = F(t, \mathbf{Q}),$$

$$F_i(Q, t) = -K_F Q_i^{2/5} \frac{Q_i - Q_{i-1}}{\Delta x}, \quad i = 1, \dots, N_x.$$

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

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

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

$$\begin{aligned} \frac{dQ}{dt} &= F(t, Q), \\ \mathbf{k}_1 &= \Delta t F(t_k, Q^k), \\ \mathbf{k}_2 &= \Delta t F(t_k + \frac{\Delta t}{2}, Q^k + \frac{1}{2}\mathbf{k}_1), \\ \mathbf{k}_3 &= \Delta t F(t_k + \frac{\Delta t}{2}, Q^k + \frac{1}{2}\mathbf{k}_2), \\ \mathbf{k}_4 &= \Delta t F(t_k + \Delta t, Q^k + \mathbf{k}_3), \\ Q^{k+1} &= Q^k + \frac{1}{6}\mathbf{k}_1 + \frac{1}{3}\mathbf{k}_2 + \frac{1}{3}\mathbf{k}_3 + \frac{1}{6}\mathbf{k}_4. \end{aligned}$$

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`:

Listing 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 ada;
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," ,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**0.4;
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)

```

```

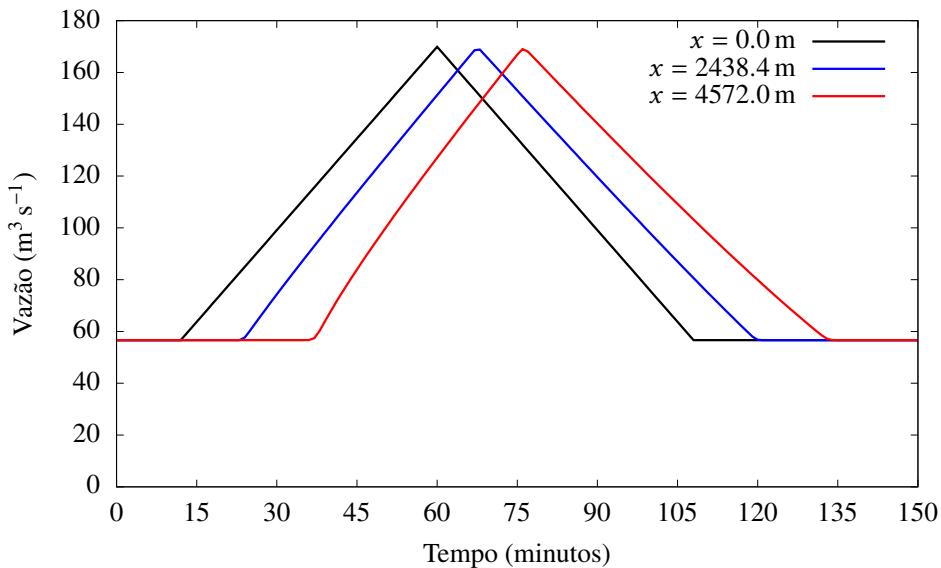
71 // -----
72 // guarda valores selecionados da condição inicial em Qs[0]
73 //
74 for isx in 0..IL do {
75   Qs[0,isx] = QQ[0][isx*acadai];
76 }
77 //
78 // prepara o início da simulação
79 //
80 var iold = 0;           // posição das alturas antigas
81 var inew = 1;           // posição das novas 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 dQdt: vec
118 ) {
119   dQdt[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 dQdt.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 [Chow et al. \(1988\)](#)).

2.9 – Problemas de valor de contorno em 1D

Considere o problema de valor de contorno ([Versteeg e Malalasekera, 2007](#), Eq. 4.13)

$$\frac{d^2T}{dx^2} = 0, \quad (2.18)$$

$$T(0) = T_0, \quad (2.19)$$

$$T(L) = T_L, \quad (2.20)$$

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

$$T_a(x) = 100 + 800x, \quad 0 \leq x \leq 0.5. \quad (2.21)$$

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 (2.18):

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

ou, simplesmente,

$$-T_{i-1} + 2T_i - T_{i+1} = 0. \quad (2.22)$$

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*

$$2T_1 - T_2 = T_0, \quad (2.23)$$

$$-T_{N-1} + 2T_{N-1} = T_N. \quad (2.24)$$

Em conjunto, (2.22)–(2.24) produzem o sistema de equações lineares

$$\left[\begin{array}{cccccc} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ \vdots & & & & \vdots & \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{array} \right] \left[\begin{array}{c} T_1 \\ T_2 \\ \vdots \\ T_{N-2} \\ T_{N-1} \end{array} \right] = \left[\begin{array}{c} T_0 \\ 0 \\ \vdots \\ 0 \\ T_L \end{array} \right]. \quad (2.25)$$

Existem várias coisas atraentes para um programador em (2.25). 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: Press et al., 1992, seção 2.4, subrotina `tridag`). Em segundo lugar, a matriz do sistema é *constante*: ela só precisa ser montada uma vez no programa, o que torna a solução numérica potencialmente muito rápida.

Nós vamos começar, então, construindo um pequeno módulo, convenientemente denominado `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 (2.18)–(2.20), na listagem 2.14.

Para $N = 10$, a solução é mostrada na figura 2.10

Nós definimos o erro médio absoluto relativo da solução por

$$\epsilon_m \equiv \frac{1}{N-1} \sum_{i=1}^{N-1} \left| \frac{T_i - T_a(i\Delta x)}{T_L - T_0} \right|. \quad (2.26)$$

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 2.11. 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.

Listing 2.13: `tridiag.chpl` — Exporta uma rotina que resolve um sistema tridiagonal.

```

1 // -----
2 // solution of a system with a tridiagonal matrix
3 // -----
4 proc tridiag(
5     const ref aa: [] real,      // lower diagonal
6     const ref ab: [] real,      // main diagonal
7     const ref ac: [] real,      // upper diagonal
8     const ref ay: [] real,      // forcing vector
9     ref ax: [] real            // solution
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 }
```

Listing 2.14: edo-l1.chpl — Solução do problema de valor de contorno (2.18)–(2.20).

```

1 // -----
2 // edoo-l1: solves d2T/dx2 = 0
3 // -----
4 config const N = 11;                                // 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 const 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 }
```

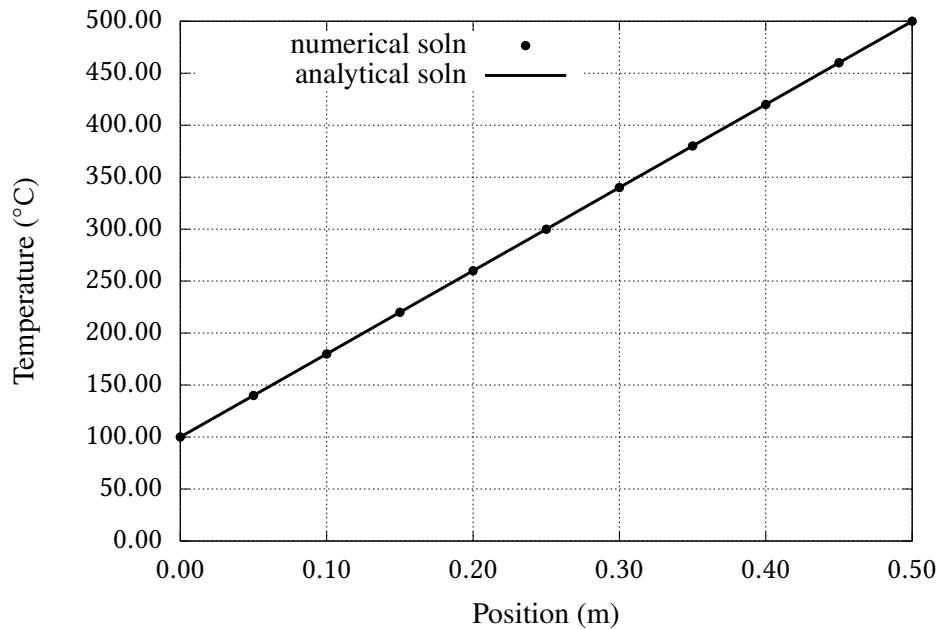


Figure 2.10: Solução numérica de (2.18)–(2.20).

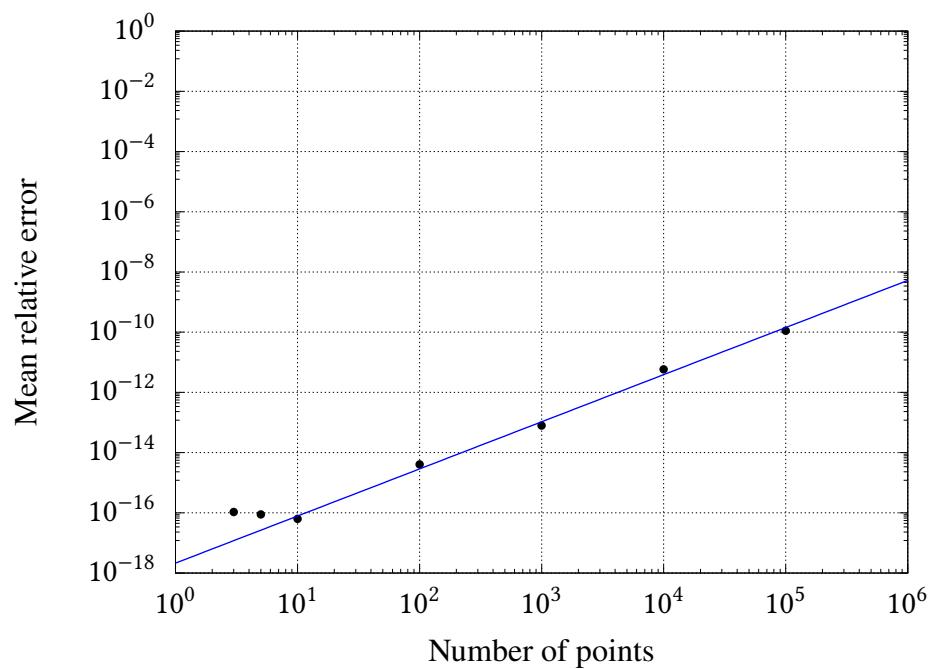


Figure 2.11: Erro da solução numérica de (2.18)–(2.20) em função do número de pontos N da grade.

Considere agora o problema de valor de contorno

$$\frac{d^2y}{dx^2} - k^2y = 0, \quad (2.27)$$

$$y(0) = y_0, \quad (2.28)$$

$$\frac{dy(L)}{dx} = 0. \quad (2.29)$$

A solução é

$$y(x) = A \cosh(kx) + B \sinh(kx), \\ y'(x) = k [A \sinh(kx) + B \cosh(kx)].$$

Então,

$$A = y_0, \\ 0 = k [y_0 \sinh(kL) + B \cosh(kL)], \\ B \cosh(kL) = -y_0 \sinh(kL), \\ B = -y_0 \operatorname{tgh}(kL).$$

Consequentemente, a solução analítica é

$$y_a(x) = y_0 [\cosh(kx) - \operatorname{tgh}(kL) \sinh(kx)]. \quad (2.30)$$

A discretização de (2.27) é a seguinte

$$\frac{y_{i-1} - 2y_i + y_{i+1}}{\Delta x^2} - k^2 y_i = 0, \\ y_{i-1} - 2y_i - (k\Delta x)^2 y_i + y_{i+1} = 0, \\ -y_{i-1} + (2 + \kappa^2) y_i - y_{i+1} = 0, \quad (2.31)$$

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

$$(2 + \kappa^2)y_1 - y_2 = y_0. \quad (2.32)$$

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

$$\frac{dy(L)}{dx} = \frac{y_N - y_{N-1}}{\Delta x} + \mathcal{O}(\Delta x) = 0, \quad (2.33)$$

$$y_N = y_{N-1},$$

$$-y_{N-2}(2 + \kappa^2)y_{N-1} - y_N = 0, \\ -y_{N-2}(2 + \kappa^2)y_{N-1} - y_{N-1} = 0, \\ -y_{N-2} + (1 + \kappa^2)y_{N-1} = 0. \quad (2.34)$$

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 2.12.

Agora nós olhamos para a *taxa de convergência* do método numérico para grades progressivamente refinadas. Isso é mostrado na figura 2.13. Observe que, ao contrário da figura 2.11, 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 2.13 é -1.0744 . Isso significa que o erro decresce *linearmente* com a diminuição de Δx , apesar do esquema centrado para a derivada segunda

Listing 2.15: edo-l2.chpl — Solução do problema de valor de contorno (2.27)–(2.29).

```

1 // -----
2 // edo-l2: solves  $d^2y/dx^2 - k^2 y = 0$ 
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 const x: [0..N] real = [i in 0..N] i*dx; // this is an expression for all
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 = 0.0;               // 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] = y0;                                     // 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 }
```

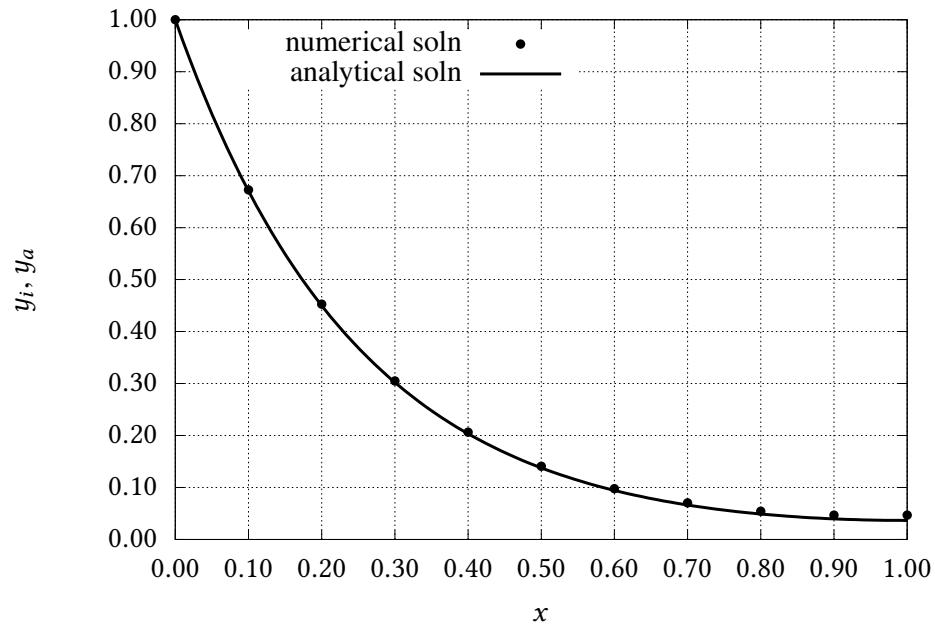


Figure 2.12: Solução numérica de (2.27)–(2.29) com uma discretização de ordem 1 para a condição de contorno direita.

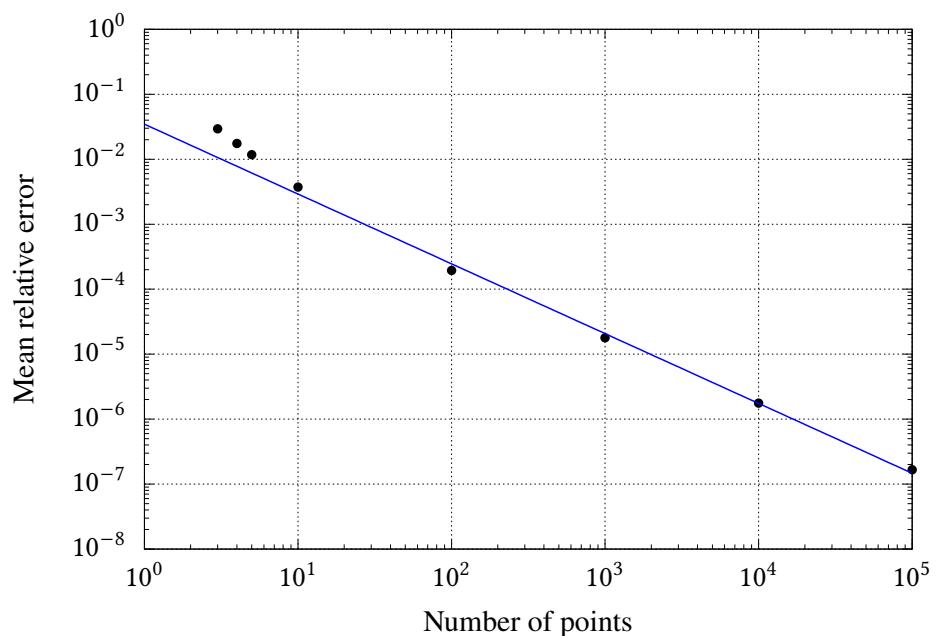


Figure 2.13: Erro da solução numérica de (2.27)–(2.29) em função do número de pontos N da grade. O coeficiente angular da reta é ~ -1 .

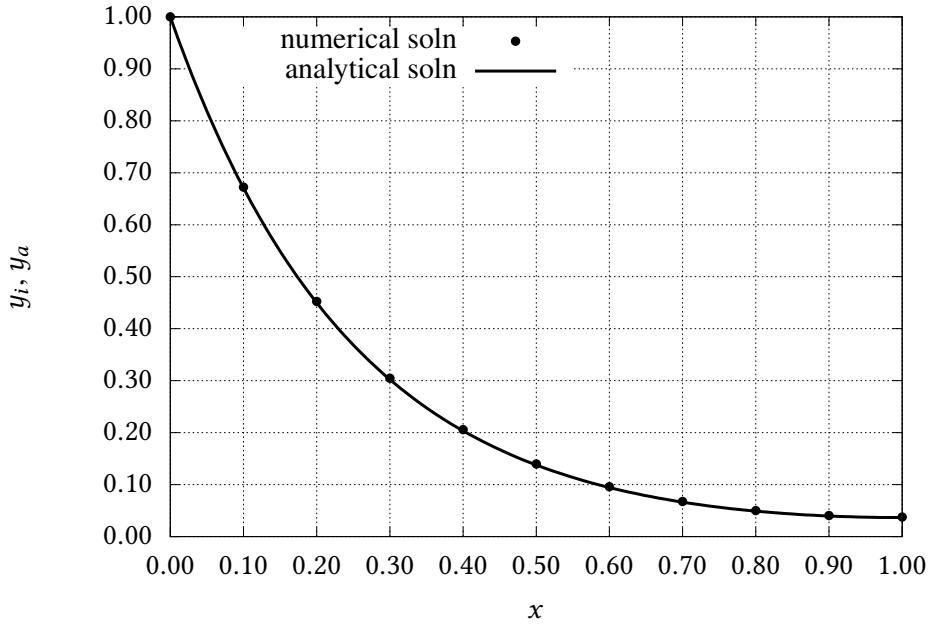


Figure 2.14: Solução numérica de (2.27)–(2.29) com uma discretização de ordem 2 para a condição de contorno direita.

ser de $\mathcal{O}(\Delta x^2)$! A culpada, neste caso, é a aproximação para a derivada da condição de contorno (2.33), 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 (2.29), temos agora a condição de contorno

$$\frac{dy(L)}{dx} = \frac{y_{N+1} - y_{N-1}}{2\Delta x} + \mathcal{O}(\Delta x^2) = 0, \quad (2.35)$$

$$\begin{aligned} y_{N+1} &= y_{N-1}, \\ -y_{N-1} + (2 + \kappa^2)y_N - y_{N+1} &= 0, \\ -2y_{N-1} + (2 + \kappa^2)y_N &= 0, \end{aligned} \quad (2.36)$$

Em resumo, agora temos N (e não mais $N - 1$) incógnitas, e a última linha da matriz do sistema é (2.36). 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 2.14.

Agora nós olhamos para a *taxa de convergência* do método numérico para grades progressivamente refinadas. Isso é mostrado na figura 2.15. 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.

Listing 2.16: `edo-l2b.chpl` — Solução do problema de valor de contorno (2.27)–(2.29) com um esquema de ordem 2 para a condição de contorno direita.

```

1 // -----
2 // edo-l2b: solves d2y/dx2 - 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 const 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;                                // the solution to the problem
23 tridiag(a,b,c,d,y[1..N]);                  // BC enforced after tridiag
24 y[0] = y0;                                 // BC enforced after tridiag
25 y[N+1] = y[N-1];
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 inline proc anasol(
43   const in x: real
44   ): real {
45   return y0*(cosh(k*x) - tanh(k*L)*sinh(k*x));
46 }
```

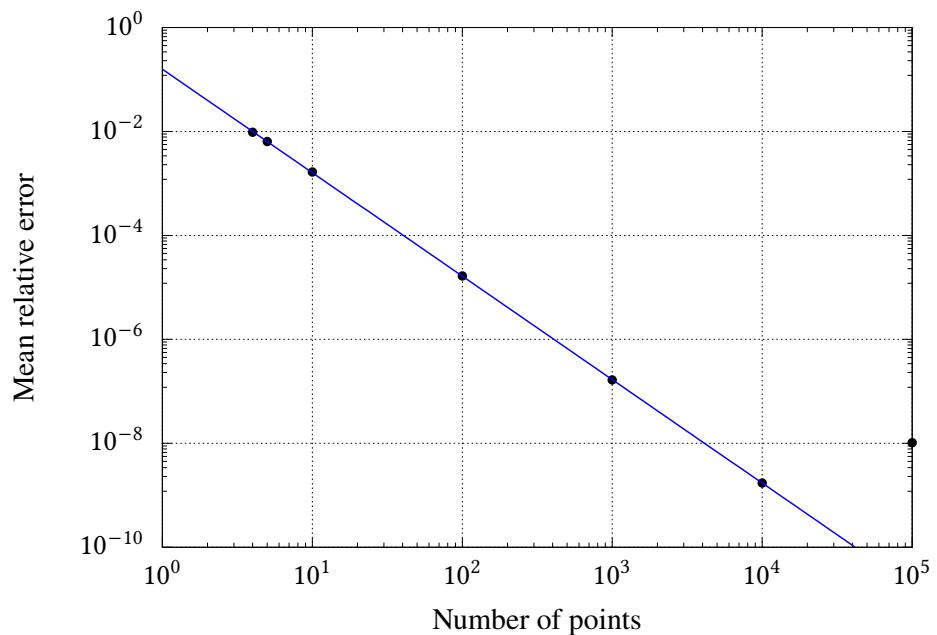


Figure 2.15: Erro da solução numérica de (2.27)–(2.29) em função do número de pontos N da grade. O coeficiente angular da reta é ~ -2

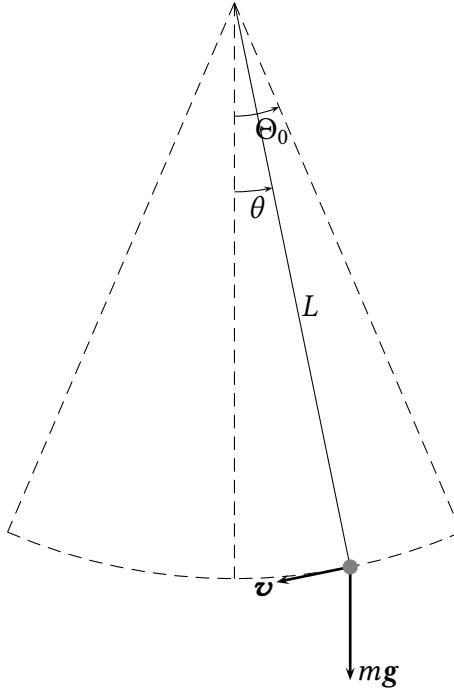


Figure 2.16: Um pêndulo (possivelmente) não-linear.

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 2.16 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 2^a lei de Newton nos dá

$$\begin{aligned} -mL \frac{d^2\theta}{dt^2} &= mg \sin(\theta), \\ \frac{d^2\theta}{dt^2} + \frac{g}{L} \sin(\theta) &= 0. \end{aligned} \tag{2.37}$$

A dimensão da equação (2.37) 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 (2.37), obtém-se

$$\frac{d^2\theta}{d\tau^2} + \operatorname{sen}(\theta) = 0, \quad \theta(0) = \Theta_0, \quad \theta'(0) = 0 \quad (2.38)$$

(note que nós agora incluímos as condições inciais).

Essa equação diferencial não pode ser resolvida por métodos analíticos em termos apenas em funções transcendentais elementares (funções baseadas nas funções trigonométricas e na função exponencial (incluindo as inversas)). Se a posição angular inicial Θ_0 for “pequena”, podemos aproximar o seno por uma série de Taylor apenas até o primeiro termo, $\operatorname{sen} \theta \approx \theta$, e transformar a equação diferencial em

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

Talvez a característica mais interessante dessa solução seja que o período de oscilação *não depende da amplitude* Θ_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 τ 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 iniciais Θ_0 , e plotar o período de cada uma dessas soluções contra Θ_0 .

Para resolver o problema não-linear, escrevemos, a partir de (2.38):

$$\frac{d\theta}{d\tau} = \omega, \quad \theta(0) = \Theta_0, \quad (2.39)$$

$$\frac{d\omega}{d\tau} = -\sin(\theta), \quad \omega(0) = 0, \quad (2.40)$$

e resolvemos o sistema (2.39)–(2.40) com o método de Runge-Kutta de 4^a ordem para um grande número de condições iniciais Θ_0 , a saber: $\Theta_0 = 0.05, 0.1, \dots, 3$. Para cada um desses valores, nós determinamos o período da solução. Em termos da variável adimensional independente τ , 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 (2.39)–(2.40). Uma idéia simples e que funciona é a seguinte:

1. Adote um passo pequeno para a solução numérica de (2.39)–(2.40). Por exemplo, $\Delta\tau = 0.001$, e simule até $\tau = 50$ para obter um certo número de períodos.
2. Para cada Θ_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 θ 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 τ_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 δ_k s. O período será

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

6. Guarde esse $f(\Theta_0)$, e prossiga para o próximo Θ_0 em 2.

Você deve plotar os seus resultados para conferir. A função $f(\Theta_0)$ é mostrada na figura 2.17

Ressonância e um pêndulo não-linear

A figura 2.16 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^a lei de Newton nos dá

$$\begin{aligned} -mL \frac{d^2\theta}{dt^2} &= mg \sin \theta + F(t), \\ \frac{d^2\theta}{dt^2} + \frac{g}{L} \sin \theta &= -\frac{F(t)}{mL}, \\ \frac{d^2\theta}{dt^2} + \frac{g}{L} \sin \theta &= \frac{mg}{mL} \phi(t). \end{aligned} \quad (2.41)$$

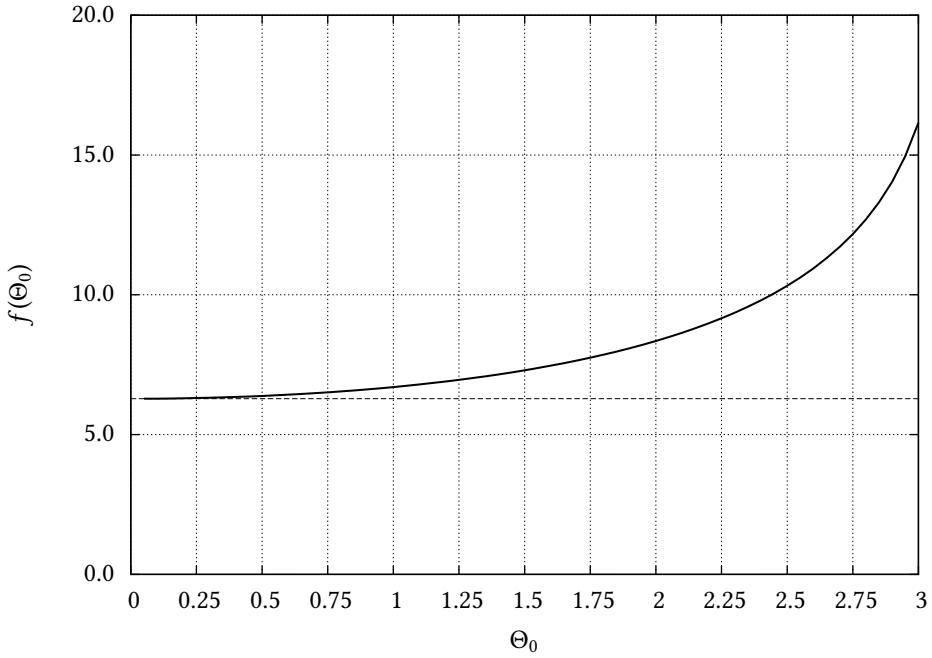


Figure 2.17: O período de um pêndulo não linear em função da amplitude inicial Θ_0 . A linha tracejada é o valor teórico $f(0) = 2\pi$ para oscilações de pequena amplitude.

A dimensão da equação (2.41) é

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

mas ela pode ser adimensionalizada via

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

Substituindo agora na equação (2.41), obtém-se

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

(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 Θ_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

$$\frac{d^2\theta}{d\tau^2} + \theta = \phi(\tau), \quad \theta(0) = \Theta_0, \quad \theta'(0) = 0. \quad (2.43)$$

Se

$$\phi(\tau) = \sin(\tau), \quad (2.44)$$

aparecerá ressonância em (2.43). Nossa questão é: aparecerá também ressonância em (2.42)?

Para resolver o problema não-linear, escrevemos, a partir de (2.42):

$$\frac{d\theta_n}{d\tau} = \omega_n, \quad \theta_n(0) = \Theta_0, \quad (2.45)$$

$$\frac{d\omega_n}{d\tau} = -\sin \theta_n + \sin(\tau), \quad \omega_n(0) = 0. \quad (2.46)$$

O problema linear tem solução numérica muito parecida:

$$\frac{d\theta_l}{d\tau} = \omega_l, \quad \theta_l(0) = \Theta_0, \quad (2.47)$$

$$\frac{d\omega_l}{d\tau} = -\theta_l + \sin(\tau), \quad \omega_l(0) = 0, \quad (2.48)$$

Em (2.45)–(2.48) 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 (2.45)–(2.46) e (2.47)–(2.48) utilizando o método de Runge-Kutta de 4ª ordem com $\Theta_0 = 1,5$. Você deve utilizar um passo $\Delta\tau = 0,01$ em τ , 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 2π (uma volta completa), e portanto é difícil interpretar fisicamente a solução numérica após valores de τ da ordem de 2π . 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

$$f_R(x) = \frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}, \quad x \geq 0. \quad (2.49)$$

A função distribuição acumulada é

$$F_R(x) = \int_0^x f_R(t) dt = 1 - e^{-\frac{x^2}{2\sigma^2}}. \quad (2.50)$$

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

$$\int_0^\infty f_R(x) dx = F_R(\infty) = 1. \quad (2.51)$$

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 (2.49), ou de sua série de Taylor.

Parte I

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

$$f_S(x) = c_1x + c_2x^2 + c_3x^3 + c_4x^4 + \dots \quad (2.52)$$

(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:

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

η 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 (2.52). Portanto, quando $x > \sqrt{10}$, calcule y , η , λ , z e N , calcule $f_T(z)$, eleve a N , e multiplique por λz .

Parte II

Verifique numericamente a validade de (2.51), usando um esquema de integração numérica de sua escolha. O problema é lidar com o limite superior da integral, que é ∞ . Como você pode fazer para garantir que sua integral numérica é uma boa aproximação de (2.51)? Afinal, é impossível colocar ∞ em um programa de computador...

Dica: o capítulo sobre integração numérica de Press et al. (1992) tem várias sugestões para lidar com esse problema. Você pode usar as sugestões de Press et al. (1992), 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 (2.51) 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 (Dias, 1995).

Importância dos cálculos de curvas de remanso Um problema clássico em hidráulica de canais é a obtenção de curvas de remanso (Chow, 1959; French, 1986). 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 Dias (1995), 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, Lin e Gray (1971). Coincidemente, o exemplo utilizado por Lin e Gray (1971) é o mesmo apresentado em Dias (1995), e discutido aqui!!

Obtenção da equação diferencial do problema A figura 2.18 mostra as características gerais de um canal. A equação dinâmica de escoamento em canais em regime permanente é

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

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

$$-\frac{Q^2}{A^2} \frac{dA}{dx} + gA \frac{dZ}{dx} + gAS_f = 0, \quad (2.54)$$

A derivada dA/dx em (2.54) é uma derivada total. A área molhada por outro lado é função de x e Z . Então, sendo B a largura superficial,

$$\frac{dA}{dx} = \frac{\partial A}{\partial x} \Big|_z + \frac{\partial A}{\partial Z} \Big|_x \frac{dZ}{dx} = \frac{\partial A}{\partial x} \Big|_z + B \frac{dZ}{dx} \quad (2.55)$$

Portanto,

$$\frac{dZ}{dx} = \frac{\frac{Q^2}{A^2} \frac{\partial A}{\partial x} \Big|_z - gAS_f}{gA - \frac{Q^2}{A^2} B} = f(x, Z) \quad (2.56)$$

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^a ordem (Press et al., 1992). Métodos mais tradicionais em hidráulica envolvem a solução numérica (por diferenças finitas) para x como variável independente da equação diferencial

$$\frac{dx}{dh} = \frac{1 - F^2}{S_0 - S_f} \quad (2.57)$$

onde h é a profundidade média do escoamento e F é o número de Froude (French, 1986, 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 French (1986, p. 218–222).

As principais vantagens de utilizar a equação (2.56) 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.

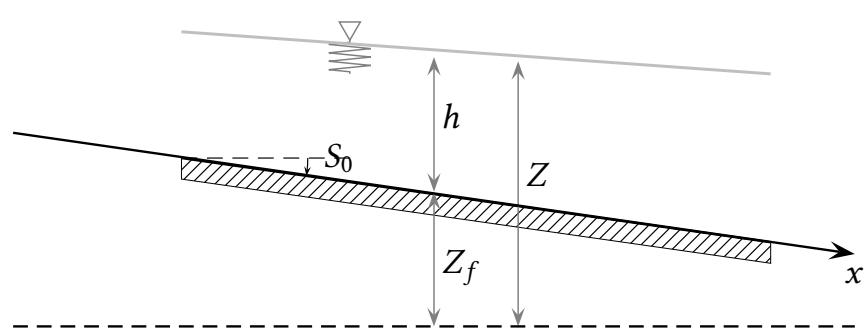
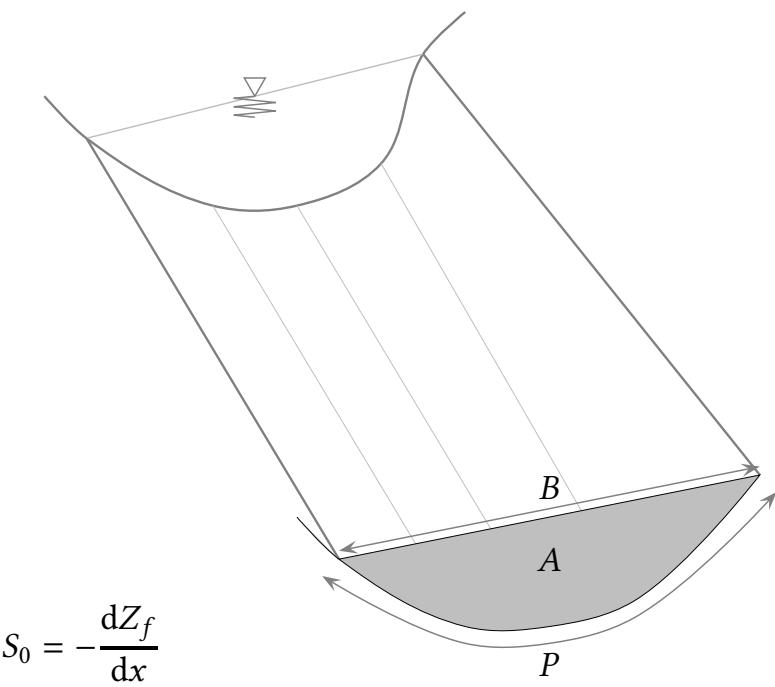


Figure 2.18: Características geométricas de um canal.

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

seção	x m	A m^2	B m	R m	S_f –	V m s^{-1}	Z 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 montante e a jusante do ponto x .

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

$$\left. \frac{\partial A}{\partial x} \right|_Z \approx \frac{A(x_j, Z) - A(x_m, Z)}{x_j - x_m}, \quad (2.58)$$

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 [Chow \(1959\)](#). 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 [Chow \(1959\)](#), 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 (2.53) pressupõe que a velocidade é positiva no sentido positivo dos xs 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:

$$Z_f = -S_0 x, \quad (2.59)$$

$$h = Z - Z_f, \quad (2.60)$$

$$B = b_f + 4h, \quad (2.61)$$

$$P = b_f + 2 * \sqrt{5}h, \quad (2.62)$$

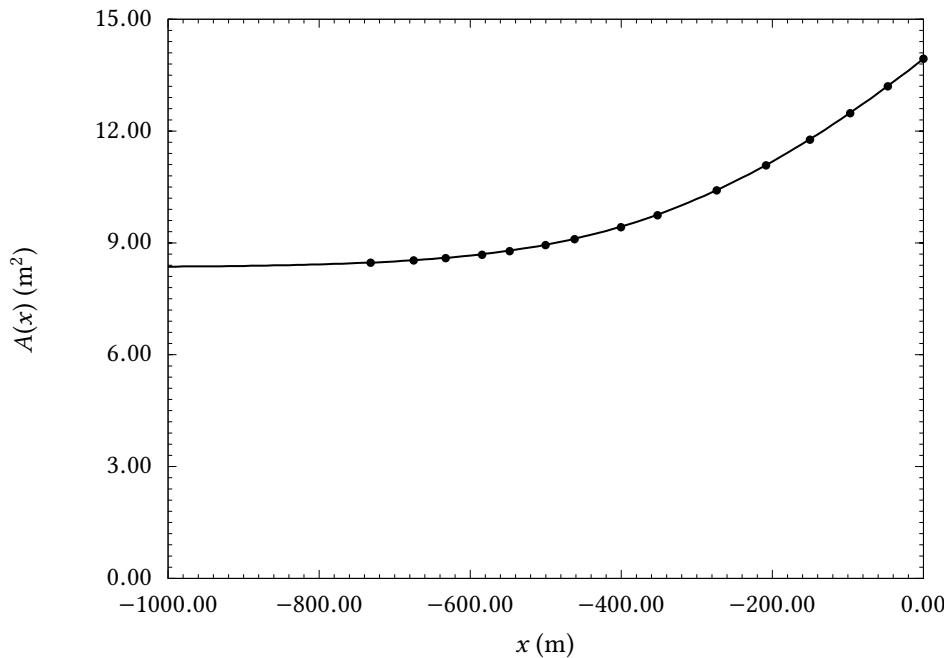


Figure 2.19: Área molhada em função da distância de jusante do exemplo 10.1 de Chow (1959), calculados por Chow (círculos) *versus* resultados do método de Runge-Kutta (linha contínua).

$$A = (B + b_f)h/2, \quad (2.63)$$

$$R = A/P, \quad (2.64)$$

$$V = Q/A, \quad (2.65)$$

$$S_f = \frac{n^2 V^2}{R^{4/3}} \quad (2.66)$$

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 2.19–2.22 contendo as curvas calculadas $A(x)$, $S_f(x)$, $V(x)$ e $Z(x)$ e os pontos correspondentes da tabela 2.1 para comparação.

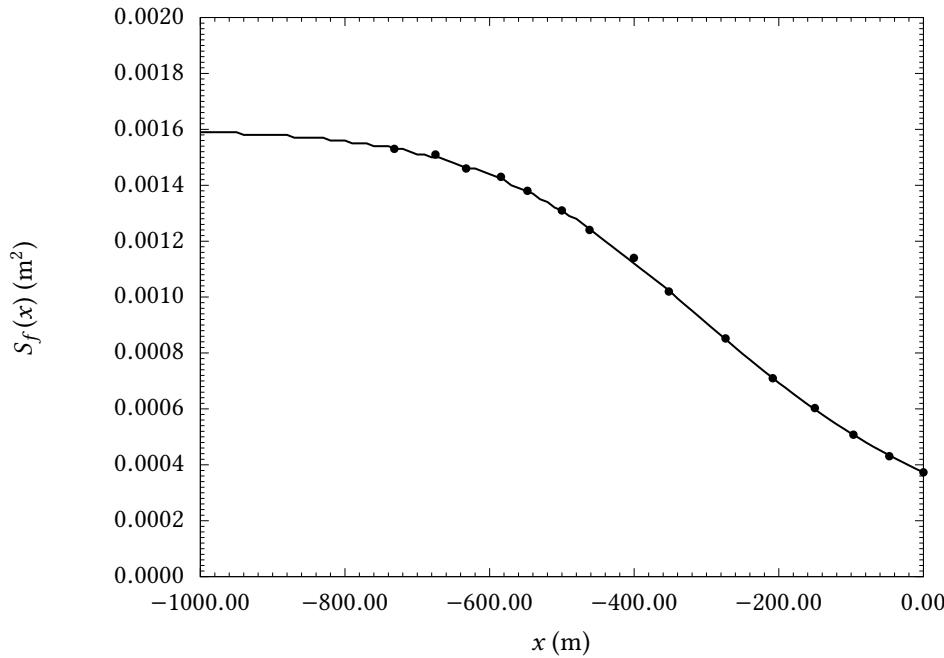


Figure 2.20: Perda de carga em função da distância de jusante do exemplo 10.1 de Chow (1959), calculados por Chow (círculos) *versus* resultados do método de Runge-Kutta (linha contínua).

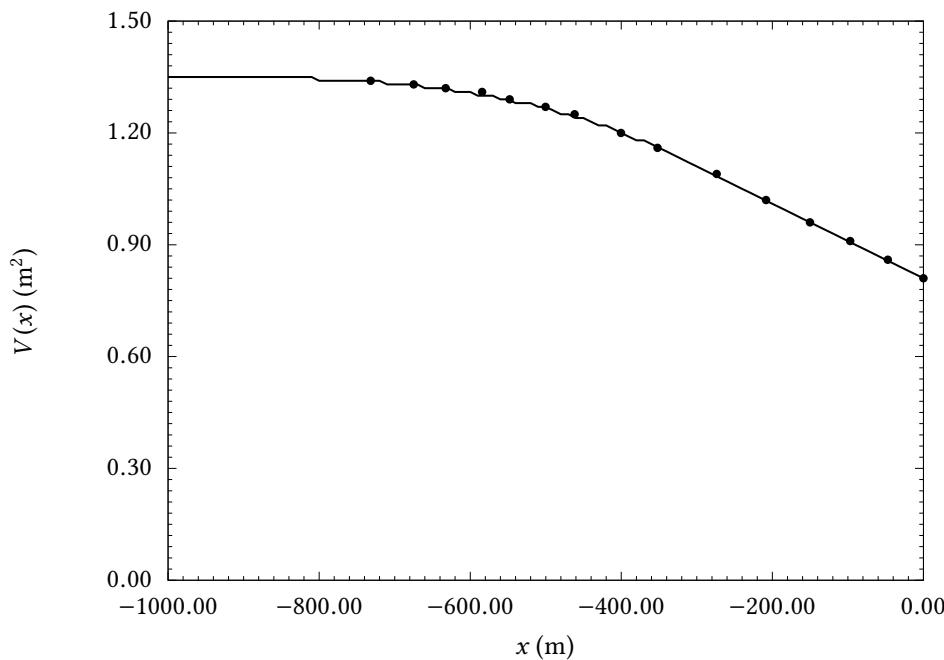


Figure 2.21: Velocidade em função da distância de jusante do exemplo 10.1 de Chow (1959), calculados por Chow (círculos) *versus* resultados do método de Runge-Kutta (linha contínua).

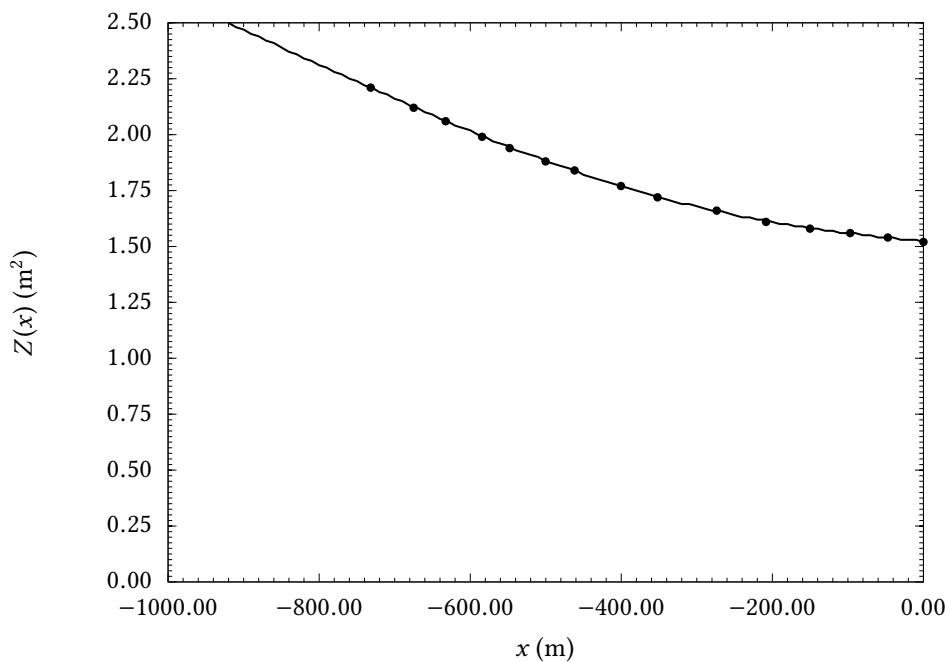


Figure 2.22: Cota em função da distância de jusante do exemplo 10.1 de [Chow \(1959\)](#), calculados por Chow (círculos) *versus* resultados do método de Runge-Kutta (linha contínua).



Figure 2.23: A traça (*Spruce budworm*) *Choristoneura orae*. Fonte: Wikipedia. ©entomart (<http://www.entomart.be>).

Table 2.2: Parâmetros do modelo de Ludwig et al. (1978)

Parâmetro	Unidade	Valor
r_B	ano ⁻¹	1.52
K'	larvas galho ⁻¹	355
β	larvas acre ⁻¹ ano ⁻¹	43200
α'	larvas galho ⁻¹	1.11
r_S	ano ⁻¹	0.095
K_S	galhos acre ⁻¹	25440
K_E	–	1
r_E	ano ⁻¹	0.92
P'	larva ⁻¹	0.00195
T	–	0.05

Um modelo ecológico para *Choristoneura orae*

A *Choristoneura orae* (figura 2.23) é uma traça conhecida nos EUA como *Spruce Budworm*. Ludwig et al. (1978) propuseram um modelo matemático simples de crescimento da traça: se B é a sua densidade populacional, o modelo é

$$\frac{dB}{dt} = r_B B \left(1 - \left(\frac{B}{K'S} \right) \frac{T^2 + E^2}{E^2} \right) - \frac{\beta}{K_S} \left(\frac{B^2}{(\alpha'S)^2 + B^2} \right), \quad (2.67)$$

$$\frac{dS}{dt} = r_S S \left(1 - \frac{SK_E}{E} \right), \quad (2.68)$$

$$\frac{dE}{dt} = r_E E \left(1 - \frac{E}{K_E} \right) - \left(\frac{P'B}{S} \right) \frac{E^2}{T^2 + E^2}. \quad (2.69)$$

Todos os parâmetros são definidos em (Ludwig et al., 1978). As variáveis que evoluem são B (a densidade de larvas, em larvas galho⁻¹), 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

Utilize as condições iniciais para o sistema, $B(0) = 1$, $S(0) = 0.070$ e $E(0) = 1$. Resolva o sistema de equações (2.67)–(2.69), utilizando um passo de tempo de 1/400 ano e um tempo total de simulação de

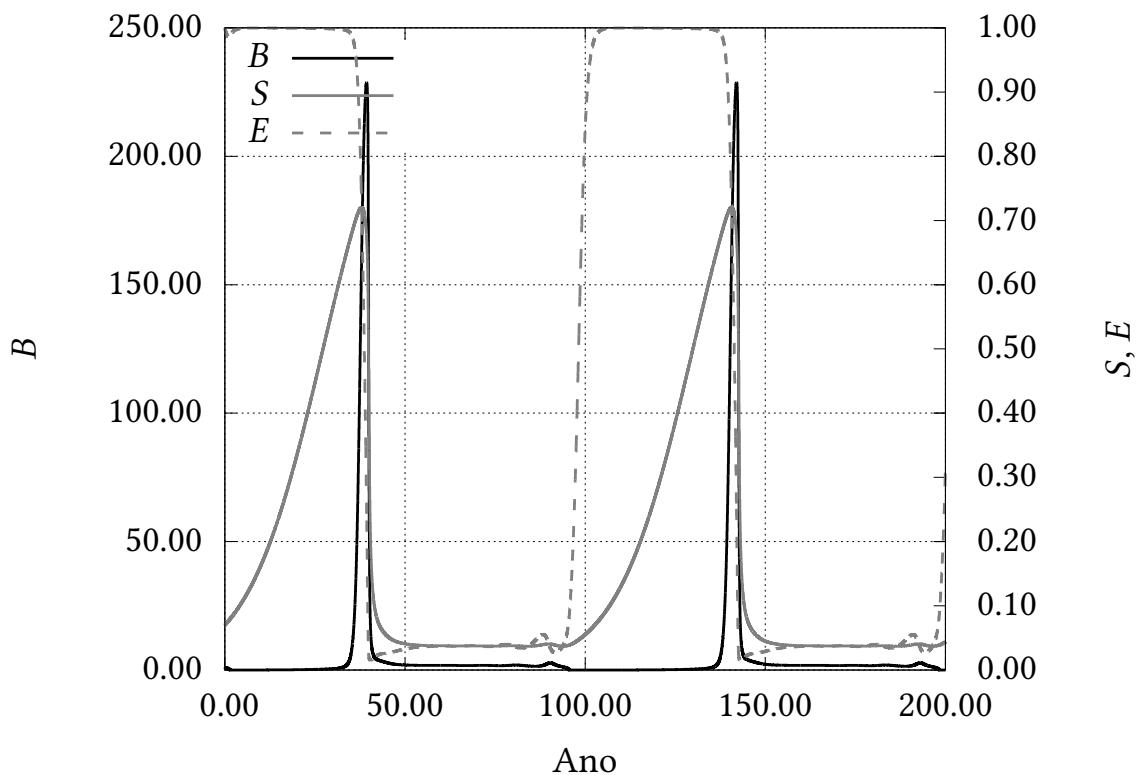


Figure 2.24: Simulação de eclosão de uma praga de traças, utilizando o modelo de [Ludwig et al. \(1978\)](#).

200 anos. O seu resultado deve ser o mostrado na figura 2.24. Compare com a figura 5 de [Ludwig et al. \(1978\)](#).

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 2.25

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 μ de 0,1 em 0,1, desde 0,1 até 10. Plote agora o período T e a amplitude A contra μ . Você deve comparar seus resultados com os obtidos por [Fisher \(1954\)](#).

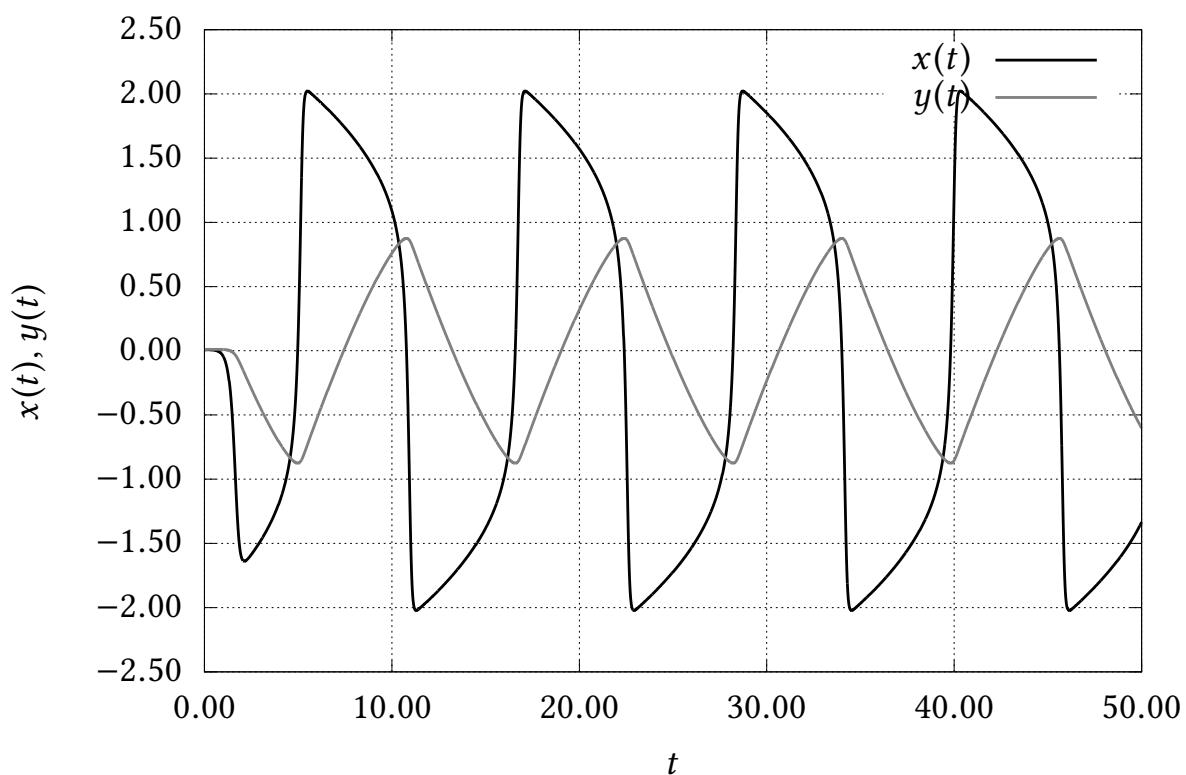


Figure 2.25: O oscilador não-linear de van der Pol ($\mu = 5$).

3

Iterative methods and multigrid in one dimension

3.1 – Introduction: a linear 1D problem

Consider the one-dimensional problem

$$\frac{d^2u}{dx^2} - k^2 u(x) = f(x), \quad (3.1)$$

$$u(0) = 0, \quad (3.2)$$

$$u(L) = 0. \quad (3.3)$$

This problem is linear and has an analytical solution. Put $f(x) = x$ and find it:

```
1 (%i1) declare(k,real);
2 (%o1)
3 (%i2) assume( k > 0 ) ;
4 (%o2)
5 (%i3) eqd : 'diff(u,x,2) - k^2*u = x ;
6
7
8 (%o3)
9
10
11 (%i4) ode2(eqd,u,x);
12
13 (%o4)
14
15
```

Hence

$$u(x) = A \cosh(kx) + B \sinh(kx) - \frac{x}{k^2}. \quad (3.4)$$

Bondary conditions give

$$\begin{aligned} A \cosh(0) + B \sinh(0) &= 0 \\ A &= 0, \\ B \sinh(kL) - \frac{L}{k^2} &= 0, \\ B &= \frac{L}{k^2 \sinh(kL)}. \end{aligned}$$

The final analytical solution, therefore, is

$$u(x) = \frac{L}{k^2 \sinh(kL)} \sinh(kx) - \frac{x}{k^2}. \quad (3.5)$$

Direct solution

A direct method (not iterative) is immediately available upon discretization,

$$\begin{aligned} \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} - k^2 u_i &= f_i, \\ u_{i-1} - (2 + k^2 h^2)u_i + u_{i+1} &= h^2 f_i \end{aligned} \quad (3.6)$$

For the coarsest possible grid, this becomes:

$$\begin{aligned} \frac{u_0 - 2u_1 + u_2}{h^2} - k^2 u_1 &= f_1, \\ u_0 - (2 + k^2 h^2)u_1 + u_2 &= h^2 f_1, \\ u_0 = u_2 = 0 &\quad \Rightarrow \\ u_1 &= -\frac{h^2 f_1}{2 + k^2 h^2} \end{aligned} \quad (3.7)$$

Remember that in this case $h = L/2$!

We can go back a little bit and do one more exercise with a direct solver (tridiag) for this case. The two boundary conditions are

$$-(2 + k^2 h^2)u_1 + u_2 = h^2 f_1 \quad (3.8)$$

$$u_{N-2} - (2 + k^2 h^2)u_{N-1} = h^2 f_{N-1}. \quad (3.9)$$

Here is the program:

Listing 3.1: `helm1d.chpl` — Solution of Helmholtz equation in 1D.

```

1 // -----
2 // helm1d: solves d2u/dx2 - k^2u = x
3 // u(0) = 0
4 // u(L) = 0
5 // -----
6 use Time only stopwatch;
7 use Math only sinh, cosh;
8 use IO only openWriter;
9 use tridiag;                                // solver
10 var etime: stopwatch;                         // the timer
11 config const N = 16;                          // start with 17 points
12 config const L = 1.0;                          // and a length of 1
13 config const k = 4.0;                          // k in helmholtz
14 const B = L/(k**2*sinh(k*L));                // B of analytical solution
15 const dx = L/N;                             // delta x
16 const x: [0..N] real = [i in 0..N] i*dx;    // this is an expression forall
17 var u: [0..N] real = 0.0;                    // the unknowns
18 var e: [0..N] real = 0.0;                    // the errors
19 var a,b,c,f: [1..N-1] real;                 // the tridiag matrix and forcing f
20 etime.start();                               // start timing
21 a = 1.0;                                     // lower diagonal
22 b = -(2.0 + dx**2 * k**2);                  // main diagonal
23 c = 1.0;                                     // upper diagonal
24 f = dx**2 * x[1..N-1];                      // the ODE forcing
25 u[0] = 0.0;                                   // left bc
26 u[N] = 0.0;                                   // right bc
27 assert ( N > 2 ) ;                          // make life simple
28 f[1] += 0.0;                                 // just to remember special left BC
29 f[N-1] += 0.0;                               // just to remember special right BC
30 tridiag(a,b,c,f,u[1..N-1]);                 // solve it
31 etime.stop();                                // stop timing
32 writeln("elapsed_time=",etime.elapsed());   // show elapsed time

```

```

33 const founam = "helm1d.out";           // output's name
34 const fou = openWriter(founam);         // open output
35 for i in 0 .. N do {                  // calculate the errors & write
36     var ua = anasol(x[i]);
37     e[i] = abs(u[i] - ua);
38     fou.writef("%8.4dr.%8.4dr.%8.4dr.%12.4er\n",x[i],u[i],ua,e[i]);
39 }
40 fou.close();                         // close output
41 const em = (+ reduce e[1..N-1])/(N-1); // mean absolute error
42 writef("%05i.%12.4er\n",N,em);       // write mean absolute error
43 // -----
44 // the analytical solution
45 // -----
46 proc anasol(
47     const in x: real
48 ): real {
49     return B*sinh(k*x) - x/k**2 ;
50 }
```

Figure 3.1 depicts the solution for $N = 64$ points.

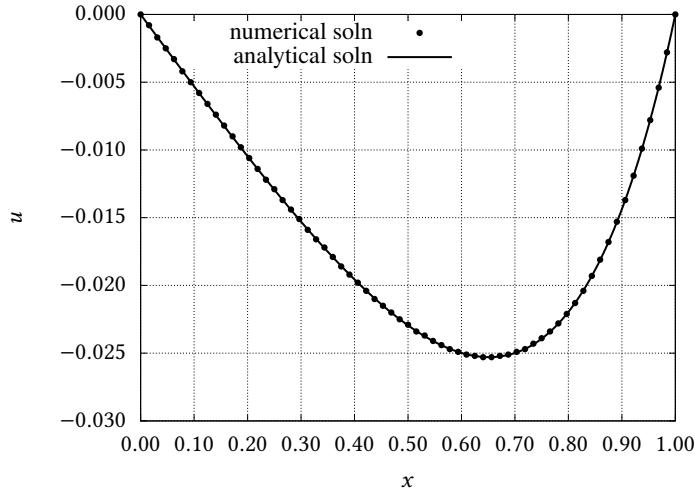


Figure 3.1: Solution of Helmholtz Equation in 1D.

Iterative solution, Jacobi

We now implement the Jacobi Method, 1D:

$$\begin{aligned}
 u_{i-1} - (2 + k^2 h^2)u_i + u_{i+1} &= h^2 f_i, \\
 -(2 + k^2 h^2)u_i &= -(u_{i-1} + u_{i+1}) + h^2 f_i, \\
 u_i &= \frac{1}{2 + k^2 h^2} [u_{i-1} + u_{i+1} - h^2 f_i]
 \end{aligned} \tag{3.10}$$

or, turning this into an iterative scheme,

$$u_{n+1,i} = \frac{1}{2 + k^2 h^2} [u_{n,i-1} + u_{n,i+1} - h^2 f_i] \tag{3.11}$$

As the last line above suggests, the idea of the method is to provide an update $u_{n+1,i}$ as a function of previous values $u_{n,i}$. The algorithm proceeds from an initial (somewhat arbitrary) guess \mathbf{u}_0 until some measure of the difference between successive estimates \mathbf{u}_{n+1} and \mathbf{u}_n is below an acceptable tolerance. Here is the program:

Listing 3.2: jacob1d.chpl — Solution of Helmholtz equation with Jacobi Method in 1D.

```

1 // -----
2 // jacob1d: solves
3 //
4 // d2u/dx2 - k^2u = x
5 // u(0) = 0
6 // u(L) = 0
7 //
8 // with an iterative soln (Jacobi method)
9 //
10 // here we follow "Briggs, W. L.; Henson, V. E. & McCormick, S. F. A multigrid
11 // tutorial SIAM, 2000".
12 //
13 // 2025-02-26T14:40:34 calculating runtime
14 // 2025-08-19T10:43:33 revisiting
15 // 2025-08-19T15:59:15 changing to initial guess
16 // u[0,j] = -0.025*(sin(pi*j/N) + sin(8*pi*j/N))
17 // 2025-09-05T16:15:04 fixing some comments
18 //
19 use IO.FormattedIO;
20 use IO only openWriter;
21 use Time only stopwatch;
22 use Math only pi, sin, sinh;
23 var etime: stopwatch;                                // the timer
24 config const N = 16;                               // start with 17 points
25 config const L = 1.0;                             // and a length of 1
26 config const k = 4.0;                            // k in helmholtz
27 const B = L/(k**2*sinh(k*L));                  // const of analytical soln
28 const dx = L/N;                                 // delta x
29 const x: [0..N] real = [i in 0..N] i*dx;        // this is an expression forall
30 var u: [0..1,0..N] real;                         // (old,new) numerical solution
31 var ua: [0..N] real;                            // analytical solution
32 var e: [0..N] real = 0.0;                        // the errors
33 const f = dx**2 * x[1..N-1];                   // the forcing
34 const maxeps = 1.0e-10;                          // the tolerance
35 var eps = maxeps;                             // the diff between solns
36 //
37 // BCs are very important!
38 //
39 u[0,0] = 0.0;
40 u[1,0] = 0.0;
41 u[0,N] = 0.0;
42 u[1,N] = 0.0;
43 //
44 // This is a surprisingly interesting initial guess for the problem at hand
45 //
46 for j in 1..N-1 do {
47     u[0,j] = -0.025*(sin(pi*j/N) + sin(8*pi*j/N));
48 }
49 var iold = 0;
50 var inew = 1;
51 //
52 // Calculate the analytical solution, using promotion.
53 //
54 ua = anasol(x);
55 //
56 // open error vs iteration file
57 //
58 var ferr = openWriter("ferr.out");
59 //
60 // and start to iterate
61 //
62 const den = (2.0 + k**2 * dx**2);

```

```

63 etime.start();                      // start timing
64 var count = 0;                      // iteration count
65 while eps >= maxeps do {
66     count += 1;
67     var tospit = (count == 10 || count == 100 || count == 250);
68     if tospit then {
69         var spitname = "jacob1d-spit-%04i".format(count);
70         Spit(spitname);
71     }
72     for i in 1..N-1 do {
73         u[inew,i] = (u[iold,i-1] + u[iold,i+1] - f[i])/den;
74     }
75     var err = max reduce abs(u[inew,0..N] - ua) ;
76     ferr.writef("%08i.%16.12dr\n",count,err);
77     eps = max reduce abs(u[inew,1..N-1] - u[iold,1..N-1]);
78     inew <=> iold;
79 }
80 etime.stop();                        // stop timing
81 ferr.close();                       // close error output
82 writeln("#_of_iterns_=~,count);      // how many iterations?
83 writeln("elapsed_time_=~,etime.elapsed()); // how much time?
84 inew <=> iold;                     // exchange new and old
85 const founam = "jacob1d.out";       // output's name
86 const fou = openWriter(founam);      // open output ,locking=false);
87 for i in 0 .. N do {               // calculate the errors & write
88     e[i] = abs(u[inew,i] - ua[i]);
89     fou.writef("%8.4dr.%8.4dr.%8.4dr.%12.4er\n",x[i],u[inew,i],ua[i],e[i]);
90 }
91 fou.close();                        // close output
92 const em = (+ reduce e[1..N-1])/(N-1); // absolute mean error
93 writef("%05i.%12.4er\n",N,em);      // write error
94 // -----
95 // the analytical solution
96 // -----
97 proc anasol(
98     const in x: real
99 ): real {
100    return B*sinh(k*x) - x/k**2 ;
101 }
102 // -----
103 // --> Spit: spits the error at the current iteration
104 // -----
105 proc Spit(
106     const in name: string           // name of output file
107 ) {
108     const nomout = name + ".out";
109     const fou = openWriter(nomout);
110     for i in 0..N do {
111         var ei = ua[i] - u[inew,i];
112         fou.writef("%10.6r.%10.6r\n",x[i],ei);
113     }
114     fou.close();
115 }
```

Figure 3.2 shows the initial guess used, against the analytical and numerical solutions solution. The initial guess is chosen intentionally so that there are relatively “high-frequency” errors in it.

Jacobi takes a long time to achieve the desired accuracy. Theoretically, we know that iterative methods are *better* than direct methods, but $N = 64$ is just two small and the superiority of iterative methods is still not apparent.

In general, direct methods are $\mathcal{O}(N^3)$, and iterative methods are $\mathcal{O}(N^2)$. But there is something even better. Multigrid is $\mathcal{O}(N)$. Let’s look at it from a practical point of view.

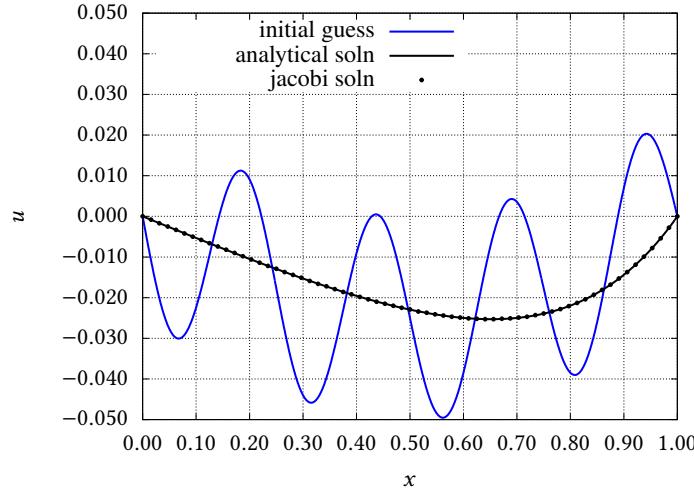


Figure 3.2: Jacobi method: initial guess, analytical, and numerical solution.

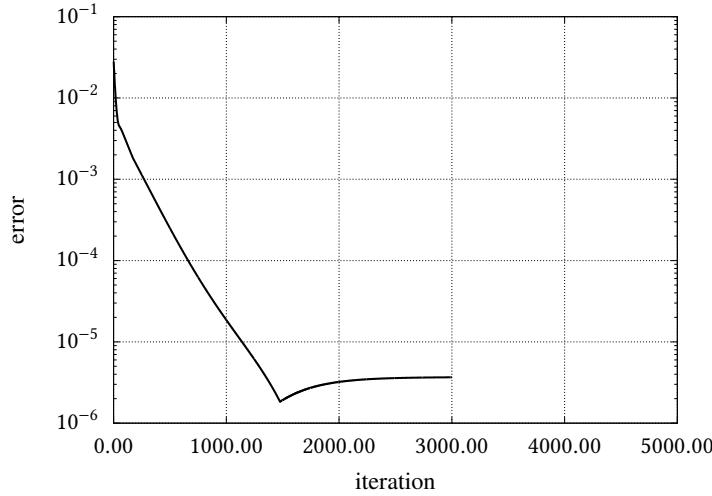


Figure 3.3: Error of the Jacobi method as the iterations progress. Note the leveling of the error.

The first thing is to look at how fast Jacobi converges. To do that, we have made the program print the mean absolute error as a function of where the iteration is. This is plotted in figure 3.3. After some time, the error “levels off”, and this explains why iterative methods are still relatively slow.

Iterative methods such as Jacobi are good at eliminating the high-frequency component of the error fast, but the low-frequency components remain for a long time. This is seen in figure 3.4. Therefore, after a few iterations that eliminate the high-frequency components, the low-frequency ones take a long time to eliminate, because they are being calculated at many more points than are needed to resolve the low-frequency components (Briggs et al., 2000, Chapter 2).

Weighted Jacobi

Weighted Jacobi is a small change to the Jacobi method that looks a little bit like split operator techniques that will appear later on. The idea is to transform (3.10) into an intermediate step

$$u^* = \frac{1}{2 + k^2 h^2} [u_{n,i-1} + u_{n,i+1} - h^2 f_i], \quad (3.12)$$

and then weigh this with the previous estimate:

$$u_{n+1,i} = (1 - \omega)u_{n,i} + \omega u^*. \quad (3.13)$$

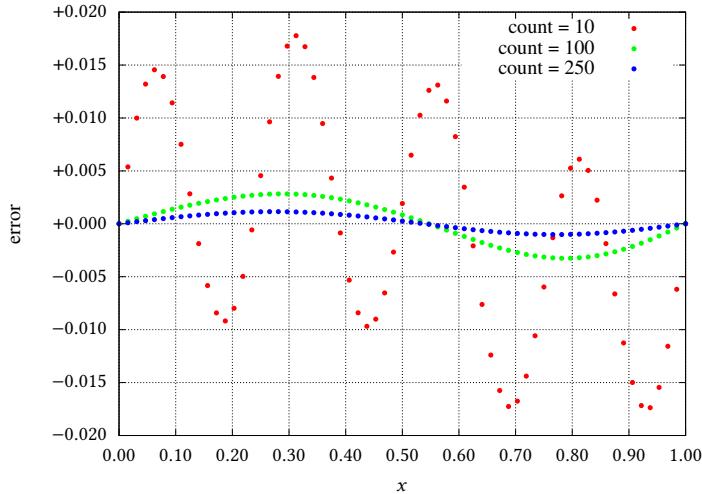


Figure 3.4: Pointwise error of the Jacobi method for the 10th, 100th and 250th iterations.

In (Briggs et al., 2000, Chapter 2) it is shown that $\omega = 2/3$ is the optimal value for the 1D Helmholtz equation. The changes to `jacob1d.chpl` needed to implement Weighted Jacobi are trivial, so we will not show the corresponding `jacob1dw.chpl` code. However, it is interesting to check if the weighted version converges faster. Interestingly, the number of iterations (4291×3000) *increases* to achieve convergence with the same tolerance! For some reason, Weighted Jacobi is the building block used by Briggs et al. (2000) in the multigrid method, so we will keep it for the time being.

3.2 – Multigrid

Equation 3.6 is a system of linear equations. It is useful to write it in operator form as either one of

$$\mathbf{A} \cdot \mathbf{u}_e = \mathbf{f}, \quad (3.14)$$

$$\mathbf{A} \cdot \mathbf{u} \approx \mathbf{f}, \quad (3.15)$$

where \mathbf{u}_e is the exact solution and \mathbf{u} is an approximation. With them, we define the error \mathbf{e} and the residual \mathbf{r} :

$$\mathbf{e} \equiv \mathbf{u}_e - \mathbf{u}, \quad (3.16)$$

$$\mathbf{r} \equiv \mathbf{f} - \mathbf{A} \cdot \mathbf{u}. \quad (3.17)$$

Several norms can be used to estimate errors and residuals, including

$$\|\mathbf{x}\|_\infty \equiv \max_i |x_i|, \quad (3.18)$$

$$\|\mathbf{x}\|_2 \equiv \left[\sum_{i=1}^n x_i^2 \right]^{1/2}. \quad (3.19)$$

For the time being, we will be using mostly (3.18) in the numerical implementations.

Now consider the sequence

$$\begin{aligned} \mathbf{A} \cdot \mathbf{e} &= \mathbf{A} \cdot [\mathbf{u}_e - \mathbf{u}] \\ &= \mathbf{A} \cdot \mathbf{u}_e - \mathbf{A} \cdot \mathbf{u} \\ &= \mathbf{f} - \mathbf{A} \cdot \mathbf{u} \Rightarrow \\ \mathbf{A} \cdot \mathbf{e} &= \mathbf{r}. \end{aligned} \quad (3.20)$$

(3.20) is the *residual equation*.

Both (3.15) and 3.20 will need to be solved in the sequence by a few iterations. This can be better understood in operator form by decomposing \mathbf{A} as follows:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}. \quad (3.21)$$

Above, \mathbf{D} is a diagonal matrix, \mathbf{L} is a matrix with non-zero values below the diagonal, and \mathbf{U} is a matrix with non-zero values above the diagonal. Now,

$$\begin{aligned} \mathbf{A} \cdot \mathbf{u} &= f, \\ [\mathbf{D} + \mathbf{L} + \mathbf{U}] \cdot \mathbf{u} &= f, \\ \mathbf{D} \cdot \mathbf{u} &= f - [\mathbf{L} + \mathbf{U}] \cdot \mathbf{u}. \end{aligned}$$

For two successive iterations \mathbf{u}_n and \mathbf{u}_{n+1} of the Jacobi method, then, we put

$$\mathbf{u}_{n+1} = \mathbf{D}^{-1} \cdot [f - [\mathbf{L} + \mathbf{U}] \cdot \mathbf{u}_n]. \quad (3.22)$$

The left side involves only elementary operations, since \mathbf{D}^{-1} is also diagonal such that (the parentheses around the indices emphasize that indicial notation is not to be used)

$$D_{(ii)}^{-1} = \frac{1}{D_{(ii)}}.$$

It is straightforward to generalize now to weighted Jacobi using operator notation:

$$\mathbf{u}_{n+1} = \omega \mathbf{D}^{-1} \cdot [f - [\mathbf{L} + \mathbf{U}] \cdot \mathbf{u}_n] + (1 - \omega) \mathbf{u}_n \quad (3.23)$$

A single iteration, or application of (3.22) or (3.23) is called a “relaxation”; there are other possibilities than the weighted Jacobi method implied in (3.23), but in the sequence we will use weighted Jacobi as a concrete example of relaxation. We will refer in general to any kind of relaxation by

$$\mathbf{u}_{n+1} = \mathbf{X}(\mathbf{u}_n). \quad (3.24)$$

Next, we need some terminology to explain the multigrid method of solving systems of linear equations.

Prolongation (= Interpolation)

First, we need to extend the definition of a general array \mathbf{v} to several resolutions. We start with $N = 2^{G+1}$ intervals ($N + 1$ points including the extremities $x = 0$ and $x = L$) and systematically look at lower resolutions. In our convention $g = 0$ means the finest (initial) resolution with N intervals. The number of intervals in each resolution is n_g , so that $n_0 = N$. At each stage we divide the number of intervals by 2 until $g = G$, which is the coarsest resolution such that $n_G = 2$ (see table 3.1). The array at resolution g is going to be denoted by \mathbf{v}^g , with $n_g + 1$ points. *Prolongation is simply interpolation* (for now at least). We write it in operator form as

$$\mathbf{v}^{g-1} = \mathbf{P}^g \cdot \mathbf{v}^g, \quad g = 1, \dots, G \quad (3.25)$$

Pointwise, the rightmost point is simply repeated:

$$v_{n_{g-1}}^{g-1} = v_{n_g}^g, \quad (3.26)$$

while the leftmost and interior points are effectively interpolated:

$$v_{2i}^{g-1} = v_i^g, \quad (3.27)$$

$$v_{2i+1}^{g-1} = \frac{1}{2} (v_i^g + v_{i+1}^g), \quad (3.28)$$

for $i = 0, \dots, n_g - 1$.

Table 3.1: Number of intervals $n_g = 2^g$ at each resolution g .

2^g	n_g
2^{G+1}	n_0
2^G	n_1
\vdots	\vdots
2^1	n_G

Restriction (= Averaging)

Restriction is the opposite of prolongation. In operator form, we will write

$$\mathbf{v}^{g+1} = \mathbf{R}^g \cdot \mathbf{v}^g, \quad g = 0, \dots, G-1. \quad (3.29)$$

Restriction can be understood as a low-pass filtering from a higher resolution array \mathbf{v}^g to a lower resolution array \mathbf{v}^{g+1} . Pointwise, we again repeat the endpoints

$$v_0^{g+1} = v_0^g, \quad (3.30)$$

$$v_{n_{g+1}}^{g+1} = v_{n_g}^g, \quad (3.31)$$

and average the inner points:

$$v_i^{g+1} = \frac{1}{4} (v_{2i-1}^g + 2v_{2i}^g + v_{2i+1}^g), \quad 1 \leq i \leq n_{g+1} - 1. \quad (3.32)$$

Two-grid correction scheme

Look again at figure 3.4: after a few iterations of the Jacobi method, the high-frequency error is eliminated. Then, trying to iterate to reduce the low-frequency error is inefficient, because it involves too many points that are superfluous to represent the low frequencies. The idea then is to iterate once on the fine grid, and then iterate in the next coarser grid (by restriction) with only half as many unknowns. This will accelerate the process, and we can get back to the original grid with prolongation. Briggs et al. (2000, p. 37) call this the “Two-grid correction scheme”. We write the corresponding algorithm here (using our notation) as

1. Relax v_1 times on grid g from \mathbf{u}_0^g :

$$\begin{aligned} \mathbf{u}_1^g &= X(\mathbf{u}_0^g), \\ \mathbf{u}_2^g &= X(\mathbf{u}_1^g), \\ &\vdots \\ \mathbf{u}_{v_1}^g &= X(\mathbf{u}_{v_1-1}^g). \end{aligned}$$

2. Calculate the fine-grid residual

$$\mathbf{r}^g = \mathbf{f}^g - \mathbf{A}^g \cdot \mathbf{u}_{v_1}^g.$$

3. Restrict the residual:

$$\mathbf{r}^{g+1} = \mathbf{R}^g \cdot \mathbf{r}^g.$$

4. Solve for \mathbf{e}^{g+1} (maybe approximately)

$$\mathbf{A}^{g+1} \cdot \mathbf{e}^{g+1} = \mathbf{r}^{g+1}.$$

5. Prolong the error from $g+1$ back to g :

$$\mathbf{e}^g = \mathbf{P}^{g+1} \cdot \mathbf{e}^{g+1}.$$

u_0^0	u_1^0	u_2^0	u_3^0	u_4^0	u_5^0	u_6^0	u_7^0	u_8^0	u_9^0	u_{10}^0	u_{11}^0	u_{12}^0	u_{13}^0	u_{14}^0	u_{15}^0	u_{16}^0	u_0^1	u_1^1	u_2^1	u_3^1	u_4^1	u_5^1	u_6^1	u_7^1	u_8^1	u_0^2	u_1^2	u_2^2	u_3^2	u_4^2	u_0^3	u_1^3	u_2^3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
$g = 0$						$g = 1$						$g = 2$						$g = 3$															
↔						↔						↔						↔															

Figure 3.5: Multigrid static storage strategy.

6. Correct the available approximation at scale g :

$$\boldsymbol{u}_{v_1}^g \leftarrow \boldsymbol{u}_{v_1}^g + \boldsymbol{e}^g.$$

Steps 5 and 6 above are crucial: because we are estimating *errors* from $g = 1$ onwards, these errors can be prolonged to the finer g grid and used to *update* the former estimate. This provides a natural way to update when we are moving back to grid g .

7. Rename the approximation (we are reusing the symbols here, for economy, without loss of clarity!)

$$u_0 = u_{\nu_1}^g,$$

and relax v_2 times:

$$\begin{aligned} \mathbf{u}_1^g &= X(\mathbf{u}_0^g), \\ \mathbf{u}_2^g &= X(\mathbf{u}_1^g), \\ &\vdots \\ \mathbf{u}_{v_2}^g &= X(\mathbf{u}_{v_2-1}^g). \end{aligned}$$

The only thing that is not completely specified (mathematically) in the steps above is item 4, the solution for the error e^{g+1} . Suppose now that we keep moving *down* (in resolution) from grid g to grid $g+1$ (starting at $g = 0$) until we reach grid G , which only has 3 points and two intervals. For the Helmholtz 1D equation that is being used as an example, there is now a trivial solution given by (3.7). Therefore, all we need to do now is to extend the two-grid scheme outlined above all the way to the coarsest grid; find the trivial solution to the error e^G , and move up back grid by grid. We have just described the simple multigrid *V-cycle*.

For the V-cycle, we will need to store \mathbf{u}^g for all g . We will implement this by allocating a *single* array representing

$$\begin{aligned} [u_0^0, u_1^0, \dots, u_N^0] &\rightarrow N+1 \text{ points,} \\ [u_0^1, u_1^1, \dots, u_{N/2}^1] &\rightarrow N/2+1 \text{ points,} \\ &\vdots \\ [u_0^G, u_1^G, u_2^G] &\rightarrow 2+1 \text{ points.} \end{aligned}$$

This is depicted in figure 3.5. The total memory needed is

$$\begin{aligned} \sum_{g=1}^{G+1} 2^g + (G+1) &= 2^{G+2} - 2 + (G+1) \\ &= 2^{G+2} + G - 1. \end{aligned}$$

Let us write this V-cycle algorithm now:

- a) Downward towards coarser grids. For $g = 0, 1, \dots, G - 1$:

1. Relax v_1 times on grid g from \mathbf{u}_0^g . Obtain $\mathbf{u}_{v_1}^g$:

$$\mathbf{u}_{v_1}^g = X(X(\dots X(\mathbf{u}_0^g))).$$

A single relaxation using the weighted Jacobi method is very straightforward and is done by the procedure `RelaxWJ` in program `vcycle`, to be presented in the sequence.

2. Calculate the fine-grid residual

$$\mathbf{r}^g = \mathbf{f}^g - \mathbf{A}^g \cdot \mathbf{u}_{v_1}^g.$$

3. Restrict the residual:

$$\mathbf{r}^{g+1} = \mathbf{R}^g \cdot \mathbf{r}^g.$$

Steps 2 and 3 above can be done in the same procedure (called `Restrict` in program `vcycle`). Actually, storage for both \mathbf{r}^{g+1} and \mathbf{e}^{g+1} defined for the two-correction scheme is superfluous, and for the computational implementation we can set

$$\begin{aligned}\mathbf{r}^{g+1} &\equiv \mathbf{f}^{g+1}, \\ \mathbf{e}^{g+1} &\equiv \mathbf{u}^{g+1}.\end{aligned}$$

Differently from the two-grid correction, we *do not* solve for \mathbf{e}^{g+1} ; instead, we will go to the next g in the loop and relax over the $\mathbf{u}^{g+1} \equiv \mathbf{e}^{g+1}$ using \mathbf{f}^{g+1} obtained above as the forcing.

- b) Solve for $\mathbf{e}^G \equiv \mathbf{u}^G$ on the 2-interval grid G using equation (3.7).

- c) Upward towards finer grids. For $g = G, \dots, 1$:

1. Prolong \mathbf{u}^g to grid $g - 1$ and *correct* \mathbf{u}^{g-1} :

$$\mathbf{u}_0^{g-1} \leftarrow \mathbf{u}^{g-1} + \mathbf{P}^g \cdot \mathbf{u}^g.$$

2. Relax v_2 times on \mathbf{u}_0^{g-1} to obtain $\mathbf{u}_{v_2}^{g-1}$:

$$\mathbf{u}_{v_2}^{g-1} = X(X(\dots X(\mathbf{u}_0^{g-1}))).$$

The end result is a new estimate of \mathbf{u}^0 (the fine grid solution that we are seeking), at the end of this V-cycle. It is important to note that \mathbf{u}^1 is a *correction* to the solution \mathbf{u}^0 ; \mathbf{u}^2 again is a *correction* to \mathbf{u}^1 , and so forth. This means that the end effect on the estimate of the solution can only be “seen” at the end of the V-cycle, when grid 0 has been corrected. Moreover, a single V-cycle may not be enough to achieve the desired accuracy; by trial and error, we can repeat the whole algorithm above a few times (say, n_v).

We are now in the position to present the program `vcycle.chpl` in listing 3.3.

The crucial connection between grids is in the `Prolong` procedures, in lines 243, 244 and 251, where the array `u` is updated with the prolonged values from the coarser grid.

Figure 3.6 shows the result of the multigrid procedure applied to a single V-cycle (the first).

To increase accuracy, we chose to repeat the V-cycle 4 times. The final solution is shown in figure 3.7.

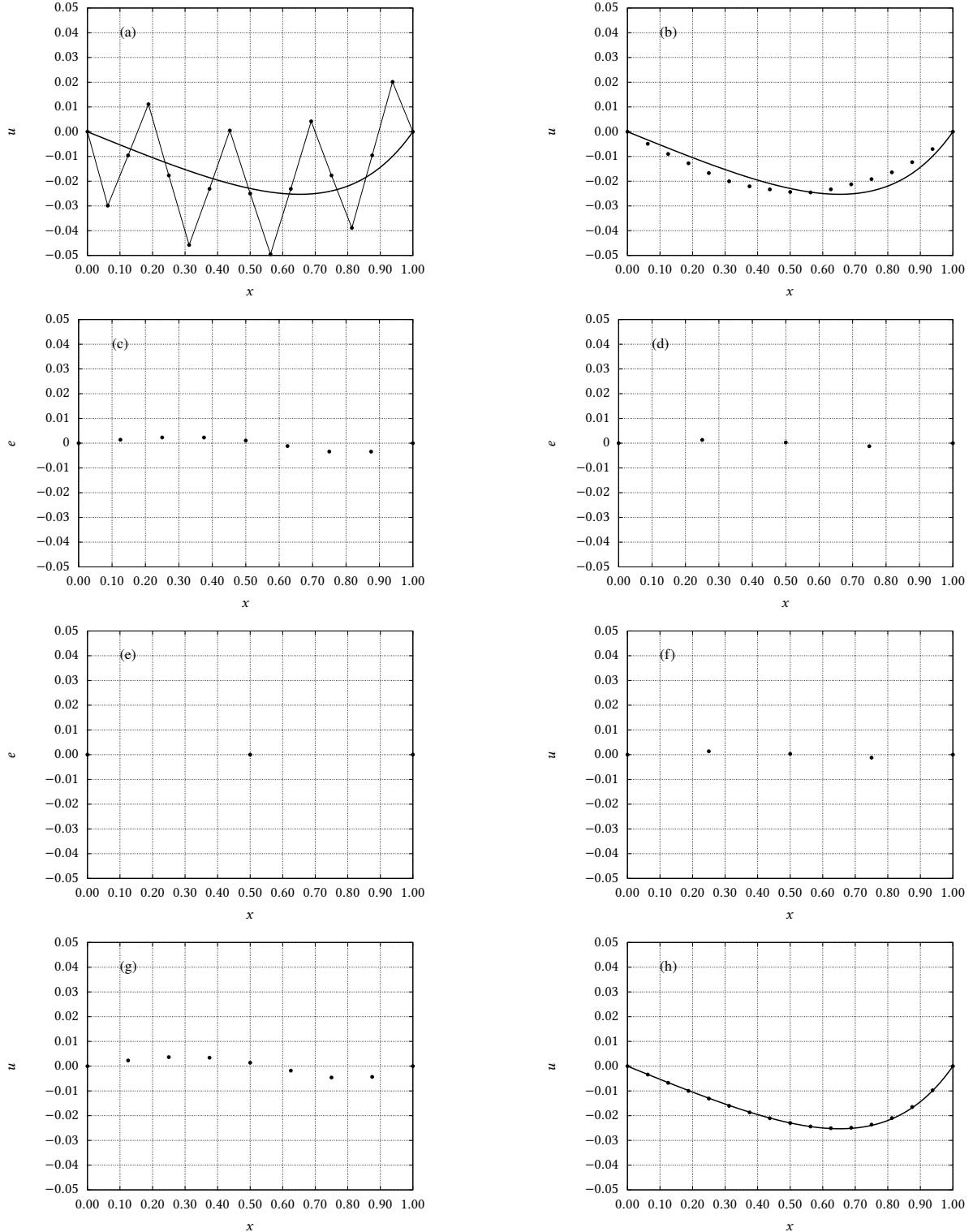


Figure 3.6: The multigrid method for the Helmholtz 1D equation for a grid of 17 points ($N = 16$). The solid line is the analytical solution. All datapoints (except for (e)) are shown after 4 relaxations. (a): initial guess (note the high-frequency oscillations around the analytical solution). (b): downward solution for grid 0. (c): downward error corrections for grid 1. (d): downward error corrections for grid 2. (e): algebraic solution for the error in grid 3 using (3.7). (f): upward error correction for grid 2. (g): upward error correction for grid 1. (h): upward solution for grid 0 at the end of one V-cycle.

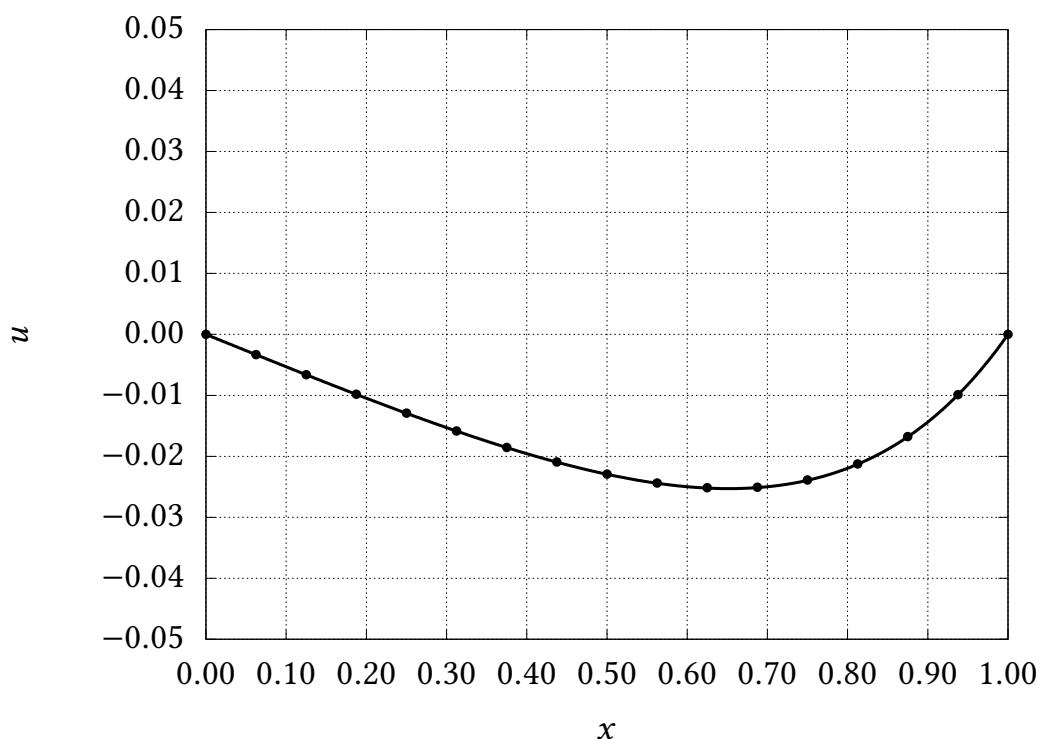


Figure 3.7: Solution of the Helmholtz 1D equation with the multigrid method after 4 V-cycles.

Listing 3.3: `vcyle.chpl` — Solution of the 1D Helmholtz equation using V-cycles.

```

1 // -----
2 // Use of V-cycles to solve the Helmholtz eqn
3 // -----
4 use IO only openWriter;
5 use IO.FormattedIO;
6 use Time only stopwatch;
7 use Math only log2, pi, sin;
8 use nchpl;                                // provides the reverse function
9 config const N = 16;                      // N + 1 grid points
10 config const L = 1.0;                     // and a length of 1
11 config const k = 4.0;                     // k in helmholtz
12 const dx = L/N;                          // this is dx for the finest grid
13 const x = [i in 0..N] i*dx;              // this is an expression forall
14 const nudown = 4;                        // number of sweeps as we go down
15 const nuup = 4;                          // number of sweeps we come up
16 var etime: stopwatch;                   // elapsed time
17 // -----
18 // begin program: how many grids?
19 // -----
20 var ig = 0 ;                            // initially index of finest grid
21 var nn = N ;                           // length of grid ig
22 while nn > 2 do {
23     nn /= 2 ;
24     ig += 1 ;
25 }
26 const G = ig;                         // index of coarsest grid
27 writeln("#_of_grids_=_" ,G+1);        // the number of grids
28 writeln("log_2(N)_=_" ,log2(N));       // are they the same?
29 // -----
30 // Using N, find all first, length, and last indices of each grid inside u and
31 // f.
32 // -----
33 var first: [0..G] int;
34 var last: [0..G] int;
35 var length: [0..G] int;
36 first[0] = 0;
37 last[0] = N;
38 length[0] = N;
39 writeln("-"*80);
40 writeln("first_index,_last_index,_and_length_of_each_grid:");
41 writeln("-"*80);
42 writeln(first[0],"_",last[0],"_",length[0]);
43 // -----
44 // Calculate indices for all grids once and for all: long and detailed, but
45 // clear! Note that it is not too hard to figure out that
46 //
47 // NT + 1 = 2^{G+2} + G - 1
48 //
49 // is the total memory needed for the array u with the numerical solutions over
50 // all grids.
51 // -----
52 for ig in 1..G do {
53     length[ig] = length[ig-1]/2;
54     first[ig] = last[ig-1] + 1;
55     last[ig] = first[ig] + length[ig];
56     writeln(first[ig],"_",last[ig],"_",length[ig]);
57 }
58 const NT = last[G];
59 writeln("NT+1=_" ,NT+1);
60 writeln("calculation_gives_," ,2**((G+2) + G - 1));
61 var u:[0..1,0..NT] real = 0.0;          // old and new solutions
62 var f:[0..NT] real;                    // RHS forcing

```

```

63 var iold = 0;                      // swap old and new solutions ...
64 var inew = 1;                      // in iterative procedure
65 // -----
66 // set the forcing
67 // -----
68 f[1..N-1] = x[1..N-1];           // fill f for finest grid
69 f[1] += 0;                        // remember special left BC
70 f[N] += 0;                        // remember special right BC
71 etime.start();                  // start timing
72 // -----
73 // This is a surprisingly interesting initial guess for the problem at hand
74 // -----
75 for j in 0..N do {
76     u[inew,j] = -0.025*(sin(pi*j/N) + sin(8*pi*j/N));
77 }
78 Spit(0,"vcycle-iniguess");        // print the initial guess
79 // -----
80 // for large N, one V-cycle is not enough: use 4!
81 // -----
82 config const nv = 4;
83 for iv in 1..nv do {
84 // -----
85 // loop over grids
86 // -----
87 var eps = 0.0;
88 for ig in 0..G-1 do {
89 // -----
90 // Relax nudown times on current grid.
91 // -----
92     for k in 1..nudown do {
93         iold <=> inew;
94         eps = RelaxWJ(ig,u,f);
95     }
96     var down:string = "vcycle-down-%02i".format(ig);
97     if iv == 1 then Spit(ig,down);
98 // -----
99 // Now restrict (average). Note that u is a slice!
100 // -----
101 Restrict(ig,u[inew,...],f);
102 // -----
103 // The initial guess for the *error* in the next, coarser, grid is initially 0,
104 // and thereafter inherited from the last V-cycle. This will leave
105 // -----
106 // u[inew,first[ig+1]]=0
107 // u[inew,last[ig+1]] =0
108 // -----
109 // forever
110 // -----
111     if iv == 1 then u[inew,first[ig+1]..last[ig+1]] = 0.0;
112 }
113 // -----
114 // Now solve directly on the coarsest grid for u
115 // -----
116 Solve(G,u,f);
117 var down:string = "vcycle-down-%02i".format(G);
118 Spit(G,down);
119 // -----
120 // Go back, prolonging from coarser to finer. Start at coarsest.
121 // -----
122     for ig in reverse(1..G) do {
123         Prolong(ig,u[inew,...]);
124 }
125 // Relax on the finer grid ig-1 !!!

```

```

126 // -----
127     for k in 1..nuup do {
128         iold <=> inew;
129         eps = RelaxWJ(ig-1,u,f);
130     }
131     var up:string = "vcycle-up-%02i".format(ig-1);
132     if iv == 1 then Spit(ig-1,up);
133 }
134 }
135 etime.stop();
136 writeln("lasted.",etime.elapsed(), "s");
137 // -----
138 // now write the solution that was found
139 // -----
140 Spit(0,"vcycle-lastguess");
141 //
142 // --> Spit: spits a solution at resolution ig
143 //
144 proc Spit(
145     const in ig: int,           // grid
146     const ref name: string    // name of output file
147 ) {
148     const nomout = name + ".out"; // output file name + .out
149     const fou = openWriter(nomout); // open output file
150     const fi = first[ig];        // first index, this grid
151     const nn = length[ig];       // length of this grid
152     const dx = L/nn;             // delta x, this grid
153     const x = [i in 0..nn] i*dx; // positions, this grid
154     for j in 0..nn do {
155         fou.writef("%10.6r.%10.6r\n",x[j],u[inew,fi+j]);
156     }
157     fou.close();                // close output file
158 }
159 //
160 // --> RelaxWJ: one sweep of weighted Jacobi. Relaxes on grid ig
161 //
162 proc RelaxWJ(
163     const in ig: int,           // relax on grid ig
164     ref u: [] real,            // estimates
165     ref f: [] real             // forcing
166 ): real where (u.rank == 2 && f.rank == 1) {
167     const omega = 2.0/3.0;      // best omega for weighted Jacobi
168 //
169 // one loop only!
170 //
171     const fi = first[ig];
172     const la = last[ig];
173     const nn = length[ig];
174     const h = L/nn;              // delta x == h
175     const den = (2.0 + k**2 * h**2); // needed everywhere
176 //
177 // loop on the internal points of this grid only: u[iold/inew,first[ig]] and
178 // u[iold/inew,last[ig]] are the BCs and should remain untouched throughout
179 // the process
180 //
181     for i in fi+1..la-1 do {
182         var ustarc = (u[iold,i-1] + u[iold,i+1] - (h**2)*f[i])/den;
183         u[inew,i] = (1.0-omega)*u[iold,i] + omega*ustarc;
184     }
185     var eps = max reduce abs(u[inew,fi+1..la-1] - u[iold,fi+1..la-1]);
186     return eps;
187 }
188 //

```

```

189 // --> Restrict: "restrict"  $r^{ig}$  to the coarse grid:
190 //
191 //  $f^{ig+1} = R^g[f^g - Au^g]$ . OR:
192 //
193 //  $r^g = f^g - A^g u^g$ 
194 //  $r^{ig+1} = R^g r^g$ 
195 //
196 proc Restrict(
197     const in ig: int,           // current grid coarseness
198     ref u: [] real,            // new solution (given from u[inew,...]), all grids
199     ref f: [] real             // forcings, all grids
200 ) where (u.rank == 1 && f.rank == 1) {
201     const fi = first[ig];      // first index, current grid
202     const nn = length[ig];
203     const h = L/nn;
204     const fiplus1 = first[ig+1]; // first index, next (coarser) grid
205     for j in 1..length[ig+1] - 1 do {
206         //
207         // left, center and right values of Au: note that  $r = f - Au$  is actually
208         // stored in f at the next, coarser, grid
209         //
210         const twoj = 2*j;
211         var Aul = (u[fi + twoj - 2] - 2*u[fi + twoj - 1] + u[fi + twoj])/h**2
212             - k**2 * u[fi + twoj - 1] ;
213         var Auc = (u[fi + twoj - 1] - 2*u[fi + twoj] + u[fi + twoj + 1])/h**2
214             - k**2 * u[fi + twoj] ;
215         var Aur = (u[fi + twoj] - 2*u[fi + twoj + 1] + u[fi + twoj + 2])/h**2
216             - k**2 * u[fi + twoj + 1] ;
217         var j1 = fiplus1+j;
218         f[j1] = ((f[fi + twoj - 1] - Aul) + 2*(f[fi + twoj] - Auc)
219             + (f[fi + twoj + 1] - Aur))/4.0;
220     }
221 }
222 //
223 // --> Prolong: prolongation P is the adjoint of restriction R: will go from ig
224 // (coarser) *down to* ig-1 (finer)
225 //
226 proc Prolong(
227     const in ig: int,           // current grid coarseness
228     ref u: [] real,            // solutions, all grids
229 ) {
230     assert ( (ig > 0) && (ig <= G) );
231     //
232     // straightforward linear interpolation! Note that prolongation/interpolation
233     // is always added to the u in the previous grid, as in
234     //
235     //  $u^{ig-1} \leftarrow u^{ig-1} + \text{prolongation}(u^{ig})$ . Hence the += below!
236     //
237     // for j = 0, we are summing 0 to the leftmost point, which is a BC always
238     // equal to 0 for ig > 0
239     //
240     const fi = first[ig];
241     const fiminus1 = first[ig-1];
242     for j in 0..length[ig] - 1 do {
243         u[fiminus1 + 2*j] += u[fi+j];           // update finer grid
244         u[fiminus1 + 2*j + 1] += (u[fi+j] + u[fi+j+1])/2.0; // update finer grid
245     }
246     //
247     // the rightmost point needs to be "interpolated" as well! Since ig > 0,
248     // however, we are only summing 0.
249     //
250     var j = length[ig];
251     u[fiminus1 + 2*j] += u[fi+j];           // update finer grid

```

```

252 }
253 // -----
254 // --> Solve: direct solver for the Helmholtz equation *at the coarsest possible
255 // grid*.
256 // -----
257 proc Solve(
258   const in ig,           // the deepest level, coarsest grid,
259   ref u: [] real,        // the solutions
260   ref f: [] real         // the forcings
261 ) where (u.rank == 2 && f.rank == 1) {
262   const nn = length[ig]; // hope so!
263   // -----
264   // is this the right grid? there may be a lot of redundancy in the asserts
265   // below
266   // -----
267   assert (ig == G && nn == 2);
268   // -----
269   // synonyms for local things
270   // -----
271   const ref fi = first[ig];
272   const ref la = last[ig];
273   // -----
274   // are the BCs on f homogeneous?
275   // -----
276   assert (f[fi] == 0.0 && f[la] == 0.0);
277   // -----
278   // we need local h and den to match the coarser resolution
279   // -----
280   var h = L/nn;
281   var den = (2.0 + h**2 * k**2);
282   u[inew, fi+1] = - h**2 * f[fi+1]/den;
283 }
```

4

Solução numérica de equações diferenciais parciais

Dada uma função ϕ de duas variáveis (digamos, x e t) e uma equação diferencial parcial para a mesma, é possível encontrar representações aproximadas de ϕ na forma $\phi(x_i, t_n)$ 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 $\phi(x_i, t_n)$. À medida que o algoritmo da solução progide, mais e mais pontos são obtidos.

4.1 – Advecção pura: a onda cinemática

Considere a equação

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

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

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

Seja então o problema

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

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

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

Vamos adotar a notação

$$\phi_i^n \equiv \phi(x_i, t_n), \quad (4.5)$$

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

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

com

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

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

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

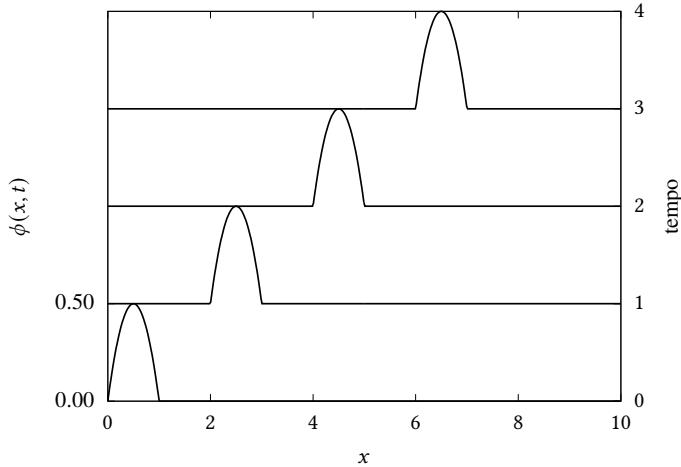


Figure 4.1: Solução analítica das equações 4.3–4.4.

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

$$\frac{\partial \phi}{\partial t} \Big|_{i,n} = \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + \mathcal{O}(\Delta t), \quad (4.10)$$

$$\frac{\partial \phi}{\partial x} \Big|_{i,n} = \frac{\phi_{i+1}^n - \phi_{i-1}^n}{2\Delta x} + \mathcal{O}(\Delta x^2). \quad (4.11)$$

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

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

(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 4.1. Por motivos que ficarão mais claros na sequência, nós escolhemos $\Delta x = 0,01$, e $\Delta t = 0,0005$.

O programa gera um arquivo de saída binário, que por sua vez é lido pelo próximo programa na sequência, `surf1d_ins.py`, mostrado na listagem 4.2. 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 Δt . 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 4.2. Repare como a solução se torna rapidamente instável. Repare também como a solução numérica, em $t = 750\Delta t = 0,375$, ainda está bastante distante dos tempos mostrados na solução analítica da figura 4.1 (que vão até $t = 4$). Claramente, o esquema explícito que nós programamos jamais nos levará a uma solução numérica satisfatória para tempos $t \sim 1$.

Por que o esquema utilizado em (4.12) fracassa? Uma forma de obter a resposta é fazer uma *análise de estabilidade de von Neumann*. A análise de estabilidade de von Neumann consiste primeiramente em observar que, em um computador real, (4.12) jamais será calculada com precisão infinita. O que o computador realmente calcula é um valor *arredondado* $\tilde{\phi}_i^n$. Por enquanto, nós só vamos fazer essa

Listing 4.1: 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", serializer = new binarySerializer());
8 const dx = 0.01;
9 const dt = 0.0005;
10 writef("// dx=%9.4dr\n",dx);
11 writef("// dy=%9.4dr\n",dt);
12 const nx = round(10.0/dx):int;           // número de pontos em x
13 const nt = round(1.0/dt):int;           // número de pontos em t
14 writef("// nx=%9i\n",nx);
15 writef("// nt=%9i\n",nt);
16 var phi: [0..1,0..nx] real;            // apenas 2 posições no tempo
17                                // são necessárias!
18 for i in 0..nx do {                  // monta a condição inicial
19     var xi = i*dx;
20     phi[0,i] = CI(xi);
21 }
22 fou.write(phi[0,0..nx]);              // imprime a condição inicial
23 var iold = 0;
24 var inew = 1;
25 const c = 2.0;                      // celeridade da onda
26 const couhalf = c*dt/(2.0*dx);      // metade do número de Courant
27 for n in 1..nt do {                // loop no tempo
28     writeln("n=",n);
29     for i in 1..nx-1 do {          // loop no espaço
30         phi[inew,i] = phi[iold,i] - couhalf*(phi[iold,i+1] - phi[iold,i-1]);
31     }
32     phi[inew,0] = 0.0;
33     phi[inew,nx] = 0.0;
34     fou.write(phi[inew,0..nx]);    // imprime uma linha com os novos dados
35     iold <=> inew;              // troca os índices
36 }
37 fou.close();
38 proc CI(const in x: real): real {    // define a condição inicial
39     if 0 <= x && x <= 1.0 then {
40         return 2.0*x*(1.0-x);
41     }
42     else {
43         return 0.0;
44     }
45 }
```

Listing 4.2: `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());
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 const founam = "surf1d_ins.dat";
30 writeln(founam);
31 const fou = openWriter(founam,
32                         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 fou.close();

```

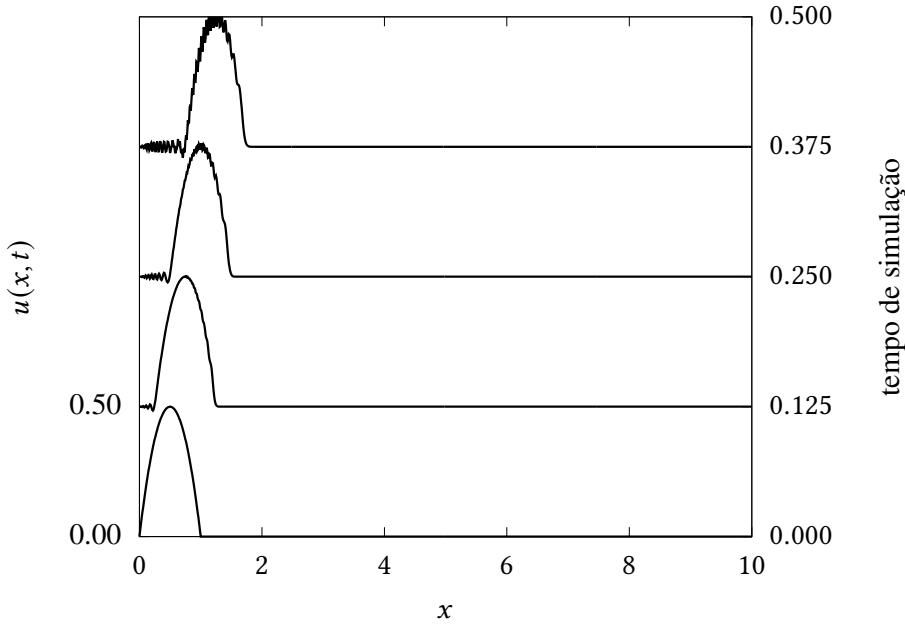


Figure 4.2: Solução numérica produzida por `onda1d-ins.chpl`, para $t = 250\Delta t$, $500\Delta t$ e $750\Delta t$.

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

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

Note que (4.12) se aplica tanto para ϕ 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 ϵ_i^n :

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

onde

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

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

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

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

onde e é a base dos logaritmos naturais, $i = \sqrt{-1}$, $N = L/\Delta x$ é o número de pontos da discretização em x , e L é o tamanho do domínio em x .

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

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

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

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

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

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

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

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

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

Note que $\beta \neq 0$, donde $e^{\alpha\Delta t} > 1$ via (4.19), e o esquema de diferenças finitas é *incondicionalmente instável*.

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

$$\phi_i^{n+1} = \frac{1}{2} [(\phi_{i+1}^n + \phi_{i-1}^n) - \text{Co}(\phi_{i+1}^n - \phi_{i-1}^n)]. \quad (4.21)$$

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

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

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

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

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

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

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

Não custa repetir: (4.24) é *idêntica* a (4.21). Porém, comparando-a com (4.12) (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:

$$\begin{aligned}\phi_{i+1} + \phi_{i-1} &= 2\phi_i + \frac{d^2\phi}{dx^2} \Big|_i \Delta x^2 + \mathcal{O}(\Delta x^4), \\ \frac{d^2\phi}{dx^2} \Big|_i &= \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} + \mathcal{O}(\Delta x^2).\end{aligned}\tag{4.25}$$

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

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

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 (4.21).

Mesmo assim, vamos programá-lo! O programa `onda1d_lax.chpl` está mostrado na listagem 4.3. 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 4.4. O único trabalho desse programa é selecionar algumas “linhas” da saída de `onda1d_lax.chpl`; no caso, nós o rodamos com o comando

Listing 4.3: 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("//dt=%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 }
```

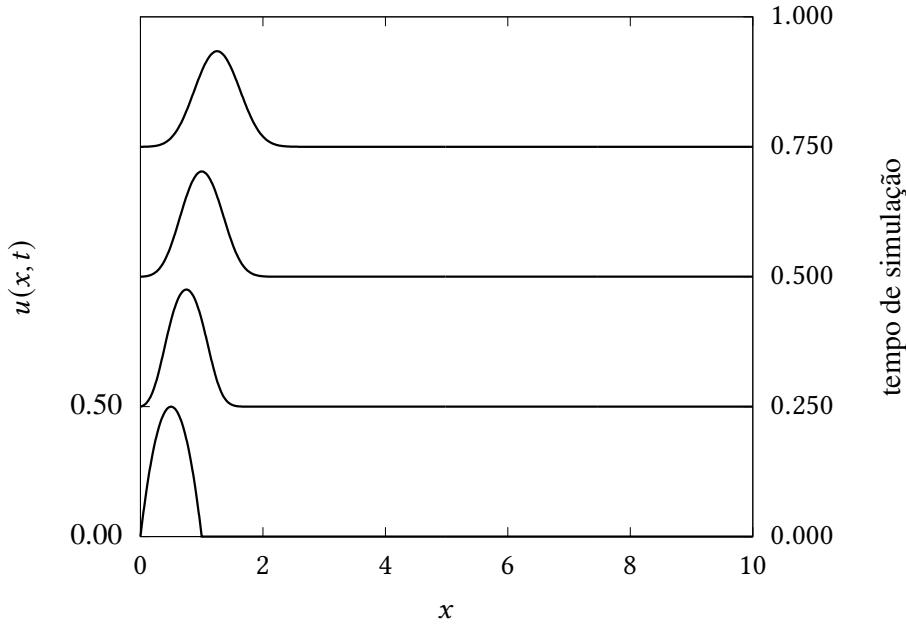


Figure 4.3: Solução numérica produzida por `onda1d_lax.chpl`, para $t = 500\Delta t$, $1000\Delta t$ e $1500\Delta t$.

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

o que significa selecionar 3 saídas (além da condição inicial), de 500 em 500 intervalos de tempo Δt . 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 4.3. 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 (4.1) é o esquema de diferenças regressivas; nesse esquema, chamado de esquema *upwind* — literalmente, “corrente acima” na literatura de língua inglesa — a discretização utilizada é

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= -c \frac{\phi_i^n - \phi_{i-1}^n}{\Delta x}, \\ \phi_i^{n+1} &= \phi_i^n - \text{Co} [\phi_i^n - \phi_{i-1}^n]. \end{aligned} \quad (4.26)$$

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:

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

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

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

Listing 4.4: `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> latoantes:
24     for ir in 1..n do {              // l <n> vezes, s guarda a ltima
25         fin.read(u);
26         v[it,0..nx] = u;            // guarda a ltima
27     }
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();

```

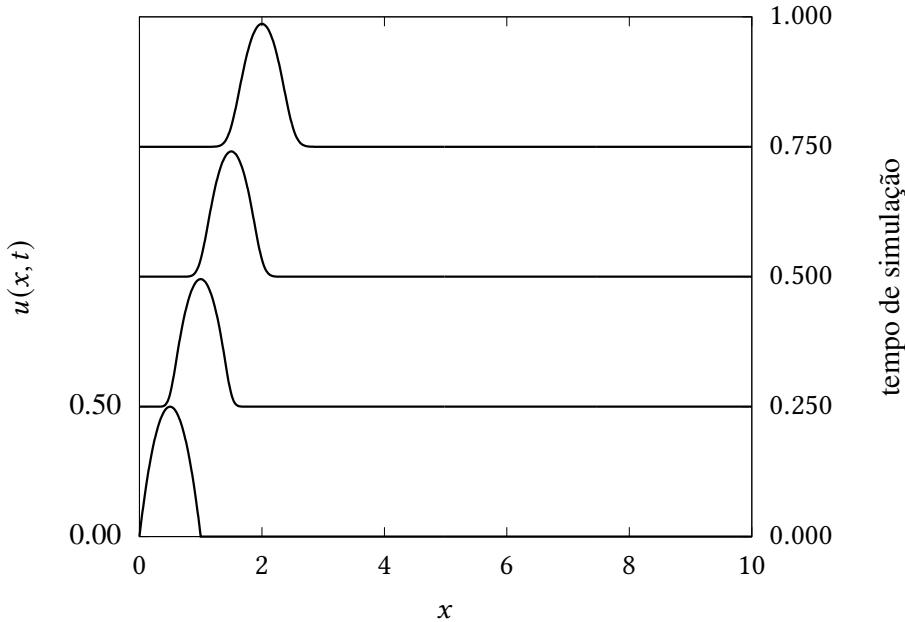


Figure 4.4: Solução numérica produzida pelo esquema *upwind*, para $t = 250, 500$ e 750 .

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 (4.23), mas em um outro esquema de diferenças finitas. A lição não deve ser mal interpretada: longe de supor que (4.23) vale sempre, é a análise de estabilidade que deve 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 Δt e de Δx de antes. As mudanças necessárias nos códigos computacionais são óbvias, e são deixadas a cargo do(a) leitor(a).

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

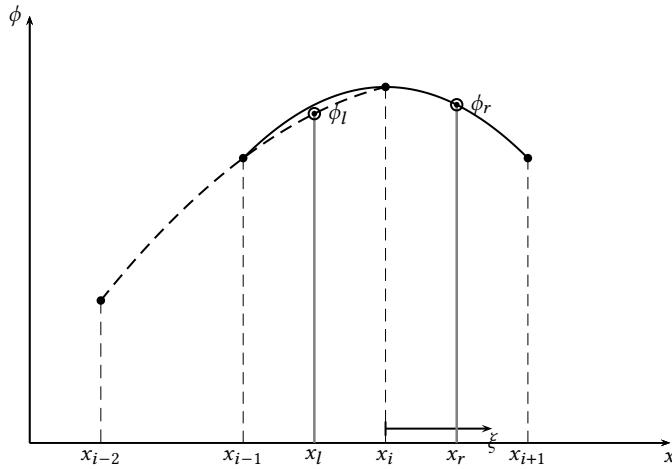


Figure 4.5: Quick's interpolation scheme.

Quick Leonard (1979) 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 4.5. For definitiveness, consider the parabola through (x_{i-1}, ϕ_{i-1}) , (x_i, ϕ_i) , and (x_{i+1}, ϕ_{i+1}) , and set a local axis ξ centered at x_i such that $\xi = x - x_i$. The equation for the parabola is then

$$\phi = a_0 + a_1\xi + a_2\xi^2, \quad (4.28)$$

such that

$$\phi_i = a_0, \quad (4.29)$$

$$\phi_{i-1} = a_0 - a_1\Delta x + a_2\Delta x^2, \quad (4.30)$$

$$\phi_{i+1} = a_0 + a_1\Delta x + a_2\Delta x^2. \quad (4.31)$$

The values of a_1 and a_2 can easily be obtained by subtracting and summing (4.30) and (4.31):

$$\begin{aligned} \phi_{i+1} - \phi_{i-1} &= 2a_1\Delta x, \\ a_1 &= \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x}; \end{aligned} \quad (4.32)$$

and

$$\begin{aligned} \phi_{i+1} + \phi_{i-1} &= 2a_0 + 2a_2\Delta x^2 \\ &= 2\phi_i + 2a_2\Delta x^2; \end{aligned} \quad (4.33)$$

$$a_2 = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{2\Delta x^2}. \quad (4.34)$$

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

$$\begin{aligned} \phi_r &= \phi_i + \frac{(\phi_{i+1} - \phi_{i-1})}{2\Delta x} \frac{\Delta x}{2} + \frac{(\phi_{i+1} - 2\phi_i + \phi_{i-1})}{2\Delta x^2} \frac{\Delta x^2}{4} \\ &= \phi_i + \frac{1}{4}(\phi_{i+1} - \phi_{i-1}) + \frac{1}{8}(\phi_{i+1} - 2\phi_i + \phi_{i-1}) \\ &= \frac{1}{2}(\phi_i + \phi_{i+1}) - \frac{1}{8}(\phi_{i+1} - 2\phi_i + \phi_{i-1}) \\ &= \frac{1}{8}(3\phi_{i+1} + 6\phi_i - \phi_{i-1}) \end{aligned} \quad (4.35)$$

Listing 4.5: quick_grid.chpl — A stable grid for QUICK’s solution of (4.1)

```

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);

```

By the same token we have, for ϕ_l ,

$$\begin{aligned}\phi_l &= \frac{1}{2}(\phi_{i-1} + \phi_i) - \frac{1}{8}(\phi_i - 2\phi_{i-1} + \phi_{i-2}) \\ &= \frac{1}{8}(3\phi_i + 6\phi_{i-1} - \phi_{i-2})\end{aligned}\quad (4.36)$$

Discretization of (4.1) 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

$$\phi_i^{n+1} = \phi_i^n - \frac{Co}{8} (\phi_{i-2}^n - 7\phi_{i-1}^n + 3\phi_i^n + 3\phi_{i+1}^n). \quad (4.37)$$

Although QUICK is a very popular scheme in computational fluid dynamics, for the case of the pure kinematic wave equation (4.1) it turns out to be *almost* unconditionally unstable! This is clearly discussed in [Chen e Falconer \(1992\)](#); their Eq. (18) for the simple case of (4.1) gives

$$Co \leq \frac{1}{2} \left(\frac{\pi}{N} \right)^2, \quad (4.38)$$

where n is the number of grid points. Consider again listing 4.3; 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 4.6.

In order to “see” the results, we have surf1d_quick.chpl in listing 4.7, 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 4.6. 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.

Listing 4.6: 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 }
```

Listing 4.7: `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                         deserializer = 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 condição 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();

```

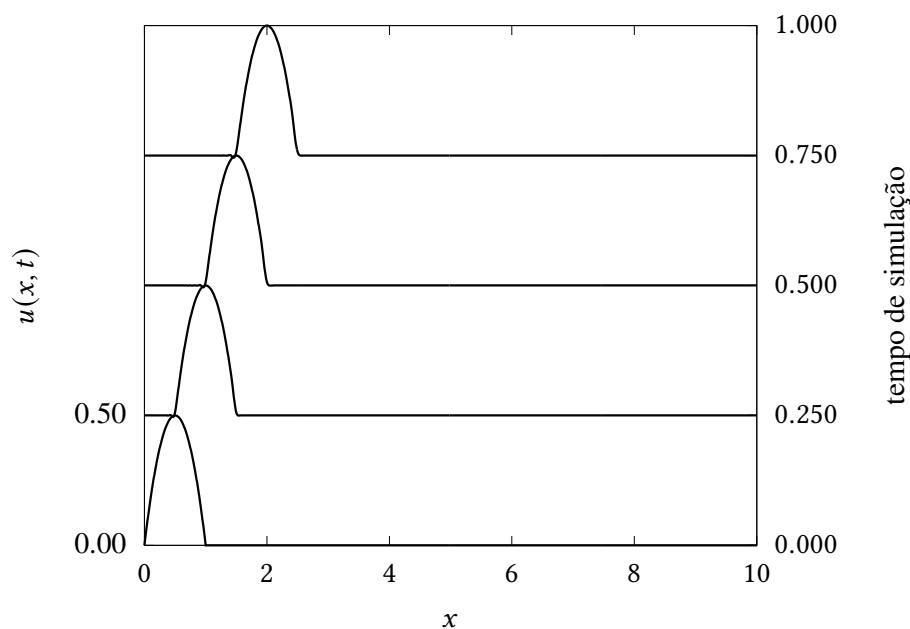


Figure 4.6: Solução numérica produzida pelo esquema QUICK, para $t = 250, 500$ e 750 .

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 (4.1) in a Taylor series up to order 3,

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

the mean value of ϕ inside the control volume is

$$\begin{aligned}\bar{\phi}(t) &= \frac{1}{\Delta x} \int_{\xi=-\Delta x/2}^{\xi=+\Delta x/2} \left[\phi(0, t) + \frac{\partial\phi(0, t)}{\partial\xi}\xi + \frac{1}{2}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\xi^2 + \frac{1}{6}\frac{\partial^3\phi(0, t)}{\partial\xi^3}\xi^3 + \mathcal{O}(\xi^4) \right] d\xi \\ &= \frac{1}{\Delta x} \left[\phi(0, t)\xi + \frac{1}{2}\frac{\partial\phi(0, t)}{\partial\xi}\xi^2 + \frac{1}{3} \times \frac{1}{2}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\xi^3 + \frac{1}{4} \times \frac{1}{6}\frac{\partial^3\phi(0, t)}{\partial\xi^3}\xi^4 + \mathcal{O}(\xi^5) \right]_{-\Delta x/2}^{+\Delta x/2} \\ &= \frac{1}{\Delta x} \left[\phi(0, t)\Delta x + \frac{1}{3}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\frac{\Delta x^3}{8} + \mathcal{O}(\Delta x^5) \right] \\ &= \phi(0, t) + \frac{1}{24}\frac{\partial^2\phi(0, t)}{\partial\xi^2}\Delta x^2 + \mathcal{O}(\Delta x^4).\end{aligned}\tag{4.39}$$

The QUICKEST method, also proposed by [Leonard \(1979\)](#), is a finite-volume version of QUICK which can be derived from (4.39) [Chen e Falconer \(1992\)](#); [Nishikawa \(2021\)](#). To see this, write the control-volume balance equivalent to (4.1) as

$$\begin{aligned}\frac{\partial}{\partial t} \int_{-\Delta x/2}^{+\Delta x/2} \phi(\xi, t) d\xi + [-c\phi(-\Delta x/2, t) + c\phi(+\Delta x/2, t)] &= 0, \\ \frac{\partial[\Delta x \bar{\phi}]}{\partial t} + c [\phi(+\Delta x/2, t) - \phi(-\Delta x/2, t)] &= 0, \\ \frac{d\bar{\phi}}{dt} + c \frac{[\phi(+\Delta x/2, t) - \phi(-\Delta x/2, t)]}{\Delta x} &= 0.\end{aligned}\tag{4.40}$$

But now we need to differentiate (4.39)! 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 (4.35)–(4.36) for $\phi(+\Delta x/2, t)$ and $\phi(-\Delta x/2, t)$ in (4.40), 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:

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

or

$$\phi_i^{n+1} = \phi_i^n - \frac{Co}{6} (\phi_{i-2}^n - 6\phi_{i-1}^n + 3\phi_i^n + 2\phi_{i+1}^n), \quad (4.41)$$

which is the QUICKEST scheme for pure advection.

Exemplo 4.1 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, $Co = u\Delta t/\Delta x$, e de Fourier, $Fo = D\Delta t/\Delta x^2$, da forma

$$(a + bFo)^2 + (c + dCo)^2 \leq 1;$$

que garanta a estabilidade do esquema.

SOLUÇÃO

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

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

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

$$\begin{aligned}
|e^{a\Delta t}|^2 &< 1, \\
(1 - 4\text{Fo} \sin^2(\theta))^2 + (2\text{Co} \sin(\theta) \cos(\theta))^2 &\leq 1, \\
1 - 8\text{Fo} \sin^2(\theta) + 16\text{Fo}^2 \sin^4(\theta) + 4\text{Co}^2 \sin^2(\theta) \cos^2(\theta) &\leq 1, \\
-8\text{Fo} \sin^2(\theta) + 16\text{Fo}^2 \sin^4(\theta) + 4\text{Co}^2 \sin^2(\theta) \cos^2(\theta) &\leq 0.
\end{aligned} \tag{4.42}$$

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

$$\begin{aligned}
-8\text{Fo} + 16 \sin^2(\theta) \text{Fo}^2 + 4 \cos^2(\theta) \text{Co}^2 &\leq 0, \\
-2\text{Fo} + 4 \sin^2(\theta) \text{Fo}^2 + \cos^2(\theta) \text{Co}^2 &\leq 0,
\end{aligned} \tag{4.43}$$

onde

$$\text{Co} \leq \left[\frac{2\text{Fo} (1 - 2 \sin^2(\theta) \text{Fo})}{\cos^2(\theta)} \right]^{1/2}. \tag{4.44}$$

Para cada θ , (4.44) define uma região *diferente* do plano $\text{Fo} \times \text{Co}$. É preciso portanto *varrer* os valores de θ e encontrar a região do plano em que o esquema é estável independentemente de θ (lembre-se de que θ 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 π (figura 4.7). No entanto, a função definida em (4.44) é simétrica em relação a $\theta = \pi/2$, como mostra a figura 4.8 para diversos valores de Fo . Portanto, basta “varrer” θ desde 0 até $\pi/2$.

Antes de fazer essa varredura com “força bruta”, entretanto, vale a pena entender as curvas representadas por 4.43 tanto em geral, quanto para os casos particulares $\theta = 0$ (entendido como um limite, já que (4.43) 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 Fo e y para Co . No limite da igualdade, manipulamos agora (4.42) da seguinte forma:

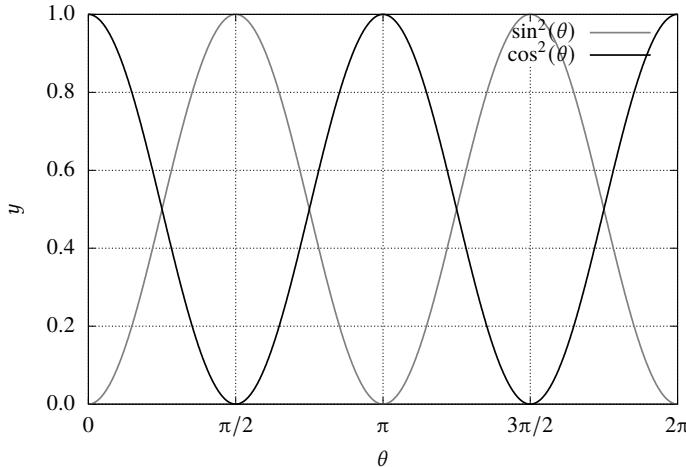
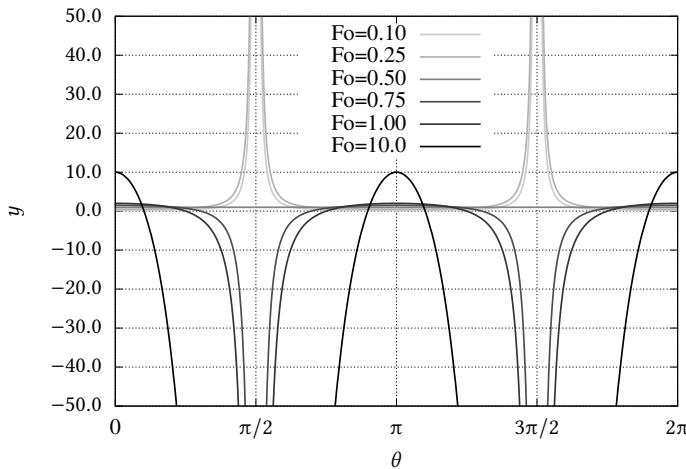
$$\begin{aligned}
(1 - 4 \sin^2(\theta)x)^2 + (\sin(2\theta)y)^2 &= 1, \\
(4 \sin^2(\theta))^2 \left(\frac{1}{4 \sin^2(\theta)} - x \right)^2 + (\sin(2\theta)y)^2 &= 1, \\
(4 \sin^2(\theta))^2 \left(x - \frac{1}{4 \sin^2(\theta)} \right)^2 + (\sin(2\theta)y)^2 &= 1, \\
\frac{\left(x - \frac{1}{4 \sin^2(\theta)}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

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 θ 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 $\theta = 0$, (4.43) simplifica-se para

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

Figure 4.7: As funções $\sin^2(\theta)$ e $\cos^2(\theta)$ Figure 4.8: A função definida em 4.44 (para $Fo = 1$)

$$Co \leq \sqrt{2Fo}. \quad (4.45)$$

Esta é uma região não mais sob uma elipse, mas sim sob uma parábola. Para $\theta = \pi/2$, (4.43) simplifica-se para

$$\begin{aligned} -2Fo + 4Fo^2 &\leq 0, \\ -Fo + 2Fo^2 &\leq 0, \\ Fo(-1 + 2Fo) &\leq 0, \\ Fo &\leq \frac{1}{2}. \end{aligned} \quad (4.46)$$

Voltamos agora para uma varredura sistemática: a figura 4.9 mostra as meias-elipses que definem a região de estabilidade para cada θ , em incrementos $\Delta\theta = \pi/32$ a partir de $\theta = \pi/32$, até $\theta = 15\pi/32$, utilizando (4.44). 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 4.9, representada por

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

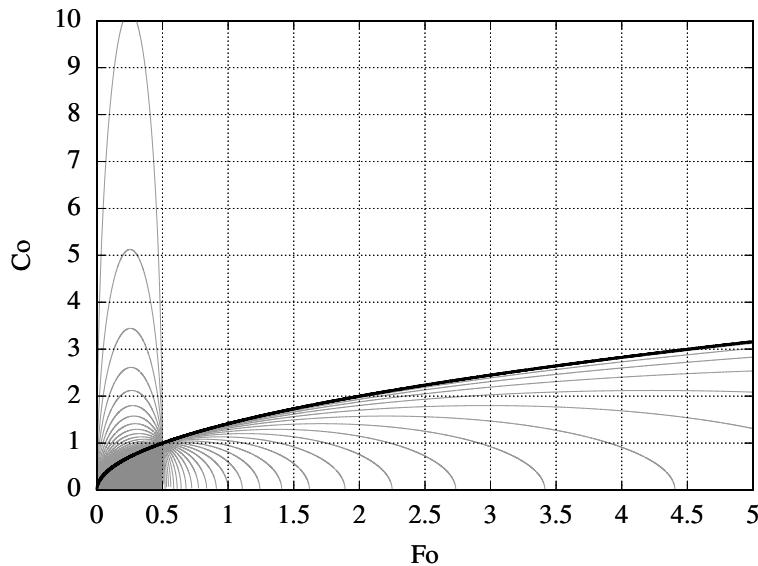


Figure 4.9: Exemplo 4.1: interseção (em cinza escuro) de 32 regiões (em cinza-claro; incrementos de $\pi/32$) na equação (4.44) A parábola em preto representa o caso degenerado $\theta = 0$.

Exercícios Propostos

4.1 Escreva o programa `onda1d-upw` e `surf1d-upw`, que implementam o esquema *upwind*. Reproduza a figura 4.4.

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

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

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.

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

4.4 Dado esquema explícito

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

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

4.5 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 $\text{Co} = (c\Delta t)/\Delta x$. 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?

4.6 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 $O(\Delta x^2)$.

4.7 Utilizando o esquema *upwind* do Exercício 4.6, 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.

4.2 – Difusão pura

Considere agora a equação da difusão,

$$\frac{\partial \phi}{\partial t} = D \frac{\partial^2 \phi}{\partial x^2}, \quad (4.47)$$

com condições iniciais e de contorno

$$\phi(x, 0) = f(x) \quad (4.48)$$

$$\phi(0, t) = \phi(L, t) = 0. \quad (4.49)$$

A solução analítica é

$$\phi(x, t) = \sum_{n=1}^{\infty} A_n e^{-\frac{n^2 \pi^2 D}{L^2} t} \sin \frac{n \pi x}{L}, \quad (4.50)$$

$$A_n = \frac{2}{L} \int_0^L f(x) \sin \frac{n \pi x}{L} dx. \quad (4.51)$$

Em particular, se

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

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

$$\begin{aligned} A_{2n+1} &= \frac{16}{\pi^3 (2n+1)^3}, \\ \phi(x, t) &= \sum_{n=0}^{\infty} \frac{16}{\pi^3 (2n+1)^3} e^{-((2n+1)^2 \pi^2 D)t} \sin((2n+1)\pi x) \end{aligned} \quad (4.52)$$

O programa `difusao1d-ana.chpl`, mostrado na listagem 4.8, 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 4.9, seleciona alguns instantes de tempo da solução analítica para visualização:

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

A figura 4.10 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.

Listing 4.8: difusa01d-ana.chpl — Solução analítica da equação da difusão

```

1 // -----
2 // difusa01d-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: ./difusa01d-ana
11 // -----
12 use IO only openWriter, binarySerializer;
13 use Math only exp,pi,sin;
14 const fou = openWriter("difusa01d-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 }
```

Listing 4.9: `divisaold-ana.py` — Seleciona alguns instantes de tempo da solução analítica para visualização

```

1 // -----
2 // divisaold-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();

```

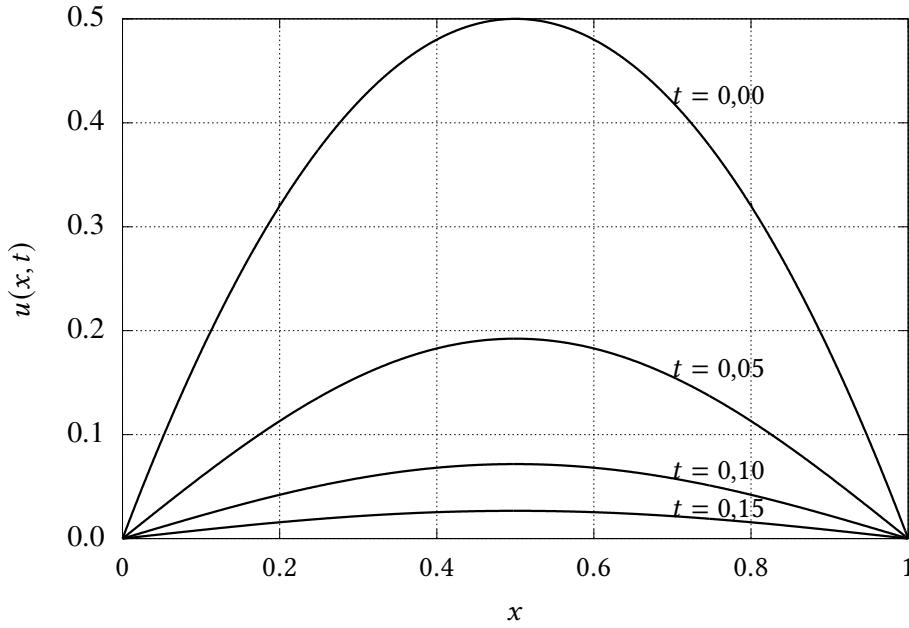


Figure 4.10: Solução analítica da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$.

Esquema explícito Talvez o esquema explícito mais óbvio para discretizar (4.47) seja

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} = D \frac{\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{\Delta x^2}. \quad (4.53)$$

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 (4.25), 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 (4.53) é *estável*?

Explicitamos ϕ_i^{n+1} em (4.53):

$$\phi_i^{n+1} = \phi_i^n + \text{Fo} [\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n], \quad (4.54)$$

onde

$$\text{Fo} = \frac{D \Delta t}{\Delta x^2} \quad (4.55)$$

é o *número de Fourier de grade* (Jaluria e Torrance, 1986, capítulo 3). A análise de estabilidade de von Neumann agora produz

$$\begin{aligned} \xi_l e^{a(t_n + \Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} + \\ &\quad \text{Fo} [\xi_l e^{at_n} e^{ik_l (i+1) \Delta x} - 2\xi_l e^{at_n} e^{ik_l i \Delta x} + \xi_l e^{at_n} e^{ik_l (i-1) \Delta x}], \\ e^{a \Delta t} &= 1 + \text{Fo} [e^{+ik_l \Delta x} - 2 + e^{-ik_l \Delta x}] \\ &= 1 + 2\text{Fo} [\cos(k_l \Delta x) - 1] \\ &= 1 - 4\text{Fo} \sin^2\left(\frac{k_l \Delta x}{2}\right) \end{aligned} \quad (4.56)$$

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

$$\begin{aligned} -8\text{Fo} \operatorname{sen}^2\left(\frac{k_l \Delta x}{2}\right) + 16\text{Fo}^2 \operatorname{sen}^4\left(\frac{k_l \Delta x}{2}\right) &\leq 0, \\ 8\text{Fo} \operatorname{sen}^2\left(\frac{k_l \Delta x}{2}\right) \left[-1 + 2\text{Fo} \operatorname{sen}^2\left(\frac{k_l \Delta x}{2}\right)\right] &\leq 0, \\ \text{Fo} &\leq \frac{1}{2}. \end{aligned} \quad (4.57)$$

Podemos agora calcular o número de Fourier que utilizamos para plotar a solução analítica (verifique nas listagens 4.8 e 4.9):

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

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

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

Nós esperamos que nosso esquema explícito agora rode muito lentamente. Mas vamos implementá-lo. O programa que implementa o esquema é o `difusao1d-exp.chpl`, mostrado na listagem 4.10.

O programa `divisao1d-exp.chpl`, mostrado na listagem 4.11, 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 4.11: ele é impressionantemente bom, embora seja computacionalmente muito caro. A escolha judiciosa de Δt e Δx para obedecer ao critério (4.57) 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.

Listing 4.10: difusa01d-exp.py — Solução numérica da equação da difusão: método explícito.

```

1 // -----
2 // difusa01d-exp resolve uma equação de difusão com um método explícito
3 //
4 // uso: ./difusa01d-exp
5 //
6 use IO only openWriter, binarySerializer;
7 const fou = openWriter("difusa01d-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                                         // 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 D = 2.0;                         // difusividade
28 const Fon = D*dt/((dx)**2);             // número de Fourier
29 writef("Fo=%10.6dr\n",Fon);
30 for n in 1..nt do {                    // loop no tempo a partir de 1
31     writeln(n);
32     for i in 1..nx-1 do {               // loop no espaço
33         phi[inew,i] = phi[iold,i] +
34             Fon*(phi[iold,i+1] - 2*phi[iold,i] + phi[iold,i-1]);
35     }
36     phi[inew,0] = 0.0;                  // condição de contorno, x = 0
37     phi[inew,nx] = 0.0;                 // condição de contorno, x = 1
38     fou.write(phi[inew,0..nx]);          // imprime uma linha com os novos dados
39     iold <=> inew;                   // troca os índices
40 }
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 }
```

Listing 4.11: `divisao1d-exp.chpl` — Seleciona alguns instantes de tempo da solução analítica para visualização

```

1 // -----
2 // divisao1d-exp.py: imprime em <arq> <m>+1 saídas de divisao1d-exp a cada <n>
3 // intervalos de tempo
4 //
5 // uso: ./divisao1d-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("divisao1d-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 = "divisao1d-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();

```

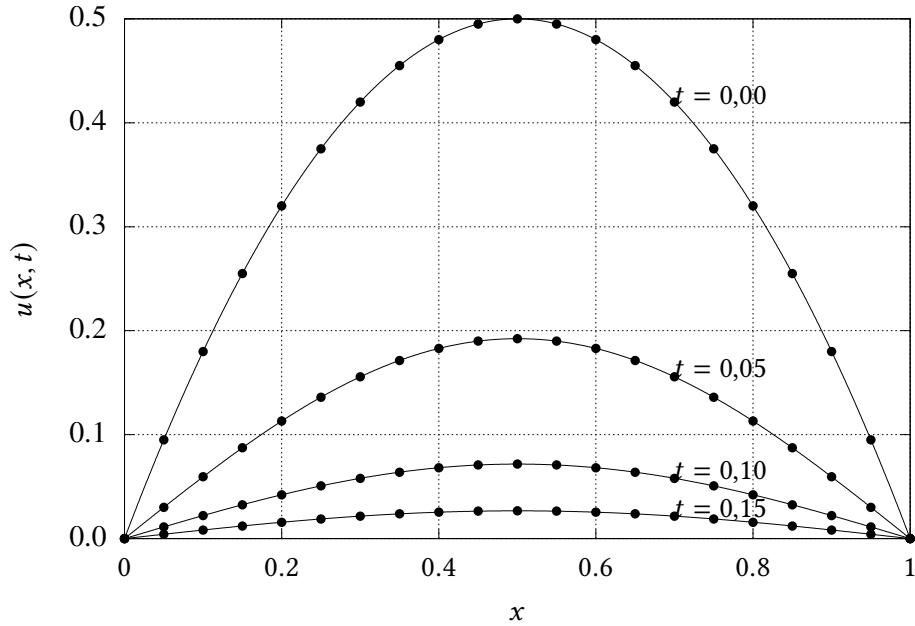


Figure 4.11: Solução numérica com o método explícito (4.54) (círculos) versus a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

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 $n + 1$:

$$\begin{aligned} \frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= D \frac{\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}}{\Delta x^2}, \\ \phi_i^{n+1} - \phi_i^n &= \text{Fo}(\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}), \\ -\text{Fo}\phi_{i-1}^{n+1} + (1 + 2\text{Fo})\phi_i^{n+1} - \text{Fo}\phi_{i+1}^{n+1} &= \phi_i^n. \end{aligned} \quad (4.58)$$

Reveja a discretização (4.5)–(4.9): para $i = 1, \dots, N_x - 1$, (4.58) acopla 3 valores das incógnitas ϕ^{n+1} no instante $n + 1$. Quando $i = 0$, e quando $i = N_x$, não podemos utilizar (4.58), porque não existem os índices $i = -1$, e $i = N_x + 1$. Quando $i = 1$ e $i = N_x - 1$, (4.58) 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

$$(1 + 2\text{Fo})\phi_1^{n+1} - \text{Fo}\phi_2^{n+1} = \phi_1^n, \quad (4.59)$$

$$-\text{Fo}\phi_{N_x-2}^{n+1} + (1 + 2\text{Fo})\phi_{N_x-1}^{n+1} = \phi_{N_x-1}^n. \quad (4.60)$$

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

$$\begin{bmatrix} 1 + 2\text{Fo} & -\text{Fo} & 0 & \dots & 0 & 0 \\ -\text{Fo} & 1 + 2\text{Fo} & -\text{Fo} & 0 & \dots & 0 \\ \vdots & & & & & \vdots \\ 0 & \dots & 0 & -\text{Fo} & 1 + 2\text{Fo} & -\text{Fo} \\ 0 & 0 & \dots & 0 & -\text{Fo} & 1 + 2\text{Fo} \end{bmatrix} \begin{bmatrix} \phi_1^{n+1} \\ \phi_2^{n+1} \\ \vdots \\ \phi_{N_x-2}^{n+1} \\ \phi_{N_x-1}^{n+1} \end{bmatrix} = \begin{bmatrix} \phi_1^n \\ \phi_2^n \\ \vdots \\ \phi_{N_x-2}^n \\ \phi_{N_x-1}^n \end{bmatrix}. \quad (4.61)$$

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

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

$$\begin{aligned}
\xi_l e^{a(t_n + \Delta t)} e^{ik_l i \Delta x} &= \xi_l e^{at_n} e^{ik_l i \Delta x} \\
&\quad + \text{Fo} \left(\xi_l e^{a(t_n + \Delta t)} e^{ik_l(i+1) \Delta x} - 2 \xi_l e^{a(t_n + \Delta t)} e^{ik_l i \Delta x} \right. \\
&\quad \left. + \xi_l e^{a(t_n + \Delta t)} e^{ik_l(i-1) \Delta x} \right), \\
e^{a \Delta t} &= 1 + e^{a \Delta t} \text{Fo} \left(e^{ik_l \Delta x} - 2 + e^{-ik_l \Delta x} \right), \\
e^{a \Delta t} &= 1 + e^{a \Delta t} 2 \text{Fo} (\cos(k_l \Delta x) - 1), \\
e^{a \Delta t} &= 1 - e^{a \Delta t} 4 \text{Fo} \operatorname{sen}^2 \left(\frac{k_l \Delta x}{2} \right), \\
e^{a \Delta t} \left[1 + 4 \text{Fo} \operatorname{sen}^2 \left(\frac{k_l \Delta x}{2} \right) \right] &= 1, \\
|e^{a \Delta t}| &= \frac{1}{1 + 4 \text{Fo} \operatorname{sen}^2 \left(\frac{k_l \Delta x}{2} \right)} \leq 1 \quad \text{sempre.}
\end{aligned} \tag{4.62}$$

Portanto, o esquema implícito (4.58) é incondicionalmente estável, e temos confiança de que o programa correspondente não se instabilizará.

Existem várias coisas atraentes para um programador em (4.61). 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: Press et al., 1992, seção 2.4, subrotina `tridag`). Em segundo lugar, a matriz do sistema é *constante*: ela só precisa ser montada uma vez no programa, o que torna a solução numérica potencialmente muito rápida. 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 4.12. 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]` recebem 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 4.12

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 (4.47) é aproximada, no esquema implícito (4.58), 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 (4.58) por

$$\begin{aligned}
\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} &= \frac{D}{2} \left[\frac{\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{\Delta x^2} + \frac{\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}}{\Delta x^2} \right], \\
\phi_i^{n+1} &= \phi_i^n + \frac{\text{Fo}}{2} [\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n + \phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1}].
\end{aligned} \tag{4.63}$$

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

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

Listing 4.12: 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     }

```

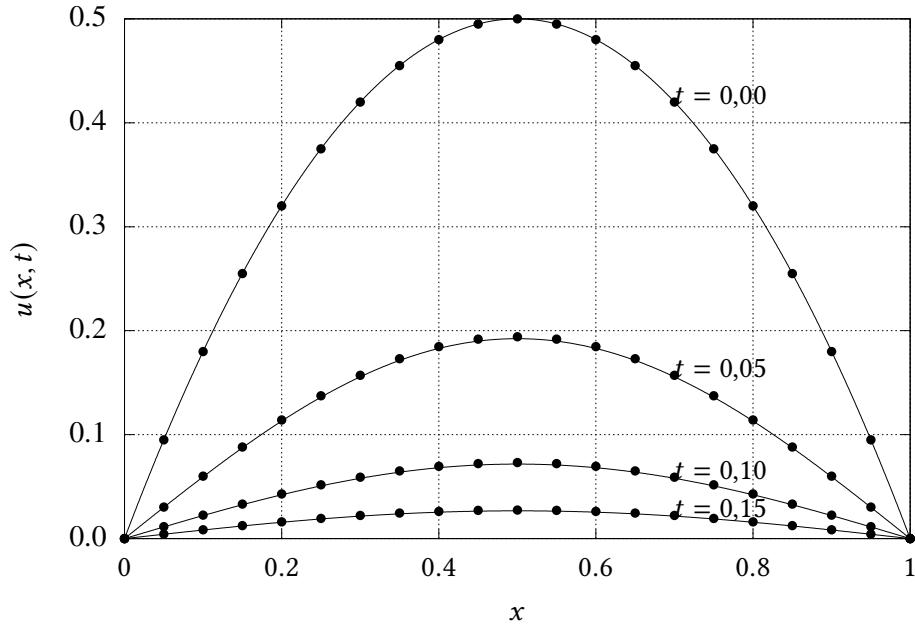


Figure 4.12: Solução numérica com o método implícito (4.58) (círculos) *versus* a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

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 (4.58):

$$-\frac{\text{Fo}}{2} u_{i-1}^{n+1} + (1 + \text{Fo}) u_i^{n+1} - \frac{\text{Fo}}{2} u_{i+1}^{n+1} = \frac{\text{Fo}}{2} u_{i-1}^n + (1 - \text{Fo}) u_i^n + \frac{\text{Fo}}{2} u_{i+1}^n \quad (4.64)$$

Para as condições de contorno de (4.49), as linhas correspondentes a $i = 1$ e $i = N_x - 1$ são

$$(1 + \text{Fo}) u_1^{n+1} - \frac{\text{Fo}}{2} u_2^{n+1} = (1 - \text{Fo}) u_1^n + \frac{\text{Fo}}{2} u_2^n, \quad (4.65)$$

$$-\frac{\text{Fo}}{2} u_{N_x-2}^{n+1} + (1 + \text{Fo}) u_{N_x-1}^{n+1} = \frac{\text{Fo}}{2} u_{N_x-2}^n + (1 - \text{Fo}) u_{N_x-1}^n \quad (4.66)$$

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 4.13.

A grande novidade computacional de `difusao1d-ckn.py` é a linha 52: é possível escrever (4.64) *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 (4.58); 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 4.13.

Listing 4.13: `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=%i\n",nx);
16 writef("//nt=%i\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("Fon=%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

```

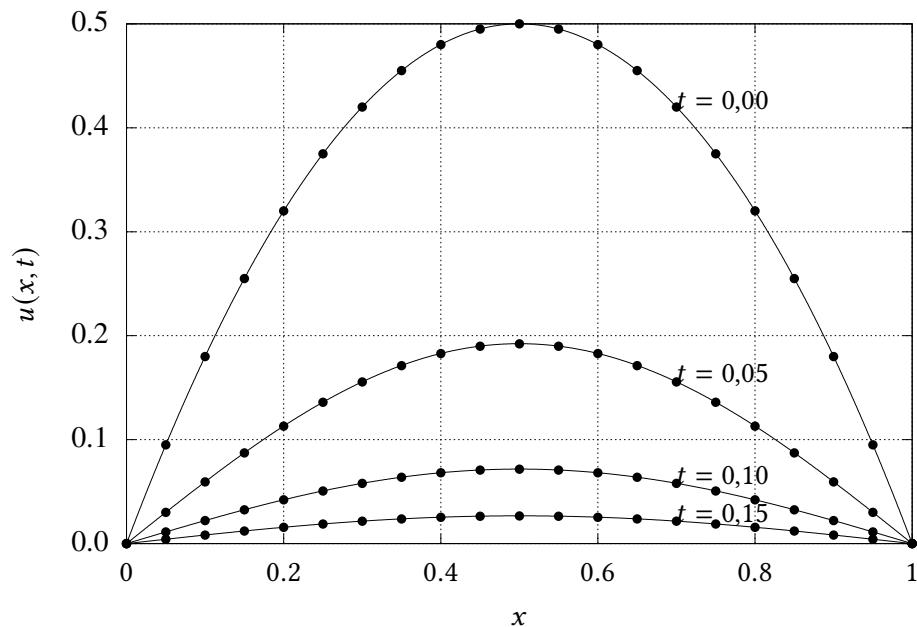


Figure 4.13: Solução numérica com o método de Crank-Nicholson ((4.64)) (círculos) versus a solução analítica (linha cheia) da equação de difusão para $t = 0$, $t = 0,05$, $t = 0,10$ e $t = 0,15$. Apenas 1 a cada 5 pontos da grade numérica são mostrados, para facilitar a comparação com a solução analítica.

```

45 // -----
46 use tridiag;
47 for n in 1..nt do {           // loop no tempo
48     writeln(n);
49 // -----
50 // recaculta 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 4.2 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 4.3 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 α e β 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 4.4 Considere a equação de advecção-difusão unidimensional

$$\frac{\partial \phi}{\partial t} + U \frac{\partial \phi}{\partial x} = D \frac{\partial^2 \phi}{\partial x^2} - K\phi, \quad 0 \leq x \leq \infty, \quad t > 0, \quad (4.67)$$

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 (O’Loughlin e Bowmer, 1975; Dias, 2003)

$$\begin{aligned} \phi(x, t) = \frac{\Phi_M}{2} \left[\exp \left(\frac{Ux}{2D} (1 + (1 + 2H)^{1/2}) \right) \operatorname{erfc} \left(\frac{x + Ut(1 + 2H)^{1/2}}{\sqrt{4Dt}} \right) + \right. \\ \left. \exp \left(\frac{Ux}{2D} (1 - (1 + 2H)^{1/2}) \right) \operatorname{erfc} \left(\frac{x - Ut(1 + 2H)^{1/2}}{\sqrt{4Dt}} \right) \right] \quad (4.68) \end{aligned}$$

onde

$$H = 2KD/U^2. \quad (4.69)$$

- a) Mostre que (4.68) atende à equação diferencial. Você pode fazer tudo analiticamente com lápis e papel, ou usar Maxima.
- b) Discretize (4.67) usando um esquema *upwind* implícito para $\frac{\partial \phi}{\partial x}$ e um esquema totalmente implícito no lado direito. Faça

$$\Delta x = 0,01 = L/N, \quad (4.70)$$

$$x_i = i\Delta x. \quad (4.71)$$

- c) Resolva (4.67) 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 4.14 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 4.14

Listing 4.14: Implementação de um esquema numérico implícito para a equação da difusão-advecção.

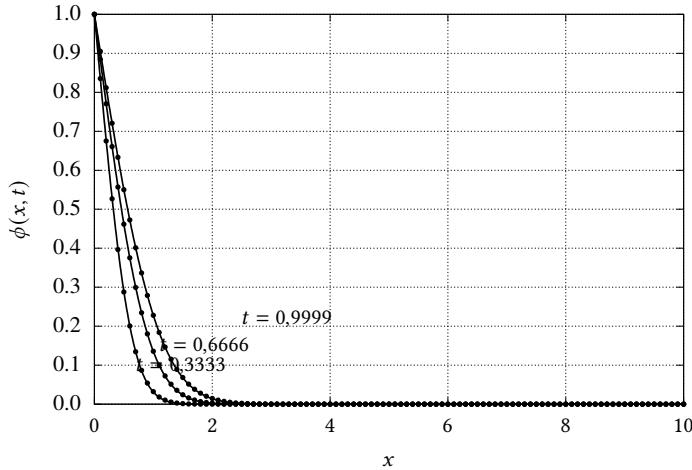


Figure 4.14: Comparação entre as soluções analítica (linhas) e numérica com um esquema implícito (pontos) da equação da difusão-advecção com termo de decaimento, para $t = 0,333$, $t = 0,666$ e $t = 0,999$.

Exercícios Propostos

4.8 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 $[A]$ (5×5) e $[B]$ (5×1) 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 $[B]$.

4.9 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 $N_x = 8$* .

4.10 O problema difusivo

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

$$\begin{aligned}\frac{\partial \phi(L, t)}{\partial x} &= 0, \\ \phi(x, 0) &= f(x),\end{aligned}$$

possui discretização

$$-\text{Fo}\phi_{i-1}^{n+1} + (1 + 2\text{Fo})\phi_i^{n+1} - \text{Fo}\phi_{i+1}^{n+1} = \phi_i^n, \quad (4.72)$$

onde

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

e $i = 1, \dots, N - 1$ (i é o índice do eixo x). Modifique (4.72) 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.

4.11 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 ϕ_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

$$\begin{aligned}\text{Fo} &= \frac{D\Delta t}{\Delta x^2}, \\ \text{Kt} &= k\Delta t.\end{aligned}$$

4.12 Dada a equação diferencial não-linear de Boussinesq,

$$\begin{aligned}\frac{\partial h}{\partial t} &= \frac{\partial}{\partial x} \left[h \frac{\partial h}{\partial x} \right], \\ h(x, 0) &= H, \\ h(0, t) &= H_0, \\ \left. \frac{\partial h}{\partial x} \right|_{x=1} &= 0,\end{aligned}$$

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.

4.13 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 ϕ_i^{n-1} , ϕ_i^n , e ϕ_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 ϕ (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
Δt	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.

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

Consider the two-dimensional diffusion equation

$$\frac{\partial \phi}{\partial t} = D \left[\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \right] \quad (4.73)$$

with simple initial and boundary conditions

$$\phi(0, x, y) = 1, \quad 0 < x < L_x, \quad 0 < y < L_y, \quad (4.74)$$

$$\phi(t, 0, y) = \phi(t, L_x, y) = 0, \quad t > 0, \quad 0 \leq y \leq M, \quad (4.75)$$

$$\phi(t, x, 0) = \phi(t, x, L_y) = 0, \quad t > 0, \quad 0 \leq x \leq L. \quad (4.76)$$

This has an analytical solution (modified from [Kreider et al. \(1966, section 13–7\)](#)) of the form

$$\phi(t, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} A_{mn} \sin\left(\frac{(2m+1)\pi x}{L_x}\right) \sin\left(\frac{(2n+1)\pi y}{L_y}\right) \times \exp\left\{-\pi^2 D \left[((2m+1)/L_x)^2 + ((2n+1)/L_y)^2 \right] t\right\}, \quad (4.77)$$

where

$$A_{mn} = \frac{16}{\pi^2 (2m+1)(2n+1)}. \quad (4.78)$$

The values of the initial condition $\phi(x, y, 0)$ in (4.74) at $x = 0$, $x = L_x$, $y = 0$ and $y = L_y$ 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 (4.77) 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,

$$x_i = i\Delta x, \quad i = 0, \dots, N_x, \quad (4.79)$$

$$y_j = j\Delta y, \quad j = 0, \dots, N_y, \quad (4.80)$$

$$\Delta x = L/N_x, \quad (4.81)$$

$$\Delta y = M/N_y, \quad (4.82)$$

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

For simplicity, put

$$\Delta x = \Delta y = \Delta \ell, \quad (4.83)$$

$$L = M = 1, \quad (4.84)$$

$$N_x = N_y = N_n. \quad (4.85)$$

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

Listing 4.15: `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 config const Nx = Nn;
7 config const Ny = Nn;
8 config const Nt = 1000;          // number of points in t
9 const L = 1.0;                  // common length L
10 const dl = L/Nn;               // delta l
11 const Lx = L;                  // Lx
12 const Ly = L;                  // Ly
13 const dx = dl;                // when needed
14 const dy = dl;                // when needed
15 const dt = 0.1/Nt;             // delta t
16 config const deln = 100;        // write output every deln steps
17 // writeln("# dl = %9.4dr\n", dl);
18 // writeln("# Nn = %9i\n", Nn);
19 // writeln("# Nt = %9i\n", Nt);
20 config const Dif = 2.0;         // diffusivity

```

Now we move to programming the analytical solution. Summing (4.77)–(4.78) 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 4.1. The table suggests that we use a main index k , and then run (simultaneously) “partial sums” over $m = 0, \dots, k$ and $n = k - m, \dots, 0$.

Table 4.1: First values of m , n , their sum, and $(2m + 1)$ and $(2n + 1)$ in (4.77)–(4.78).

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:

Listing 4.16: 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 //  $\phi(0,0,y) = 0$ ,  $\phi(0,1,y) = 0$ ,
8 //  $\phi(0,x,0) = 0$ ,  $\phi(0,x,1) = 0$ .
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/Lx)**2 + (c2n1/Ly)**2)*t);
69         ps += Amn*sin(c2m1*pi*x/Lx)*sin(c2n1*pi*y/Ly)*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

$$\phi_{n,i,j} \approx \phi(n\Delta t, i\Delta x, j\Delta y). \quad (4.86)$$

The ADI is a very clever solution that can still harness the simplicity of the TDMA even in a multidimensional (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 Δt . The successive discretizations are

$$\frac{\phi_{n+1,i,j} - \phi_{n,i,j}}{\Delta t} = D \left(\frac{\phi_{n+1,i+1,j} - 2\phi_{n+1,i,j} + \phi_{n+1,i-1,j}}{\Delta x^2} + \frac{\phi_{n,i,j+1} - 2\phi_{n,i,j} + \phi_{n,i,j-1}}{\Delta y^2} \right), \quad (4.87)$$

and

$$\frac{\phi_{n+2,i,j} - \phi_{n+1,i,j}}{\Delta t} = D \left(\frac{\phi_{n+1,i+1,j} - 2\phi_{n+1,i,j} + \phi_{n+1,i-1,j}}{\Delta x^2} + \frac{\phi_{n+2,i,j+1} - 2\phi_{n+2,i,j} + \phi_{n+2,i,j-1}}{\Delta y^2} \right). \quad (4.88)$$

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

$$\text{Fo} = \frac{D\Delta t}{\Delta \ell^2}. \quad (4.89)$$

Then, rearranging (4.87–4.88), we obtain the somewhat tighter versions

$$-\text{Fo} \phi_{n+1,i-1,j} + (1 + 2\text{Fo})\phi_{n+1,i,j} - \text{Fo} \phi_{n+1,i+1,j} = \text{Fo} \phi_{n,i,j-1} + (1 - 2\text{Fo})\phi_{n,i,j} + \text{Fo} \phi_{n,i,j+1}, \quad (4.90)$$

$$-\text{Fo} \phi_{n+2,i,j-1} + (1 + 2\text{Fo})\phi_{n+2,i,j} - \text{Fo} \phi_{n+2,i,j+1} = \text{Fo} \phi_{n+1,i-1,j} + (1 - 2\text{Fo})\phi_{n+1,i,j} + \text{Fo} \phi_{n+1,i+1,j}. \quad (4.91)$$

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 (4.90),

$$-(1 + 2\text{Fo})\phi_{n+1,1,j} - \text{Fo} \phi_{n+1,2,j} = \text{Fo} \phi_{n,1,j-1} + (1 - 2\text{Fo})\phi_{n,1,j} + \text{Fo} \phi_{n,1,j+1} + \text{Fo} \phi_{n+1,0,j}, \quad (4.92)$$

$$\begin{aligned}
 & -\text{Fo} \phi_{n+1,N_x-2,j} + (1 + 2\text{Fo})\phi_{n+1,N_x-1,j} = \\
 & \quad \text{Fo} \phi_{n,N_x-1,j-1} + (1 - 2\text{Fo})\phi_{n,N_x-1,j} + \text{Fo} \phi_{n,N_x-1,j+1} + \text{Fo} \phi_{n+1,N_x,j}, \quad (4.93)
 \end{aligned}$$

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

Listing 4.17: 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("rtimes= ", 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) :

Listing 4.18: 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 ~ 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`:

Listing 4.19: `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...); // I am seeking a range ...
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 4.15. 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`.

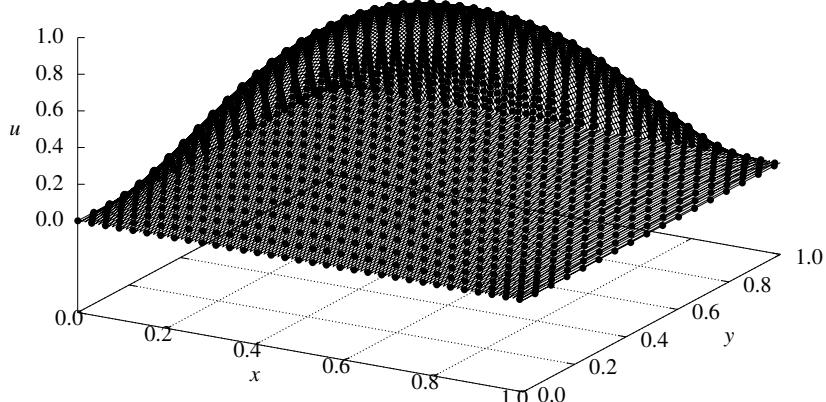


Figure 4.15: Comparison between the analytical (mesh) and numerical (points; ADI) solutions of the two-dimensional diffusion equation.

Exemplo 4.5 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 Δt e dividindo por Δx^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:

$$e^{a\Delta t} - 1 = \text{Fo} [e^{a\Delta t} e^{+ik_l \Delta s} - 2e^{a\Delta t} + e^{a\Delta t} e^{-ik_l \Delta s} + e^{+ik_m \Delta s} - 2 + e^{-ik_m \Delta s}] ;$$

$$e^{a\Delta t} [1 - \text{Fo}(e^{+ik_l \Delta s} - 2 + e^{-ik_l \Delta s})] = 1 + \text{Fo} [e^{+ik_m \Delta s} - 2 + e^{-ik_m \Delta s}],$$

$$e^{a\Delta t} [1 - 2\text{Fo}(\cos(k_l \Delta s) - 1)] = 1 + 2\text{Fo}(\cos(k_m \Delta s) - 1),$$

$$|e^{a\Delta t}| = \left| \frac{1 + 2\text{Fo}(\cos(k_m \Delta s) - 1)}{1 - 2\text{Fo}(\cos(k_l \Delta s) - 1)} \right|,$$

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

4.4 – 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

$$U \frac{\partial C}{\partial x} = K \frac{\partial^2 C}{\partial z^2}, \quad (4.94)$$

$$\frac{\partial C}{\partial x} = \alpha^2 \frac{\partial^2 C}{\partial z^2}, \quad (4.95)$$

$$\frac{\partial C(x, 0)}{\partial z} = \frac{\partial C(x, h)}{\partial z} = 0, \quad (4.96)$$

$$C(0, z) = \frac{Q}{U} B(z), \quad (4.97)$$

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

$$\alpha^2 = \frac{K}{U}, \quad (4.98)$$

$$B(z) = \begin{cases} \frac{1}{\sigma}, & |z - z_e| \leq \sigma/2, \\ 0, & |z - z_e| > \sigma/2. \end{cases} \quad (4.99)$$

Em (4.99), 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

$$C(x, z) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n e^{-k_n^2 \alpha^2 x} \cos(k_n z), \quad (4.100)$$

onde

$$k_n = \frac{\pi n}{h}. \quad (4.101)$$

É fácil verificar que a solução postulada atende à equação diferencial (4.95). Além disso, observe que (4.100) atende automaticamente às condições de contorno (4.96). Finalmente, a integral em z de (4.100) é constante:

$$\int_0^h C(x, z) dz = \frac{a_0 h}{2}. \quad (4.102)$$

Precisamos dos coeficientes de Fourier: em $x = 0$,

$$\frac{Q}{U} B(z) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(k_n z). \quad (4.103)$$

Para calcular a_0 , simplesmente integre:

$$\begin{aligned} \frac{Q}{U} \int_0^h B(z) dz &= h \frac{a_0}{2} + \underbrace{\sum_{n=1}^{\infty} \int_0^h a_n \cos(k_n z) dz}_{\equiv 0} \\ a_0 &= \frac{2Q}{hU}. \end{aligned} \quad (4.104)$$

Para os demais coeficientes, $m > 0$,

$$\begin{aligned} \frac{Q}{U} B(z) \cos(k_m z) &= \frac{a_0}{2} \cos(k_m z) + \sum_{n=1}^{\infty} a_n \cos(k_n z) \cos(k_m z), \\ Q \int_0^h B(z) \frac{\cos(k_m z)}{U} dz &= \frac{a_0}{2} \int_0^h \cos(k_m z) dz \\ &\quad + \sum_{n=1}^{\infty} a_n \int_0^h \cos(k_n z) \cos(k_m z) dz, \end{aligned}$$

$$\begin{aligned}
\frac{Q}{U\sigma} \int_{z_e-\sigma/2}^{z_e+\sigma/2} \cos(k_m z) dz &= \frac{a_0}{2} \int_0^h \cos(k_m z) dz \\
&\quad + \sum_{n=1}^{\infty} a_n \int_0^h \cos(k_n z) \cos(k_m z) dz, \\
\frac{2Q}{U\sigma k_m} \sin\left(\frac{k_m \sigma}{2}\right) \cos(k_m z_e) &= \frac{a_0}{2} \int_0^h \cos(k_m z) dz \\
&\quad + \sum_{n=1}^{\infty} a_n \int_0^h \cos(k_n z) \cos(k_m z) dz.
\end{aligned} \tag{4.105}$$

As funções no lado direito de (4.105) são ortogonais:

$$\int_0^h \cos(k_n z) \cos(k_m z) dz = \begin{cases} 0, & m \neq n, \\ h/2 & m = n \neq 0. \end{cases} \tag{4.106}$$

Segue-se que

$$a_m = \frac{4Q}{U\sigma\pi m} \sin\left(\frac{k_m \sigma}{2}\right) \cos(k_m z_e). \tag{4.107}$$

- a) Programe a solução analítica truncando a série do 200º harmônico (**obrigatoriamente**):

$$\widehat{C}(x, z) \approx \frac{a_0}{2} + \sum_{n=1}^{N=200} a_n e^{-k_n^2 \alpha^2 x} \cos(k_n z). \tag{4.108}$$

Discuta a convergência da série de Fourier para valores de N diferentes de 200. Plote alguns resultados.

- b) Resolva (4.95) numericamente, utilizando o esquema implícito (4.58). *Cuidado! O tratamento das condições de contorno é por sua conta.*

A condição inicial, dada por (4.97), 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 Δz bem menor do que σ .
2. Defina $B(z)$ por pontos com resolução Δz ; 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

$$\frac{\partial\phi}{\partial t} - U \frac{\partial\phi}{\partial x} - D \frac{\partial^2 u}{\partial x^2} = 0. \quad (4.109)$$

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 (4.109) usando um esquema implícito centrado para $\frac{\partial\phi}{\partial x}$ e $\frac{\partial^2\phi}{\partial x^2}$. Faça

$$\Delta x = 0,001, \quad (4.110)$$

$$\Delta t = 0,001. \quad (4.111)$$

- b) Resolva (4.109) 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 . 4.16

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 (Kundu, 1990, seção 13.7) — e irrealista! — postula uma viscosidade cinemática turbulenta ν_W constante. O vetor velocidade é (u, v) , e as equações do movimento nas direções x e y são, respectivamente,

$$0 = fv + \nu_W \frac{d^2 u}{dz^2}, \quad (4.112)$$

$$0 = f(U - u) + \nu_W \frac{d^2 v}{dz^2}. \quad (4.113)$$

O vetor $(U, 0)$ é denominado *vento geostrófico*, e $f \cong 1 \times 10^{-4} \text{ s}^{-1}$ (a 45° de latitude Norte) é o parâmetro de Coriolis. As condições de contorno do problema são

$$u(0) = 0, \quad v(0) = 0, \quad (4.114)$$

$$u(+\infty) = U, \quad v(+\infty) = 0. \quad (4.115)$$

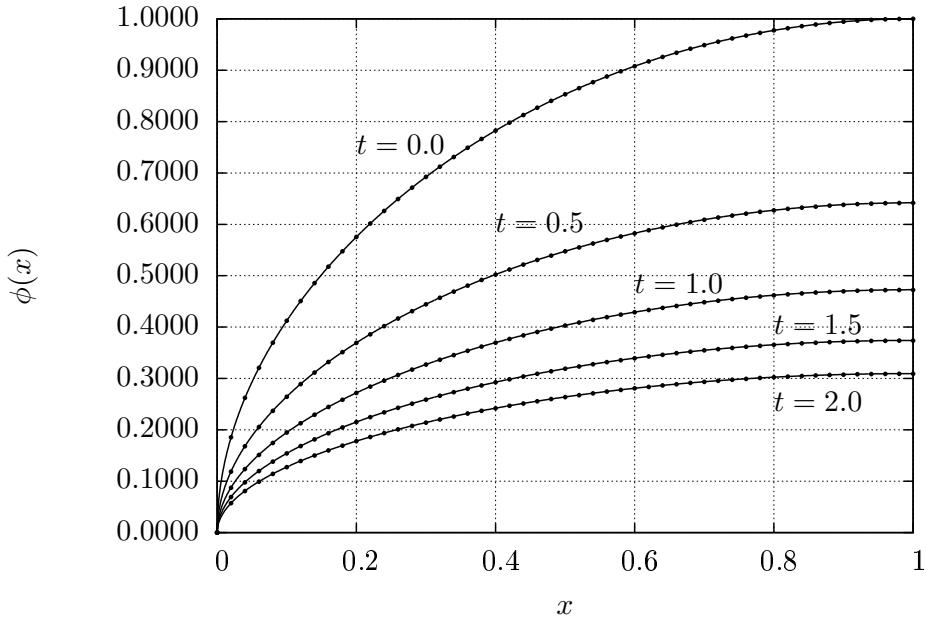


Figure 4.16: Solução correta do problema para $t = 0, 0.5, 1.0, 1.5, 2.0$. As linhas cheias são a solução analítica, e os pontos a solução numérica.

A solução apresentada por Kundu usa variáveis complexas. Constrói-se uma velocidade complexa

$$W \equiv U + iV \quad (4.116)$$

($i = \sqrt{-1}$) de tal forma que (4.112)–(4.113) podem ser escritas compactamente

$$\frac{d^2 W}{dz^2} - \frac{if}{v_W} (W - U) = 0 \quad (4.117)$$

(Cuidado! z é *real!*). A equação tem solução

$$W = U \left[1 - e^{-(1+i)z/\delta} \right], \quad (4.118)$$

com

$$\delta \equiv \sqrt{\frac{2v_W}{f}}, \quad (4.119)$$

ou seja:

$$u = U \left[1 - e^{-z/\delta} \cos(z/\delta) \right], \quad (4.120)$$

$$v = U e^{-z/\delta} \sin(z/\delta). \quad (4.121)$$

O seu trabalho é obter uma aproximação numérica para (4.118) (ou, o que dá no mesmo, para (4.120)–(4.121)), resolvendo o problema *transiente*

$$\frac{\partial W}{\partial t} = \frac{\partial^2 W}{\partial z^2} - \frac{if}{v_W} (W - U) = 0. \quad (4.122)$$

com as condições iniciais e de contorno

$$W(z, 0) = 0, \quad (4.123)$$

$$W(0, t) = 0, \quad (4.124)$$

$$W(H, t) = U, \quad (4.125)$$

observando que $W(z, \infty)$ (quando $\partial W / \partial t \rightarrow 0$) é a solução de (4.117). 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 (4.122) no computador, é muito útil adimensionalizar! Fazemos

$$\begin{aligned} t &= f\tau, \\ z &= \delta\zeta, \\ W &= \phi U, \end{aligned}$$

e obtemos o problema (já com os valores numéricos indicados acima):

$$\frac{\partial \phi}{\partial \tau} = \frac{1}{2} \frac{\partial^2 \phi}{\partial \zeta^2} - i(\phi - 1). \quad (4.126)$$

As condições inciais e de contorno são

$$\phi(\zeta, 0) = 0, \quad (4.127)$$

$$\phi(0, t) = 0, \quad (4.128)$$

$$\phi(10, t) = 1. \quad (4.129)$$

O problema a ser resolvido numericamente agora é:

- Discretize a equação usando um esquema totalmente implícito para $\frac{\partial^2 \phi}{\partial \zeta^2}$ e para ϕ . Você deve descrever a discretização e explicar o esquema numérico resultante.
- 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 4.17 e 4.18. Elas mostram as partes real e imaginária de ϕ (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 (Boussinesq, 1903)

$$\frac{\partial h}{\partial t} = \frac{k_0}{n_e} \frac{\partial}{\partial x} \left[h \frac{\partial h}{\partial x} \right], \quad t \geq 0; \quad 0 \leq x \leq B \quad (4.130)$$

com condições iniciais e de contorno

$$h(x, 0) = H, \quad h(0, t) = H_0 < H, \quad \frac{\partial h(B, t)}{\partial x} = 0 \quad (4.131)$$

ela pode ser adimensionalizada da seguinte forma (Brutsaert, 2005, seção 10.3.5):

$$\phi \equiv \frac{h}{H}, \quad \eta \equiv \frac{x}{B}, \quad \tau \equiv \frac{k_0 H}{n_e B^2} t. \quad (4.132)$$

Em (4.130)–(4.132), 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.

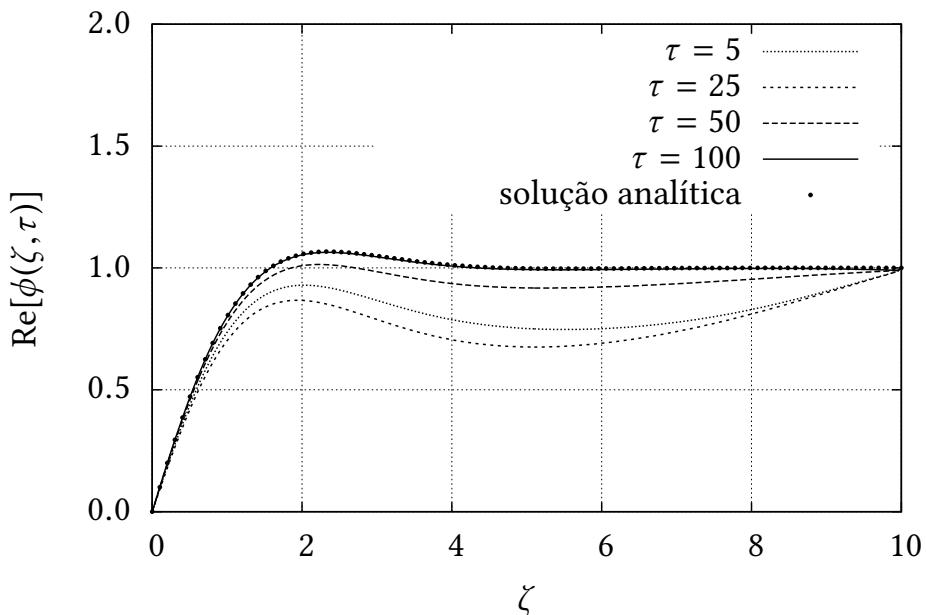


Figure 4.17: Parte real de $\phi(\zeta, \tau)$, $\tau = 5, 25, 50, 100$, e solução analítica.

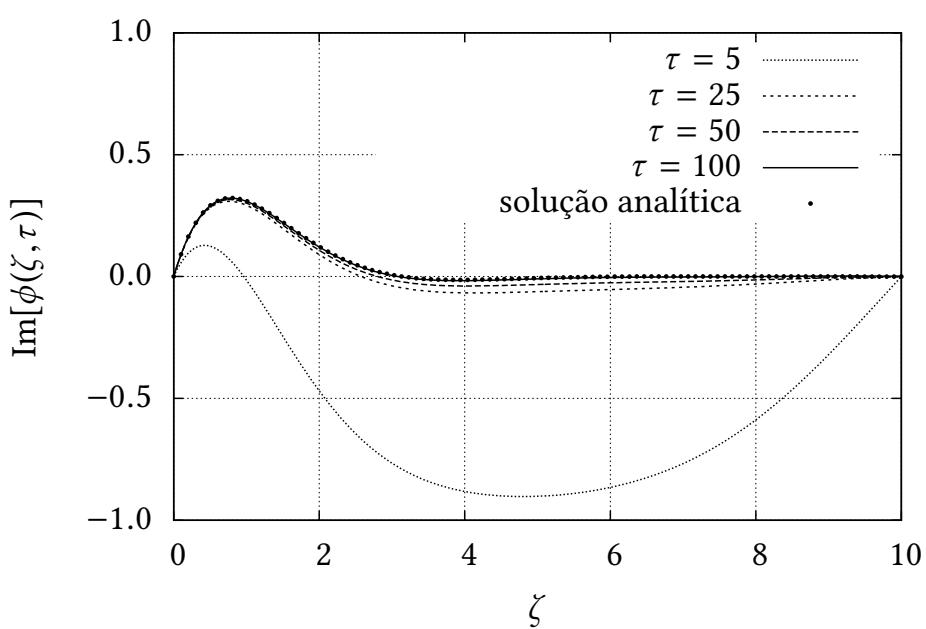


Figure 4.18: Parte imaginária de $\phi(\zeta, \tau)$, $\tau = 5, 25, 50, 100$, e solução analítica.

A vazão em $x = 0$ também pode ser adimensionalizada:

$$\begin{aligned} q(t) &= k_0 h(0, t) \frac{\partial h(0, t)}{\partial x} \\ &= k_0 H^2 \frac{h(0, t)}{H} \frac{1}{B} \frac{\partial \frac{h(0, t)}{H}}{\partial \frac{x}{B}} \\ &= \frac{k_0 H^2}{B} \left[\phi \frac{\partial \phi}{\partial \eta} \right] (0, \tau) \\ &= \frac{k_0 H^2}{B} \chi(\tau), \end{aligned} \quad (4.133)$$

onde $\chi(\tau)$ é a vazão adimensional efluente do maciço poroso em $\eta = 0$.

Nas variáveis acima, a equação pode ser reescrita como

$$\frac{\partial \phi}{\partial \tau} = \frac{\partial}{\partial \eta} \left[\phi \frac{\partial \phi}{\partial \eta} \right] \quad \tau \geq 0, \quad 0 \leq \eta \leq 1, \quad (4.134)$$

com condições iniciais e de contorno

$$\phi(\eta, 0) = 1, \quad \phi(0, \tau) = \Phi_0 < 1, \quad \frac{\partial \phi(1, \tau)}{\partial \eta} = 0. \quad (4.135)$$

Tempos longos

Quando $\Phi_0 = 0$, com a condição inicial

$$\phi(\eta, 0) = F(\eta) \quad (4.136)$$

substituindo a primeira das equações (4.135), existe a solução analítica (Boussinesq, 1904):

$$\phi(\eta, t) = \frac{F(\eta)}{1 + \alpha \tau}, \quad (4.137)$$

$$\chi(\tau) = \frac{\beta}{(1 + \alpha \tau)^2}, \quad (4.138)$$

$$\alpha = [B(2/3, 1/2)]^2 / 6, \quad (4.139)$$

$$\beta = B(2/3, 1/2)/3, \quad (4.140)$$

$$\eta = I_{F^3(2/3, 1/2)}, \quad (4.141)$$

onde $B(\alpha, \beta)$ é a função Beta, e $I_x(\alpha, \beta)$ é a função Beta incompleta (Press et al., 1992, 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 (4.134).

Tempos longos, solução linearizada

Considere novamente (4.130), linearizada:

$$\begin{aligned} \frac{\partial h}{\partial t} &= \frac{pk_0 H}{n_e} \frac{\partial^2 h}{\partial x^2}, \\ \frac{\partial(h/H)}{\partial t} &= \frac{pk_0 H}{n_e} \frac{\partial^2(h/H)}{\partial x^2}, \\ \frac{\partial \phi}{\partial t} &= \frac{pk_0 H}{n_e} \frac{\partial^2 \phi}{\partial x^2}, \end{aligned}$$

$$\begin{aligned}\frac{\partial \phi}{\partial t} &= \frac{pk_0H}{n_e B^2} \frac{\partial^2 \phi}{\partial \eta^2}, \\ \frac{\partial \phi}{\partial \tau} &= p \frac{\partial^2 \phi}{\partial \eta^2}.\end{aligned}\quad (4.142)$$

A equação (4.142) tem solução em série de Fourier, da forma

$$\phi(\eta, \tau) = \Phi_0 + \frac{4(1 - \Phi_0)}{\pi} \sum_{n=1}^{\infty} \frac{1}{(2n-1)} \sin\left((2n-1)\frac{\pi}{2}\eta\right) \exp\left(-\frac{(2n-1)^2\pi^2}{4}p\tau\right). \quad (4.143)$$

Seguindo Brutsaert e Lopez (1998) e Chor e Dias (2015), nós calcularemos p em função de Φ_0 como:

$$p = 0.3465 + 0.6535\Phi_0. \quad (4.144)$$

Nossos objetivos

O objetivo deste trabalho são:

1. Produzir um esquema numérico de solução que possa ser comparado com a solução analítica (4.137)–(4.141) 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.

1ª tarefa

Crie um programa que calcula a solução analítica dada pelas equações (4.136)–(4.141). Verifique como os perfis $\phi(\eta, \tau)$ evoluem no tempo. Para fazer isso, ajudará muito se você instalar o módulo adicional `scipy`. As funções Beta e Beta incompleta que aparecem em (4.139), (4.140) e (4.141) 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

$$\phi \frac{\partial \phi}{\partial \eta} \Big|_{i+1/2} \approx \frac{\phi_{i+1}^n + \phi_i^n}{2} \left[\frac{\phi_{i+1}^{n+1} - \phi_i^{n+1}}{\Delta \eta} \right]; \quad (4.145)$$

$$\phi \frac{\partial \phi}{\partial \eta} \Big|_{i-1/2} \approx \frac{\phi_i^n + \phi_{i-1}^n}{2} \left[\frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta \eta} \right]; \quad (4.146)$$

como estamos usando o “ ϕ ” do passo tempo anterior (n), isso na prática lineariza o esquema. Vamos simplificar a notação:

$$\bar{\phi}_i^n \equiv \frac{\phi_{i+1}^n + \phi_i^n}{2}, \quad \bar{\phi}_{i-1}^n \equiv \frac{\phi_i^n + \phi_{i-1}^n}{2}, \quad (4.147)$$

o que nos permite reescrever (4.145)–(4.146) como

$$\phi \frac{\partial \phi}{\partial \eta} \Big|_{i+1/2} \approx \bar{\phi}_i^n \left[\frac{\phi_{i+1}^{n+1} - \phi_i^{n+1}}{\Delta \eta} \right]; \quad \phi \frac{\partial \phi}{\partial \eta} \Big|_{i-1/2} \approx \bar{\phi}_{i-1}^n \left[\frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta \eta} \right]. \quad (4.148)$$

Podemos agora calcular a segunda derivada (aproximada):

$$\frac{\partial}{\partial \eta} \left[\phi \frac{\partial \phi}{\partial \eta} \right]_i \approx \frac{1}{\Delta \eta} \left[\phi \frac{\partial \phi}{\partial \eta} \Big|_{i+1/2} - \phi \frac{\partial \phi}{\partial \eta} \Big|_{i-1/2} \right]$$

$$= \frac{1}{\Delta\eta} \left[\bar{\phi}_i^n \left[\frac{\phi_{i+1}^{n+1} - \phi_i^{n+1}}{\Delta\eta} \right] - \bar{\phi}_{i-1}^n \left[\frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta\eta} \right] \right]. \quad (4.149)$$

O esquema numérico agora é

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta\tau} = \frac{1}{\Delta\eta} \left[\bar{\phi}_i^n \left[\frac{\phi_{i+1}^{n+1} - \phi_i^{n+1}}{\Delta\eta} \right] - \bar{\phi}_{i-1}^n \left[\frac{\phi_i^{n+1} - \phi_{i-1}^{n+1}}{\Delta\eta} \right] \right] \quad (4.150)$$

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:

$$- \left[\text{Fo} \bar{\phi}_{i-1}^n \right] \phi_{i-1}^{n+1} + \left[1 + \text{Fo} \left(\bar{\phi}_{i-1}^n + \bar{\phi}_i^n \right) \right] \phi_i^{n+1} - \left[\text{Fo} \bar{\phi}_i^n \right] \phi_{i+1}^{n+1} = \phi_i^n \quad (4.151)$$

Talvez seja prático definir os coeficientes a seguir. Para $1 \leq i \leq N_x - 2$:

$$A_{i-1}^n \equiv - \left[\text{Fo} \bar{\phi}_{i-1}^n \right], \quad (4.152)$$

$$B_{i-1}^n \equiv \left[1 + \text{Fo} \left(\bar{\phi}_{i-1}^n + \bar{\phi}_i^n \right) \right], \quad (4.153)$$

$$C_{i-1}^n \equiv - \left[\text{Fo} \bar{\phi}_i^n \right], \quad (4.154)$$

e então reescrever

$$A_{i-1}^n \phi_{i-1}^{n+1} + B_{i-1}^n \phi_i^{n+1} + C_{i-1}^n \phi_{i+1}^{n+1} = \phi_i^n. \quad (4.155)$$

As condições de contorno agora são as seguintes: para a primeira linha, teremos

$$B_0^n \phi_1^{n+1} + C_0^n \phi_2^{n+1} = \phi_i^n - A_0^n \Phi_0. \quad (4.156)$$

Já na última linha, $i = N_x - 1$, e teremos

$$\phi_{N_x-1}^{n+1} = \phi_{N_x}^{n+1} \quad (4.157)$$

e portanto

$$A_{N_x-2}^n \equiv - \left[\text{Fo} \bar{\phi}_{N_x-2}^n \right], \quad (4.158)$$

$$B_{N_x-2}^n \equiv \left[1 + \text{Fo} \left(\bar{\phi}_{N_x-2}^n \right) \right]; \quad (4.159)$$

a última linha será

$$A_{N_x-2}^n \phi_{N_x-1}^{n+1} + B_{N_x-2}^n \phi_{N_x-1}^{n+1} = \phi_{N_x-1}^n. \quad (4.160)$$

Como sempre, uma boa notação torna as coisas quase triviais: notação é importante!

2ª 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 (4.136). Sugestão: use incrementos $\Delta\eta = 0,0001$ e $\Delta\tau = 0,0001$. Minha comparação é excelente, como podemos ver na figura 4.19. Os instantes mostrados do esquema numérico são os mesmos de antes, é claro: $\tau = 0,01$, $\tau = 0,1$ e $\tau = 1,0$.

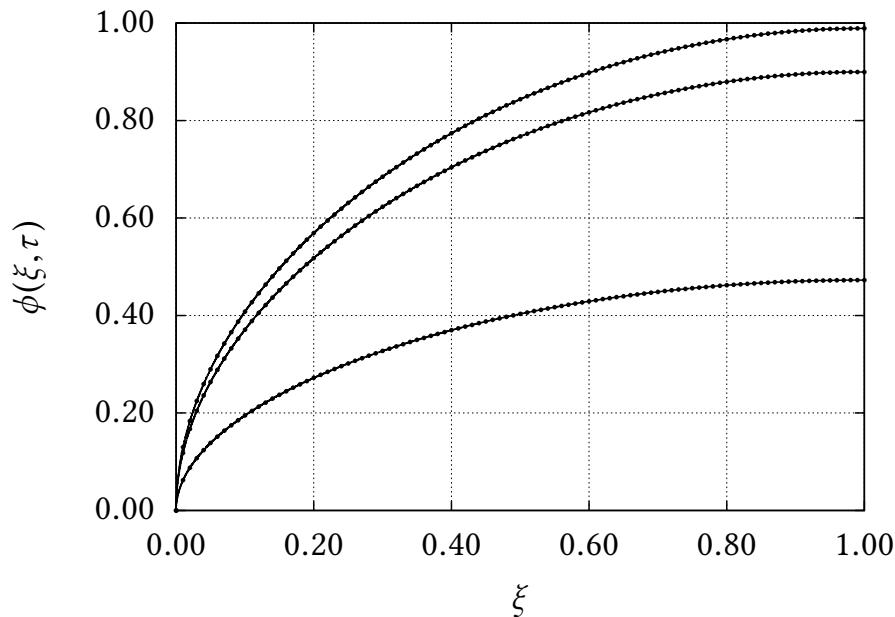


Figure 4.19: Comparação do esquema numérico com a solução analítica de [Boussinesq \(1904\)](#). De cima para baixo, $\tau = 0,01$, $\tau = 0,1$, e $\tau = 1,0$

3^a tarefa

Escreva um programa para calcular a solução analítica aproximada dada pela equação (4.143). O valor de Φ_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^a tarefa

Agora escreva um programa para resolver o problema definido pelas equações (4.134)–(4.135) numericamente. Esse programa também deve ter com argumento Φ_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 4.20 mostra a minha comparação para $\phi(\eta, 0) = 0$. Repita, para $\Phi_0 = 0,25$, $0,50$ e $0,75$. (vide figuras 4.21, 4.22 e 4.23).

Como você interpreta os resultados obtidos em função de Φ_0 ?

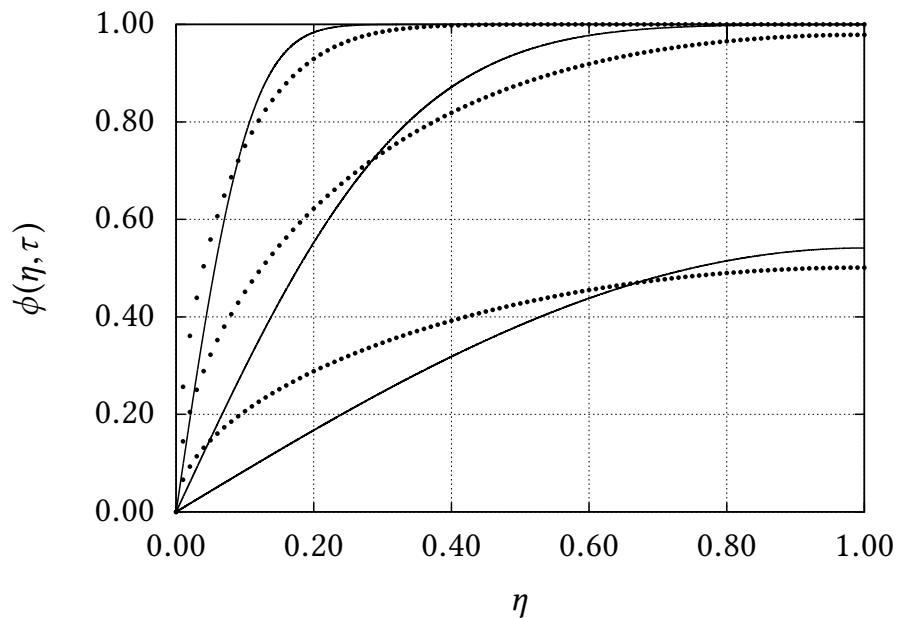


Figure 4.20: Comparação da solução numérica completa de (4.134)–(4.135) com a solução linearizada (4.143) para $\Phi_0 = 0$. De cima para baixo, $\tau = 0,01$, $\tau = 0,1$, e $\tau = 1,0$.

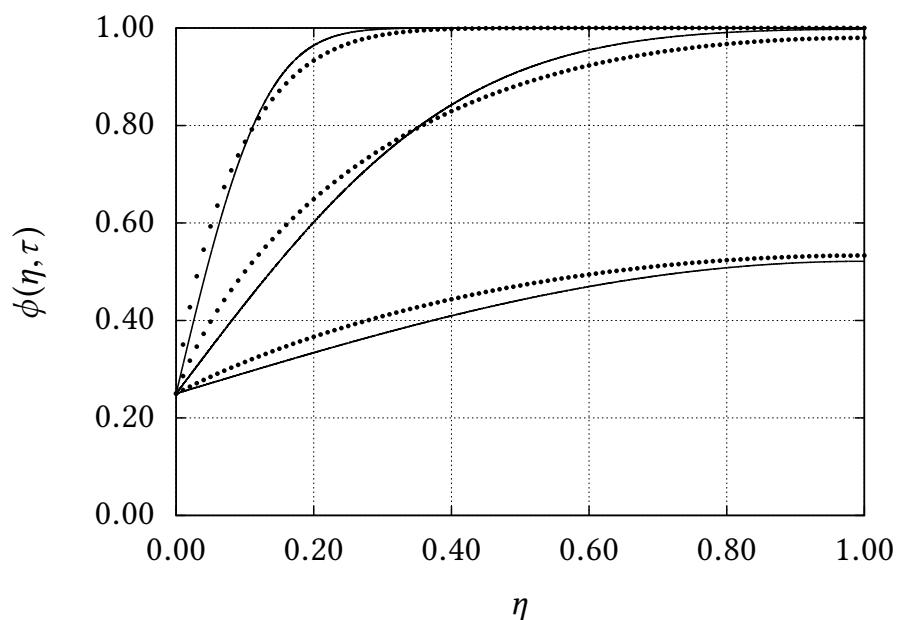


Figure 4.21: Comparação da solução numérica completa de (4.134)–(4.135) com a solução linearizada (4.143) para $\Phi_0 = 0,25$. De cima para baixo, $\tau = 0,01$, $\tau = 0,1$, e $\tau = 1,0$.

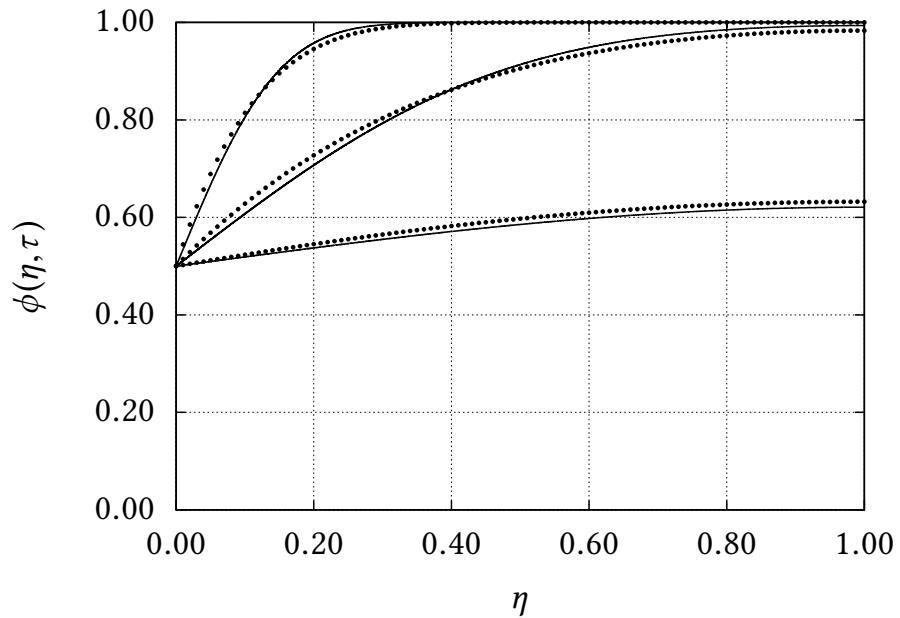


Figure 4.22: Comparação da solução numérica completa de (4.134)–(4.135) com a solução linearizada (4.143) para $\Phi_0 = 0,50$. De cima para baixo, $\tau = 0,01$, $\tau = 0,1$, e $\tau = 1,0$.

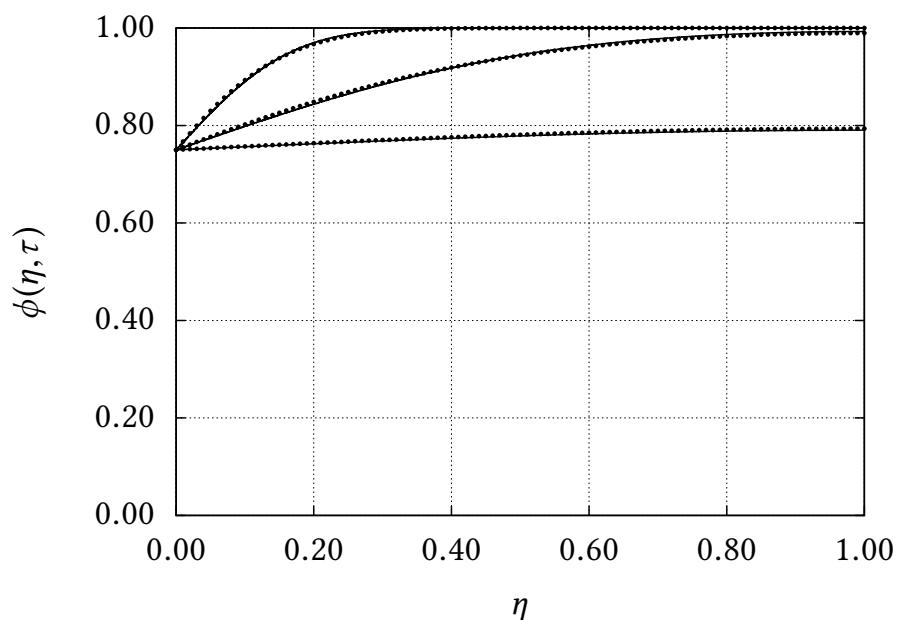


Figure 4.23: Comparação da solução numérica completa de (4.134)–(4.135) com a solução linearizada (4.143) para $\Phi_0 = 0,75$.

4.5 – Successive over-relaxation

Given Laplace's equation in two dimensions,

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0, \quad (4.161)$$

a standard discretization is

$$\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y^2} = 0. \quad (4.162)$$

With $\Delta x = \Delta y$, we obtain

$$\phi_{i,j} = \frac{1}{4} [\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}]. \quad (4.163)$$

Equation 4.163 is the basis for an over-relaxation method to solve (4.161) numerically. For any iterative method of the type

$$\mathbf{u}^{k+1} = f(\mathbf{u}^k), \quad (4.164)$$

where f defines the iterative method, a relaxation method is

$$\mathbf{u}^{k+1} = \omega f(\mathbf{u}^k) + (1 - \omega) \mathbf{u}^k. \quad (4.165)$$

where ω is the *relaxation factor*. When $1 < \omega < 2$, convergence is accelerated, and the method is called *successive over-relaxation* (SOR) (Press et al., 1992, section 19.5). A somewhat more convenient form for computation is

$$\begin{aligned} \mathbf{u}^{k+1} &= \omega [f(\mathbf{u}^k) - \mathbf{u}^k] + \mathbf{u}^k; \\ \delta \mathbf{u}^{k+1} &\equiv \mathbf{u}^{k+1} - \mathbf{u}^k = \omega [f(\mathbf{u}^k) - \mathbf{u}^k]. \end{aligned} \quad (4.166)$$

The term in brackets in (4.166) above is the original increment of the iterative method (4.164); in this sense, over-relaxation is simply to multiply each step of the iterative method by ω ; for $\omega > 1$, this (often) 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

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \delta \mathbf{u}^{k+1}. \quad (4.167)$$

The main limitation of the SOR method is finding the “best” ω . Often, the best value is found by trial-and-error, which requires considerable effort. Moreover, ω in general depends of the problem (*i.e.* the function f) and the grid size as well.

Consider then (4.161) with boundary conditions

$$\phi(x, 0) = x, \quad \phi(0, y) = y, \quad (4.168)$$

$$\phi(x, 1) = 1, \quad \phi(1, y) = 1. \quad (4.169)$$

It is straightforward to verify that the analytical solution is

$$\phi(x, y) = x + y - xy. \quad (4.170)$$

It is now relatively simple to devise an iterative solution on the basis of (4.163), by writing

$$\phi_{i,j}^{k+1} = \frac{1}{4} [\phi_{i+1,j}^k + \phi_{i-1,j}^k + \phi_{i,j+1}^k + \phi_{i,j-1}^k], \quad (4.171)$$

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. A very important point is that for the Laplace/Poisson equation, the optimal ω is known analytically [Hovland e Heath \(1997\)](#); [Yang e Gobbert \(2009\)](#) and is given by

$$\rho = \frac{1}{2} \left(\cos\left(\frac{\pi}{N_x}\right) + \cos\left(\frac{\pi}{N_y}\right) \right), \quad (4.172)$$

$$\omega = \frac{2}{1 + \sqrt{1 - \rho^2}}. \quad (4.173)$$

The serial implementation of the SOR method for Laplace's equation is fairly straightforward: we can see it in program `laplace-sor`:

Listing 4.20: `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 difgrid2d;
6 use Math only cos, pi, sqrt;
7 config const epsilon = 1.0e-8;      // accuracy
8 // -----
9 // the optimal omega is known analytically
10 // -----
11 const rho = (cos(pi/Nx) + cos(pi/Ny))/2.0;
12 config const omega = 2.0/(1.0+sqrt(1-rho**2));
13 var phi: [0..Nn,0..Nn] real = 0.0; // iterative solution
14 for i in 0..Nn do {           // the initial guess and BCs!!!
15     for j in 0..Nn do {
16         phi[i,j] = IG(i,j);
17     }
18 }
19 var runtime: stopwatch;
20 var deltam = epsilon;           // the norm
21 var nc = 0;                  // # of iterations till convergence
22 runtime.start();
23
24 while deltam >= epsilon do { // start iterations
25     deltam = 0.0;
26     for i in 1..Nn-1 do {
27         for j in 1..Nn-1 do {
28             var phiavg = (phi[i+1,j]+phi[i-1,j]+phi[i,j-1]+phi[i,j+1])/4.0;
29             var deltaphi = omega*(phiavg - phi[i,j]);
30             phi[i,j] += deltaphi;
31             deltaphi = abs(deltaphi);
32             deltam += deltaphi;
33         }
34     }
35     deltam /= ((Nn-1)**2) ;
36     nc += 1;
37 }                                // done
38 runtime.stop();
39 var sum = 0.0;
40 var mad = 0.0;
41 for i in 0..Nn do {
42     for j in 0..Nn do {
43         sum += phi[i,j];
44         var del = abs(phi[i,j] - ana(i,j));
45         mad += del;
46     }
47 }
48 mad /= ((Nn+1)**2);
49 writeref("#_Nn_=,%i", Nn);

```

```

50  writef("omega=%8.4dr",omega);
51  writef("nc=%6i",nc);
52  writef("phi_avg=%8.4dr",sum/((Nn+1)**2));
53  writef("mad=%8.4er",mad);
54  writef("rt=%8.4dr\n",runtime.elapsed());
55 // for Nn=2048 the running time at home is 69.3557
56 //
57 // --> initial guess
58 //
59 private inline proc IG(i,j: int): real {
60     const x = i*dl;
61     const y = j*dl;
62     if i == 0 then {
63         return y;
64     }
65     else if i == Nn then {
66         return 1.0;
67     }
68     else if j == 0 then {
69         return x;
70     }
71     else if j == Nn then {
72         return 1.0;
73     }
74     else {
75         return 0.50;
76     }
77 }
78 //
79 // --> the analytical solution
80 //
81 private inline proc ana(i,j): real {
82     const x = i*dl;
83     const y = j*dl;
84     return x + y - x*y;
85 }
```

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 $\text{MAD} = 2.07 \times 10^{-7}$. 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 ϕ over the domain, whose analytical value is

$$\bar{\phi} = \int_{x=0}^1 \int_{y=0}^1 (x + y - xy) \, dy \, dx = \frac{3}{4}. \quad (4.174)$$

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.

We can now run `laplace-sor` 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 4.2.

The SOR method is more efficient than, say, Gaussian elimination. The latter is an $O(n^3)$ (order of n^3) algorithm (Press et al., 1992, section 2.2), whereas SOR is $O(n^2)$, where $n = N_n^2$ is the number of unknowns. Figure 4.24, obtained from table 4.2 above by plotting $N_n^2/1000 \times t_r$, shows that the running time is well represented by a function of the type $t_r = an^2$.

Table 4.2: Grid size N_n , optimal over-relaxation parameter ω , number of iterations to convergence n_c , estimated \bar{u} , MAD and relative runtime t_r for the serial version of the solution of Laplace’s equation with SOR.

N_n	ω	n_c	\bar{u}	MAD	t_r
128	1.9521	379	0.7500	2.0768×10^{-7}	0.0186
256	1.9758	734	0.7500	3.6744×10^{-7}	0.1484
512	1.9878	1410	0.7500	7.7520×10^{-7}	1.1848
1024	1.9939	2670	0.7500	1.8319×10^{-6}	9.0633
2048	1.9969	5064	0.7499	3.7615×10^{-6}	69.3557

4.6 – Really efficient Successive Over-Relaxation

So far, we have chosen simplicity over efficiency, but SOR methods can still be made much more efficient, while avoiding race conditions in the (upcoming) parallel versions at the same time. This consists of applying (4.171) alternately on the “black” and “white” grid points shown in figure 4.25: 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 (Press et al., 1992; Hansen, 1992). In each half-sweep, the new values only depend on the old ones, and race conditions never appear.

Putting everything together, we now have the program `laplace-asor-s`, in listing 4.21.

Listing 4.21: `laplace-asor-s` — Solution of Laplace’s equation in 2D over staggered grids using Chebyshev acceleration.

```

1 // =====
2 // ==> laplace-asor-s: adaptive successive over-relaxation with Laplace's
3 // equation, serial
4 //
5 // For details, see NR and Hansen, P. B.
6 //
7 // NR == Press, W. H.; Teukolsky, S. A.; Vetterling, W. T. & Flannery,
8 // B. P. Numerical Recipes in C; The Art of Scientific Computing
9 // Cambridge University Press, 1992
10 //
11 // Hansen, P. B. Numerical solution of Laplace's equation Syracuse
12 // University, College of Engineering and Computer Science, Syracuse
13 // University, College of Engineering and Computer Science, 1992
14 // =====
15 use Time only stopwatch;
16 use difgrid2d;
17 use Math only cos,pi,sqrt;
18 var runtime: stopwatch;
19 var phi: [0..Nn,0..Nn] real = 0.0;           // iterative solution
20 runtime.start();                            // start counting time
21 for i in 0..Nn do {                      // the initial guess
22     for j in 0..Nn do {
23         phi[i,j] = IG(i,j);
24     }
25 }
26 //
27 // the optimal omega is known analytically
28 //
29 config const epsilon = 1.0e-8;             // accuracy
30 const rho = (cos(pi/Nx) + cos(pi/Ny))/2.0;
31 config const omega = 2.0/(1.0+sqrt(1-rho**2));
32 var deltam = epsilon;                     // the norm
33 var nc = 0;                             // # of iterations till convergence
34 var (r,s) = (1,2);                      // starting indices

```

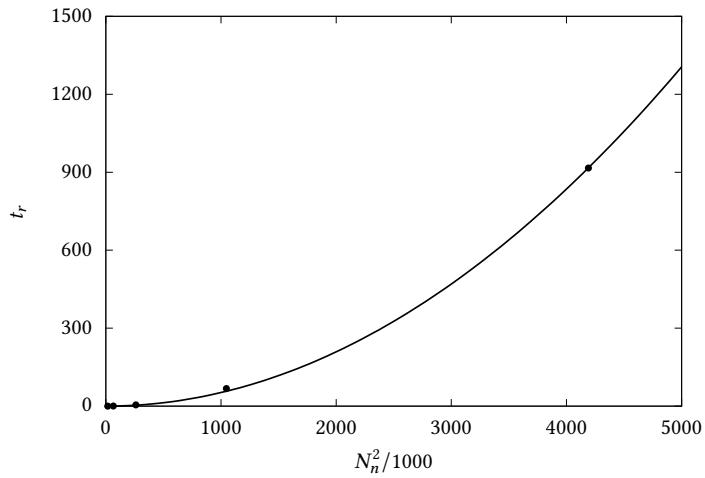


Figure 4.24: Running times t_r as a function of $n = N_n^2$ for the SOR method. The line is a parabola of the type $t_r = an^2$.

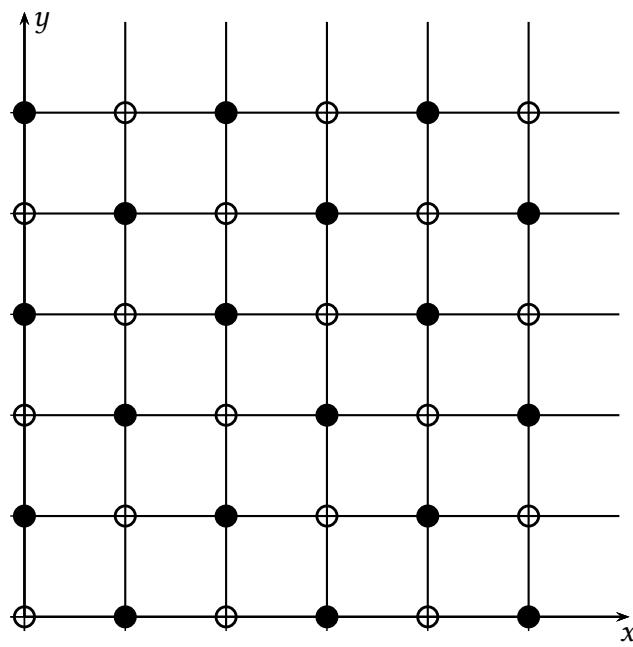


Figure 4.25: Alternating grids in 2D for the SOR methods. Note that when we update the black points using (4.171), only the white points are used, and vice-versa. In this way, we can update the whole grid in 2 half-sweeps, with the blacks depending only on the whites (and vice-versa), so that race conditions never arise.

```

35 while deltam >= epsilon do { // check convergence
36     deltam = 0.0;
37     for b in 0..1 do { // half-sweeps
38         halfsw(r,s);
39         r <=> s; // swap indices
40     }
41     nc += 1;
42     deltam /= (Nn**2);
43 }
44 runtime.stop();
45 var sum = 0.0;
46 var mad = 0.0;
47 for i in 0..Nn do {
48     for j in 0..Nn do {
49         sum += phi[i,j];
50         var del = abs(phi[i,j] - ana(i,j));
51         mad += del;
52     }
53 }
54 mad /= ((Nn+1)**2);
55 writef("# Nn=%9i", Nn);
56 writef("omega=%8.4dr", omega);
57 writef("nc=%6i", nc);
58 writef("phi_avg=%8.4dr", sum/((Nn+1)**2));
59 writef("mad=%8.4er", mad);
60 writef("rtime=%8.4dr\n", runtime.elapsed());
61 // -----
62 // --> initial guess
63 // -----
64 inline proc IG(i,j: int): real {
65     const x = i*dl;
66     const y = j*dl;
67     if i == 0 then {
68         return y;
69     }
70     else if i == Nn then {
71         return 1.0;
72     }
73     else if j == 0 then {
74         return x;
75     }
76     else if j == Nn then {
77         return 1.0;
78     }
79     else {
80         return 0.50;
81     }
82 }
83 // -----
84 // --> the analytical solution
85 // -----
86 inline proc ana(i,j): real {
87     const x = i*dl;
88     const y = j*dl;
89     return x + y - x*y;
90 }
91 // -----
92 // -->halfsw: a half-sweep
93 // -----
94 inline proc halfsw( // beginning of halfsw
95     const in r: int,
96     const in s: int
97 ) {

```

```

98  for i in 1..Nn-1 do {
99      var jinit = if i % 2 != 0 then r else s ;
100     for j in jinit..Nn-1 by 2 do {
101         var phiavg = (phi[i+1,j]+phi[i-1,j]+phi[i,j-1]+phi[i,j+1])/4.0;
102         var deltaphi = omega*(phiavg - phi[i,j]);
103         phi[i,j] += deltaphi;
104         deltaphi = abs(deltaphi);
105         deltam += deltaphi;
106     }
107 }
108 } // end of halfsw

```

The heart of the “staggering” update is programmed in procedure `halfsw` in line ??: it boils down to starting the `j` index either on 1 or 2, and incrementing it by 2. These indices are declared in line 34, and swapped in line 39.

The new approach is short of miraculous; here we compare in table 4.3 the performance of the serial programs `laplace-sor.chpl` and `laplace-asor.chpl`. The staggering update improves runtimes by a factor of approximately 4.

Table 4.3: Grid size N_n , optimal over-relaxation parameter ω , 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. All cases were run with the optimum ω .

N_n	ω	serial			accelerated serial			
		n_c	$\bar{\phi}$	MAD	t_r	n_c	$\bar{\phi}$	MAD
128	1.9521	379	0.7500	2.0768×10^{-7}	0.0186	336	0.7500	1.9599×10^{-7}
256	1.9758	734	0.7500	3.6744×10^{-7}	0.1492	644	0.7500	3.7893×10^{-7}
512	1.9878	1410	0.7500	7.7520×10^{-7}	1.1810	1241	0.7500	6.4589×10^{-7}
1024	1.9939	2670	0.7500	1.8319×10^{-6}	9.0631	2403	0.7500	1.0090×10^{-6}
2048	1.9969	5064	0.7500	3.7615×10^{-6}	70.2443	4653	0.7500	1.5635×10^{-6}
4096	1.9985	9656	0.7500	7.0949×10^{-6}	533.3514	9017	0.7500	2.3704×10^{-6}
								206.7515

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`:

Listing 4.22: `laplace-asor-p` — Parallel solution of Laplace’s equation with accelerated successive over relaxation.

```
0 // =====
```

And now we compare the serial and parallel implementations of the accelerated version in table 4.4. Can ADI outperform SOR?

Table 4.4: Grid size N_n , optimal over-relaxation parameter ω , number of iterations to convergence n_c , estimated $\bar{\phi}$, MAD and relative runtime t_r for the accelerated serial (`laplace-asor`) and accelerated parallel (`laplace-asor-p`) versions of the solution of Laplace's equation with SOR.

N_n	ω	accelerated serial				accelerated parallel			
		n_c	\bar{u}	MAD	t_r	n_c	$\bar{\phi}$	MAD	t_r
128	1.9521	336	0.7500	1.9599×10^{-7}	0.0049	321	0.7500	3.9300×10^{-7}	0.0120
256	1.9758	644	0.7500	3.7893×10^{-7}	0.0314	615	0.7500	7.7242×10^{-7}	0.0377
512	1.9878	1241	0.7500	6.4589×10^{-7}	0.3091	1185	0.7500	1.2148×10^{-6}	0.1912
1024	1.9939	2403	0.7500	1.0090×10^{-6}	2.3151	2289	0.7500	1.9388×10^{-6}	1.3176
2048	1.9969	4653	0.7500	1.5635×10^{-6}	21.1357	4421	0.7500	3.0534×10^{-6}	9.9622
4096	1.9985	9017	0.7500	2.3704×10^{-6}	206.7515	8571	0.7500	4.6177×10^{-6}	88.3975

5

Navier-Stokes

5.1 – A grade escalonada e os balanços materiais de volumes finitos

Desejamos resolver NS pelo “método da projeção”. As equações são

$$\rho \left[\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) \right] = -\nabla p + \mu \nabla^2 \mathbf{u}, \quad (5.1)$$

$$\nabla \cdot \mathbf{u} = 0. \quad (5.2)$$

Note que (5.1)–(5.2) já estão em uma forma “conservativa” (a divergência aparece explicitamente). Em (5.1) e (5.2), \mathbf{u} e p são respectivamente um campo vetorial e um campo escalar *analíticos*, ou seja: $\mathbf{u} = \mathbf{u}(x, y, z, t) = u(x, y, z, t)\mathbf{i} + v(x, y, z, t)\mathbf{j} + w(x, y, z, t)\mathbf{k}$ é uma função vetorial do \mathbb{R}^4 , e $p = p(x, y, z, t)$ é uma função escalar do \mathbb{R}^4 .

Nós agora reescrevemos todas as equações utilizando operadores *numéricos* $\delta/\delta t$ e $\bar{\nabla}$ (a serem detalhados na sequência) que aproximam $\partial/\partial t$ e ∇ :

$$\rho \left[\frac{\delta \mathbf{u}}{\delta t} + \bar{\nabla} \cdot (\mathbf{u}\mathbf{u}) \right] = -\bar{\nabla} p + \mu \bar{\nabla}^2 \mathbf{u}, \quad (5.3)$$

$$\bar{\nabla} \cdot \mathbf{u} = 0. \quad (5.4)$$

Em relação a (5.1)–(5.2), (5.3)–(5.4) contempla duas modificações. Em primeiro lugar, \mathbf{u} agora é um *array* multidimensional que inclui todos os valores de \mathbf{u} da solução discreta, na forma

$$u_{i,j+1/2}^n = u(n\Delta t, i\Delta x, (j+1/2)\Delta y), \quad i = 0, \dots, N_x, j = -1, \dots, N_y \quad (5.5)$$

$$v_{i,j+1/2}^n = v(n\Delta t, (i+1/2)\Delta x, j\Delta y), \quad i = -1, \dots, N_x, j = 0, \dots, N_y \quad (5.6)$$

$$\mathbf{u}_{i+1/2,j+1/2}^n = u_{i,j+1/2}^n \mathbf{i} + v_{i+1/2,j}^n \mathbf{j}. \quad (5.7)$$

Analogamente,

$$p_{i+1/2,j+1/2}^n = p(n\Delta t, (i+1/2)\Delta x, (j+1/2)\Delta y), \quad i = -1, \dots, N_x, j = -1, \dots, N_y. \quad (5.8)$$

A motivação para essa *grade escalonada* são os balanços materiais de massa, massa de um escalar, e quantidade de movimento, esboçados nas figuras 5.1, 5.2 e 5.3. Nas figuras, os quadrados cinza localizam escalares (tais como pressão, temperatura, etc.); os círculos pretos localizam a componente u da velocidade, e os círculos brancos localizam a componente v da velocidade. A grade é montada de tal modo que as componentes normais da velocidade em cada parede estão localizadas exatamente na parede (Prosperetti e Tryggvason, 2007, seção 2.4).

Em segundo lugar, os operadores analíticos $\partial/\partial t$ e ∇ foram substituídos pelos operadores numéricos $\delta/\delta t$ e $\bar{\nabla}$, que precisam ser definidos para cada um dos termos em (5.3)–(5.4). No entanto, com essas últimas ainda serão trabalhadas algebraicamente, as definições dos operadores numéricos serão

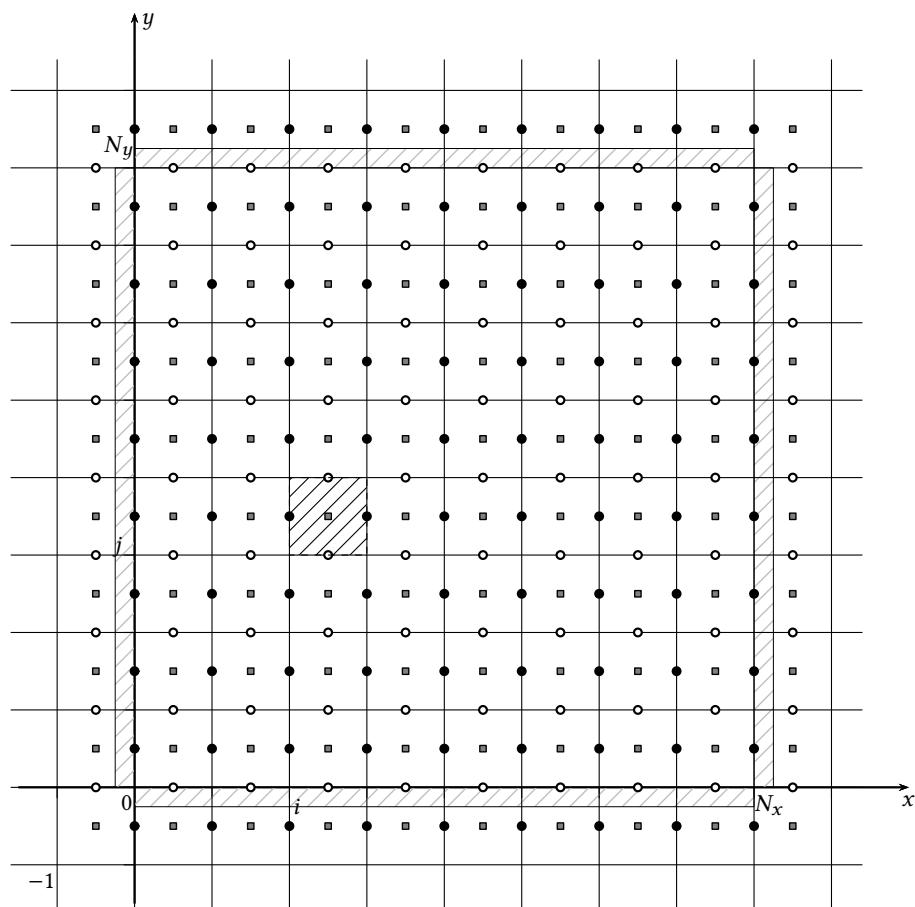


Figure 5.1: Balanços materiais de um escalar em uma grade escalonada

introduzidas apenas quando as discretizações correspondentes forem necessárias. Embora estejamos utilizando os mesmos símbolos (\mathbf{u} , u , v e p) em ambos os conjuntos de equações (5.1)–(5.2) e (5.3)–(5.4), nós eliminamos qualquer ambiguidade pelo uso dos operadores: de maneira inteiramente natural, quando os operadores forem analíticos, os símbolos indicam os campos analíticos contínuos; e quando os operadores forem numéricos, os símbolos indicam os campos discretizados.

Balanço de massa

Para um escoamento com $\rho = \text{constante}$, considere o volume de controle tracejado cujo ponto central é $(i + 1/2, j + 1/2)$. A equação de balanço de massa é (Fox et al., 2006)

$$\begin{aligned} 0 &= \frac{\partial}{\partial t} \int_{\mathcal{C}} \rho \, dV + \oint_{\mathcal{S}} \rho (\mathbf{n} \cdot \mathbf{u}) \, dA \\ 0 &= \oint_{\mathcal{S}} \rho (\mathbf{n} \cdot \mathbf{u}) \, dA \\ 0 &= -u_{i,j+1/2} \Delta y - v_{i+1/2,j} \Delta x + u_{i+1,j+1/2} \Delta y + v_{i+1/2,j+1} \Delta x \\ 0 &= \frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} + \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y}. \end{aligned} \quad (5.9)$$

Defina

$$\frac{\delta u}{\delta x} \equiv \frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x}, \quad (5.10)$$

$$\frac{\delta v}{\delta y} \equiv \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y}. \quad (5.11)$$

Claramente, (5.9) é o mesmo que, e completa a definição de, (5.4):

$$\bar{\nabla} \cdot \mathbf{u} = \frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} = \frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} + \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} = 0. \quad (5.12)$$

Balanço de massa de um escalar

A equação de balanço material para um escalar ϕ (novamente com $\rho = \text{constante}$, por simplicidade) é

$$\oint_{\mathcal{S}} \mathbf{n} \cdot \rho v_\phi \nabla \phi = \frac{\partial}{\partial t} \int_{\mathcal{C}} \rho \phi \, dV + \oint_{\mathcal{S}} \rho (\mathbf{n} \cdot [\phi \mathbf{u}]) \, dA. \quad (5.13)$$

É preciso converter o gradiente analítico $\nabla \phi$ em um termo numérico equivalente, do tipo

$$\bar{\nabla} \phi = \frac{\delta \phi}{\delta x} \mathbf{i} + \frac{\delta \phi}{\delta y} \mathbf{j}.$$

Expandindo numericamente e calculando os produtos escalares em cada uma das 4 faces do retângulo tracejado,

$$\begin{aligned} &v_\phi \left[\frac{\delta \phi}{\delta x} \Big|_{i+1,j+1/2} - \frac{\delta \phi}{\delta x} \Big|_{i,j+1/2} \right] \Delta y + v_\phi \left[\frac{\delta \phi}{\delta y} \Big|_{i+1/2,j+1} - \frac{\delta \phi}{\delta y} \Big|_{i+1/2,j} \right] \Delta x \\ &= \frac{\delta \phi}{\delta t} \Delta x \Delta y - u_{i,j+1/2} \tilde{\phi}_{i+1/2,j+1/2} \Delta y - v_{i+1/2,j} \tilde{\phi}_{i+1/2,j+1/2} \Delta x + u_{i+1,j+1/2} \tilde{\phi}_{i+1,j+1/2} \Delta y + v_{i+1/2,j+1} \tilde{\phi}_{i+1/2,j+1} \Delta x. \end{aligned}$$

os tis sobre os ϕ s serão explicados em breve. Dividindo por $\Delta x \Delta y$:

$$\begin{aligned} \frac{\delta \phi}{\delta t} + \frac{u_{i+1,j+1/2} \tilde{\phi}_{i+1,j+1/2} - u_{i,j+1/2} \tilde{\phi}_{i,j+1/2}}{\Delta x} + \frac{v_{i+1/2,j+1} \tilde{\phi}_{i+1/2,j+1} - v_{i+1/2,j} \tilde{\phi}_{i+1/2,j}}{\Delta y} = \\ \frac{v_\phi}{\Delta x} \left[\frac{\delta \phi}{\delta x} \Big|_{i+1,j+1/2} - \frac{\delta \phi}{\delta x} \Big|_{i,j+1/2} \right] + \frac{v_\phi}{\Delta y} \left[\frac{\delta \phi}{\delta y} \Big|_{i+1/2,j} - \frac{\delta \phi}{\delta y} \Big|_{i,j+1/2} \right]. \end{aligned} \quad (5.14)$$

É de se esperar que os termos entre colchetes do lado direito levem à estimativa usual da derivada segunda numérica. De fato,

$$\begin{aligned} \frac{1}{\Delta x} \left[\frac{\delta \phi}{\delta x} \Big|_{i+1,j+1/2} - \frac{\delta \phi}{\delta x} \Big|_{i,j+1/2} \right] &= \frac{1}{\Delta x} \left\{ \frac{\phi_{i+3/2,j+1/2} - \phi_{i+1/2,j+1/2}}{\Delta x} - \frac{\phi_{i+1/2,j+1/2} - \phi_{i-1/2,j+1/2}}{\Delta x} \right\} \\ &= \frac{\phi_{i+3/2,j+1/2} - 2\phi_{i+1/2,j+1/2} + \phi_{i-1/2,j+1/2}}{\Delta x^2}. \end{aligned}$$

Os termos advectivos em (5.14) são

$$\frac{\delta(u\phi)}{\delta x} = \frac{u_{i+1,j+1/2}\tilde{\phi}_{i+1,j+1/2} - u_{i,j+1/2}\tilde{\phi}_{i,j+1/2}}{\Delta x}, \quad (5.15)$$

$$\frac{\delta(v\phi)}{\delta y} = \frac{v_{i+1/2,j+1}\tilde{\phi}_{i+1/2,j+1} - v_{i+1/2,j}\tilde{\phi}_{i+1/2,j}}{\Delta y}. \quad (5.16)$$

Veja novamente a figura 5.1: os valores $\tilde{\phi}_{i+1,j+1/2}$, $\tilde{\phi}_{i,j+1/2}$, $\tilde{\phi}_{i+1/2,j+1}$ e $\tilde{\phi}_{i+1/2,j}$ não estão na grade de ϕ (quadrados cinzas), e precisam ser obtidos por interpolação. Este fato é indicado pelo til sobre eles. Isso na verdade nos deixa livres para escolher a interpolação mais adequada. Algumas opções são:

Interpolação centrada :

$$\tilde{\phi}_{i+1,j+1/2} = \frac{1}{2}(\phi_{i+1/2,j+1/2} + \phi_{i+3/2,j+1/2}). \quad (5.17)$$

Upwind :

$$\tilde{\phi}_{i+1,j+1/2} = \begin{cases} \phi_{i+1/2}, & u_{i+1,j+1/2} > 0, \\ \phi_{i+3/2}, & u_{i+1,j+1/2} < 0. \end{cases} \quad (5.18)$$

QUICK :

$$\tilde{\phi}_{i+1/2,j} = \begin{cases} \frac{1}{8}(3\phi_{i+3/2} + 6\phi_{i+1/2} - \phi_{i-1/2}), & u_{i+1,j+1/2} > 0, \\ \frac{1}{8}(3\phi_{i+1/2} + 6\phi_{i+3/2} - \phi_{i+5/2}), & u_{i+1,j+1/2} < 0. \end{cases} \quad (5.19)$$

Balanço de quantidade de movimento em x

Nós agora precisamos de um volume de controle centrado nos pontos onde localizamos a velocidade u (círculos pretos). O volume de controle em questão está mostrado (hachuriado) na figura 5.2.

A equação de balanço material para u (novamente com $\rho = \text{constante}$, por simplicidade) é

$$\oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{i}) dA + \int_{\mathcal{C}} \rho g_x dV = \frac{\partial}{\partial t} \int_{\mathcal{C}} \rho u dV + \oint_{\mathcal{S}} \rho (\mathbf{n} \cdot [\mathbf{u} \mathbf{u}]) dA. \quad (5.20)$$

O termo mais complicado é o vetor de tensões; ele precisa ser calculado para cada face do volume de controle de controle. Começamos escrevendo por extenso o tensor de tensões T em 2D com operadores numéricicos:

$$\mathbf{T} = -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta v} \right) \end{bmatrix}. \quad (5.21)$$

Para a face $i + 1/2$ da figura (5.2), $\mathbf{n} = \mathbf{i}$ em (5.20) e

$$\begin{aligned} [\mathbf{i} \cdot \mathbf{T}] \cdot \mathbf{i} &= [1 \ 0] \cdot \left\{ -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta v} \right) \end{bmatrix} \right\} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= [1 \ 0] \cdot \left\{ -p \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \frac{\delta u}{\delta x} \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) \end{bmatrix} \right\} \end{aligned}$$

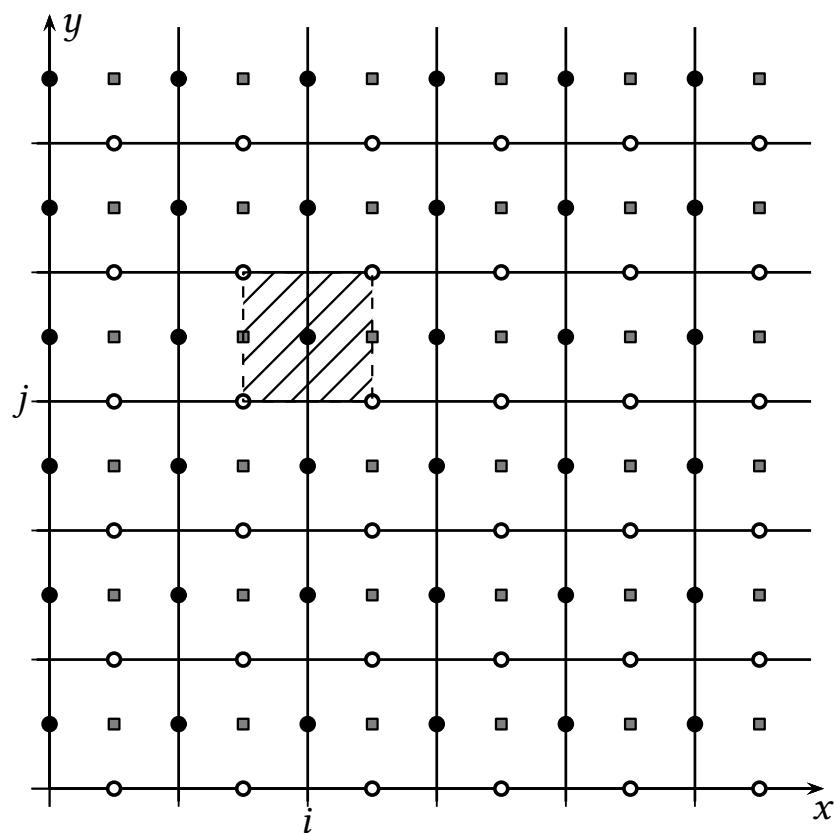


Figure 5.2: Balanço de quantidade de movimento em x em uma grade escalonada

$$\begin{aligned}
&= \left[-p + 2\mu \frac{\delta u}{\delta x} \right]_{i+1/2, j+1/2} \\
&= -p_{i+1/2, j+1/2} + 2\mu \frac{u_{i+1, j+1/2} - u_{i, j+1/2}}{\Delta x}.
\end{aligned}$$

Para a face $i - 1/2$ da figura (5.2), $\mathbf{n} = -\mathbf{i}$ em (5.20) e

$$\begin{aligned}
[-\mathbf{i} \cdot \mathbf{T}] \cdot \mathbf{i} &= [-1 \ 0] \cdot \left\{ -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta v} \right) \end{bmatrix} \right\} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\
&= [-1 \ 0] \cdot \left\{ -p \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \frac{\delta u}{\delta x} \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) \end{bmatrix} \right\} \\
&= - \left[-p + 2\mu \frac{\delta u}{\delta x} \right]_{i-1/2, j+1/2} \\
&= - \left[-p_{i-1/2, j+1/2} + 2\mu \frac{u_{i, j+1/2} - u_{i-1, j+1/2}}{\Delta x} \right].
\end{aligned}$$

Para a face $j + 1$ da figura (5.2), $\mathbf{n} = +\mathbf{j}$ em (5.20) e

$$\begin{aligned}
[j \cdot \mathbf{T}] \cdot \mathbf{i} &= [0 \ 1] \cdot \left\{ -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta v} \right) \end{bmatrix} \right\} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\
&= [0 \ 1] \cdot \left\{ -p \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \frac{\delta u}{\delta x} \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) \end{bmatrix} \right\} \\
&= \mu \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right)_{i, j+1} \\
&= \mu \left(\frac{v_{i+1/2, j+1} - v_{i-1/2, j+1}}{\Delta x} + \frac{u_{i, j+3/2} - u_{i, j+1/2}}{\Delta y} \right)
\end{aligned}$$

Para a face j da figura (5.2), $\mathbf{n} = -\mathbf{j}$ em (5.20) e

$$\begin{aligned}
[j \cdot \mathbf{T}] \cdot \mathbf{i} &= [0 \ -1] \cdot \left\{ -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta v} \right) \end{bmatrix} \right\} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\
&= [0 \ -1] \cdot \left\{ -p \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \frac{\delta u}{\delta x} \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) \end{bmatrix} \right\} \\
&= -\mu \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right)_{i+1/2, j} \\
&= -\mu \left(\frac{v_{i+1/2, j} - v_{i-1/2, j}}{\Delta x} + \frac{u_{i, j+1/2} - u_{i, j-1/2}}{\Delta y} \right)
\end{aligned}$$

Podemos agora exprimir a integral de superfície envolvendo \mathbf{T} em (5.20) como

$$\begin{aligned}
\oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{i}) \, dA &= \left[-p_{i+1/2, j+1/2} + 2\mu \frac{u_{i+1, j+1/2} - u_{i, j+1/2}}{\Delta x} \right] \Delta y \\
&\quad - \left[-p_{i-1/2, j+1/2} + 2\mu \frac{u_{i, j+1/2} - u_{i-1, j+1/2}}{\Delta x} \right] \Delta y \\
&\quad + \mu \left(\frac{v_{i+1/2, j+1} - v_{i-1/2, j+1}}{\Delta x} + \frac{u_{i, j+3/2} - u_{i, j+1/2}}{\Delta y} \right) \Delta x \\
&\quad - \mu \left(\frac{v_{i+1/2, j} - v_{i-1/2, j}}{\Delta x} + \frac{u_{i, j+1/2} - u_{i, j-1/2}}{\Delta y} \right) \Delta x;
\end{aligned}$$

e rearranjamos:

$$\begin{aligned} \oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{i}) \, dA &= \left[-p_{i+1/2,j+1/2} + \mu \frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} + \mu \frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} \right] \Delta y \\ &\quad - \left[-p_{i-1/2,j+1/2} + \mu \frac{u_{i,j+1/2} - u_{i-1,j+1/2}}{\Delta x} + \mu \frac{u_{i,j+1/2} - u_{i-1,j+1/2}}{\Delta x} \right] \Delta y \\ &\quad + \mu \left(\frac{v_{i+1/2,j+1} - v_{i-1/2,j+1}}{\Delta x} + \frac{u_{i,j+3/2} - u_{i,j+1/2}}{\Delta y} \right) \Delta x \\ &\quad - \mu \left(\frac{v_{i+1/2,j} - v_{i-1/2,j}}{\Delta x} + \frac{u_{i,j+1/2} - u_{i,j-1/2}}{\Delta y} \right) \Delta x; \\ \frac{1}{\Delta x \Delta y} \oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{i}) \, dA &= -\frac{p_{i+1/2,j+1/2} - p_{i-1/2,j+1/2}}{\Delta x} + \mu \frac{u_{i+1,j+1/2} - 2u_{i,j+1/2} + u_{i-1,j+1/2}}{\Delta x^2} \\ &\quad + \frac{\mu}{\Delta x} \left[\frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} - \frac{u_{i,j+1/2} - u_{i-1,j+1/2}}{\Delta x} \right] \\ &\quad + \mu \frac{u_{i,j+3/2} - 2u_{i,j+1/2} + u_{i,j-1/2}}{\Delta y^2} \\ &\quad + \frac{\mu}{\Delta x} \left[\frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} - \frac{v_{i-1/2,j+1} - v_{i-1/2,j}}{\Delta y} \right] \\ &= -\frac{p_{i+1/2,j+1/2} - p_{i-1/2,j+1/2}}{\Delta x} + \mu \frac{u_{i+1,j+1/2} - 2u_{i,j+1/2} + u_{i-1,j+1/2}}{\Delta x^2} \\ &\quad + \mu \frac{u_{i,j+3/2} - 2u_{i,j+1/2} - u_{i,j-1/2}}{\Delta y^2} \\ &\quad + \frac{\mu}{\Delta x} \left[\frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} + \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} \right] \\ &\quad - \frac{\mu}{\Delta x} \left[\frac{u_{i,j+1/2} - u_{i-1,j+1/2}}{\Delta x} + \frac{v_{i-1/2,j+1} - v_{i-1/2,j}}{\Delta y} \right]. \end{aligned}$$

As duas últimas linhas acima são identicamente nulas em virtude da continuidade (5.12):

$$\begin{aligned} \left[\frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} \right]_{i+1/2,j+1/2} &= \frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} + \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} = 0, \\ \left[\frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} \right]_{i-1/2,j+1/2} &= \frac{u_{i,j+1/2} - u_{i-1,j+1/2}}{\Delta x} + \frac{v_{i-1/2,j+1} - v_{i-1/2,j}}{\Delta y} = 0. \end{aligned}$$

Portanto, o balanço de tensões na direção x fica

$$\begin{aligned} \frac{1}{\Delta x \Delta y} \oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{i}) \, dA &= \\ &- \frac{p_{i+1/2,j+1/2} - p_{i-1/2,j+1/2}}{\Delta x} + \mu \left[\frac{u_{i+1,j+1/2} - 2u_{i,j+1/2} + u_{i-1,j+1/2}}{\Delta x^2} + \frac{u_{i,j+3/2} - 2u_{i,j+1/2} + u_{i,j-1/2}}{\Delta y^2} \right] = \\ &\quad - \frac{\delta p}{\delta x} \Big|_{i,j+1/2} + \mu \left[\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} \right]_{i,j+1/2}. \quad (5.22) \end{aligned}$$

O 2º termo do lado esquerdo de (5.20) é trivial e pode ser escrito como

$$\frac{1}{\Delta x \Delta y} \int_{\mathcal{C}} \rho g_x \, dV = \rho g_x. \quad (5.23)$$

Passamos portanto, agora, ao lado direito de (5.20). O termo transiente também é trivial:

$$\frac{\partial}{\partial t} \int_{\mathcal{C}} \rho u \, dV = \rho \Delta x \Delta y \frac{\delta u}{\delta t};$$

$$\frac{1}{\Delta x \Delta y} \left[\frac{\partial}{\partial t} \int_{\mathcal{C}} \rho u \, dV \right] = \rho \frac{\delta u}{\delta t} \Big|_{i,j+1/2} = \rho \frac{u_{i,j+1/2}^{n+1} - u_{i,j+1/2}^n}{\Delta t}. \quad (5.24)$$

Finalmente, a integral de superfície do lado direito de (5.20) precisa ser calculada para cada uma das 4 faces tomando-se sucessivamente $\mathbf{n} = +\mathbf{i}$, $\mathbf{n} = -\mathbf{i}$, $\mathbf{n} = +\mathbf{j}$ e $\mathbf{n} = -\mathbf{j}$ para as faces $i+1/2$, $i-1/2$, $j+1$ e j :

$$\rho(\mathbf{n} \cdot [\mathbf{u}\mathbf{u}]) = \rho(\mathbf{i} \cdot [\mathbf{u}\mathbf{u}]) = \rho [\tilde{u}^2]_{i+1/2,j+1/2}, \quad (5.25)$$

$$\rho(\mathbf{n} \cdot [\mathbf{u}\mathbf{u}]) = \rho(-\mathbf{i} \cdot [\mathbf{u}\mathbf{u}]) = -\rho [\tilde{u}^2]_{i-1/2,j+1/2}, \quad (5.26)$$

$$\rho(\mathbf{n} \cdot [\mathbf{u}\mathbf{u}]) = \rho(\mathbf{j} \cdot [\mathbf{u}\mathbf{u}]) = \rho [\tilde{u}\tilde{v}]_{i,j+1}, \quad (5.27)$$

$$\rho(\mathbf{n} \cdot [\mathbf{u}\mathbf{u}]) = \rho(-\mathbf{j} \cdot [\mathbf{u}\mathbf{u}]) = -\rho [\tilde{u}\tilde{v}]_{i,j}. \quad (5.28)$$

Note que nem u nem v estão localizados em nenhuma das 4 faces do volume de controle da figura 5.2 e, da mesma forma que aconteceu com ϕ no balanço material (5.14), precisam ser interpolados. A integral de superfície correspondente é

$$\oint_{\mathcal{S}} \rho(\mathbf{n} \cdot [\mathbf{u}\mathbf{u}]) \, dA = \rho \left\{ \left[\tilde{u}_{i+1/2,j+1/2}^2 - \tilde{u}_{i-1/2,j+1/2}^2 \right] \Delta y + \left[\tilde{u}_{i,j+1} \tilde{v}_{i,j+1} - \tilde{u}_{i,j} \tilde{v}_{i,j} \right] \Delta x \right\},$$

ou

$$\begin{aligned} \frac{1}{\Delta x \Delta y} \oint_{\mathcal{S}} \rho(\mathbf{n} \cdot [\mathbf{u}\mathbf{u}]) \, dA &= \rho \left\{ \frac{\tilde{u}_{i+1/2,j+1/2}^2 - \tilde{u}_{i-1/2,j+1/2}^2}{\Delta x} + \frac{\tilde{u}_{i,j+1} \tilde{v}_{i,j+1} - \tilde{u}_{i,j} \tilde{v}_{i,j}}{\Delta y} \right\} \\ &= \rho \left[\frac{\delta \tilde{u}^2}{\delta x} + \frac{\delta \tilde{u} \tilde{v}}{\delta y} \right]_{i,j+1/2}. \end{aligned} \quad (5.29)$$

As mesmas opções de interpolação descritas para $\tilde{\phi}$ em (5.17)–(5.19) valem também para \tilde{u} e \tilde{v} .

A obtenção da equação de quantidade de movimento na direção x para volumes finitos está agora completa e pode ser escrita de forma compacta, utilizando os operadores diferenciais numéricos, como

$$\frac{\delta u}{\delta t} \Big|_{i,j+1/2} + \left[\frac{\delta \tilde{u}^2}{\delta x} + \frac{\delta \tilde{u} \tilde{v}}{\delta y} \right]_{i,j+1/2} = g_x - \frac{1}{\rho} \frac{\delta p}{\delta x} \Big|_{i,j+1/2} + \nu \left[\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} \right]_{i,j+1/2}. \quad (5.30)$$

Para a direção y , nós usamos o volume de controle da figura 5.3. A equação de diferenças é

$$\frac{\delta v}{\delta t} \Big|_{i+1/2,j} + \left[\frac{\delta \tilde{u} \tilde{v}}{\delta x} + \frac{\delta \tilde{v}^2}{\delta y} \right]_{i+1/2,j} = g_y - \frac{1}{\rho} \frac{\delta p}{\delta y} \Big|_{i+1/2,j} + \nu \left[\frac{\delta^2 v}{\delta x^2} + \frac{\delta^2 v}{\delta y^2} \right]_{i+1/2,j}. \quad (5.31)$$

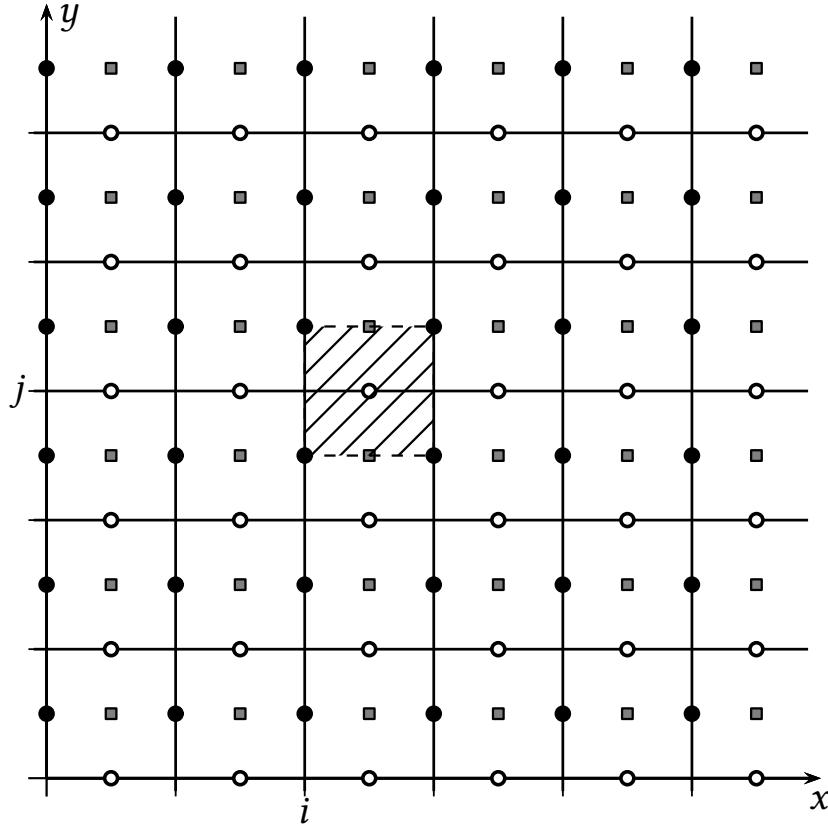


Figure 5.3: Balanço de quantidade de movimento em y em uma grade escalonada

Balanço de quantidade de movimento em y

Nós agora precisamos de um volume de controle centrado nos pontos onde localizamos a velocidade u (círculos pretos). O volume de controle em questão está mostrado (hachurado) na figura 5.3.

A equação de balanço material para v (novamente com $\rho = \text{constante}$, por simplicidade) é

$$\oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{j}) \, dA + \int_{\mathcal{C}} \rho g_y \, dV = \frac{\partial}{\partial t} \int_{\mathcal{C}} \rho v \, dV + \oint_{\mathcal{S}} \rho (\mathbf{n} \cdot [v \mathbf{u}]) \, dA. \quad (5.32)$$

O termo mais complicado é o vetor de tensões; ele precisa ser calculado para cada face do volume de controle de controle. Começamos escrevendo por extenso o tensor de tensões T em 2D com operadores numéricicos:

$$\mathbf{T} = -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta y} \right) \end{bmatrix}. \quad (5.33)$$

Para a face $j + 1/2$ da figura (5.3), $\mathbf{n} = \mathbf{j}$ em (5.32) e

$$\begin{aligned} [\mathbf{j} \cdot \mathbf{T}] \cdot \mathbf{j} &= [0 \ 1] \cdot \left\{ -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta y} \right) \end{bmatrix} \right\} \cdot [0 \ 1] \\ &= [0 \ 1] \cdot \left\{ -p \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ 2 \frac{\delta v}{\delta y} \end{bmatrix} \right\} \end{aligned}$$

$$\begin{aligned}
&= \left[-p + 2\mu \frac{\delta v}{\delta y} \right]_{i/2,j+1/2} \\
&= -p_{i+1/2,j+1/2} + 2\mu \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y}.
\end{aligned}$$

Para a face $j - 1/2$ da figura (5.2), $\mathbf{n} = -\mathbf{j}$ em (5.32) e

$$\begin{aligned}
[-\mathbf{j} \cdot \mathbf{T}] \cdot \mathbf{j} &= [0 \quad -1] \cdot \left\{ -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta y} \right) \end{bmatrix} \right\} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\
&= [0 \quad -1] \cdot \left\{ -p \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \mu \left[\left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) 2 \frac{\delta v}{\delta y} \right] \right\} \\
&= - \left[-p + 2\mu \frac{\delta v}{\delta y} \right]_{i+1/2,j-1/2} \\
&= - \left[-p_{i+1/2,j-1/2} + 2\mu \frac{v_{i+1/2,j} - v_{i+1/2,j-1}}{\Delta y} \right].
\end{aligned}$$

Para a face $i + 1$ da figura (5.3), $\mathbf{n} = +\mathbf{i}$ em (5.32) e

$$\begin{aligned}
[\mathbf{i} \cdot \mathbf{T}] \cdot \mathbf{j} &= [1 \quad 0] \cdot \left\{ -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta y} \right) \end{bmatrix} \right\} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\
&= [1 \quad 0] \cdot \left\{ -p \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \mu \left[\left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) 2 \frac{\delta v}{\delta y} \right] \right\} \\
&= \mu \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right)_{i+1,j} \\
&= \mu \left(\frac{u_{i+1,j+1/2} - u_{i+1,j-1/2}}{\Delta y} + \frac{v_{i+3/2,j} - v_{i+1/2,j}}{\Delta x} \right)
\end{aligned}$$

Para a face i da figura (5.3), $\mathbf{n} = -\mathbf{i}$ em (5.32) e

$$\begin{aligned}
[-\mathbf{i} \cdot \mathbf{T}] \cdot \mathbf{j} &= [-1 \quad 0] \cdot \left\{ -p \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \mu \begin{bmatrix} \left(\frac{\delta u}{\delta x} + \frac{\delta u}{\delta x} \right) & \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) \\ \left(\frac{\delta v}{\delta x} + \frac{\delta u}{\delta y} \right) & \left(\frac{\delta v}{\delta y} + \frac{\delta v}{\delta y} \right) \end{bmatrix} \right\} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\
&= [-1 \quad 0] \cdot \left\{ -p \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \mu \left[\left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right) 2 \frac{\delta v}{\delta y} \right] \right\} \\
&= -\mu \left(\frac{\delta u}{\delta y} + \frac{\delta v}{\delta x} \right)_{i,j} \\
&= -\mu \left(\frac{u_{i,j+1/2} - u_{i,j-1/2}}{\Delta y} + \frac{v_{i+1/2,j} - v_{i-1/2,j}}{\Delta x} \right)
\end{aligned}$$

Podemos agora exprimir a integral de superfície envolvendo \mathbf{T} em (5.32) como

$$\begin{aligned}
\oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{j}) \, dA &= \left[-p_{i+1/2,j+1/2} + 2\mu \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} \right] \Delta x \\
&\quad - \left[-p_{i+1/2,j-1/2} + 2\mu \frac{v_{i+1/2,j} - v_{i+1/2,j-1}}{\Delta y} \right] \Delta x \\
&\quad + \mu \left(\frac{u_{i+1,j+1/2} - u_{i+1,j-1/2}}{\Delta y} + \frac{v_{i+3/2,j} - v_{i+1/2,j}}{\Delta x} \right) \Delta y \\
&\quad - \mu \left(\frac{u_{i,j+1/2} - u_{i,j-1/2}}{\Delta y} + \frac{v_{i+1/2,j} - v_{i-1/2,j}}{\Delta x} \right) \Delta y;
\end{aligned}$$

e rearranjamos:

$$\begin{aligned} \oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{j}) \, dA &= \left[-p_{i+1/2,j+1/2} + \mu \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} + \mu \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} \right] \Delta x \\ &\quad - \left[-p_{i+1/2,j-1/2} + \mu \frac{v_{i+1/2,j} - v_{i+1/2,j-1}}{\Delta y} + \mu \frac{v_{i+1/2,j} - v_{i+1/2,j-1}}{\Delta y} \right] \Delta x \\ &\quad + \mu \left(\frac{u_{i+1,j+1/2} - u_{i+1,j-1/2}}{\Delta y} + \frac{v_{i+3/2,j} - v_{i+1/2,j}}{\Delta x} \right) \Delta y \\ &\quad - \mu \left(\frac{u_{i,j+1/2} - u_{i,j-1/2}}{\Delta y} + \frac{v_{i+1/2,j} - v_{i-1/2,j}}{\Delta x} \right) \Delta y; \\ \frac{1}{\Delta x \Delta y} \oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{j}) \, dA &= -\frac{p_{i+1/2,j+1/2} - p_{i+1/2,j-1/2}}{\Delta y} + \mu \frac{v_{i+1/2,j+1} - 2v_{i+1/2,j} + v_{i+1/2,j-1}}{\Delta y^2} \\ &\quad + \frac{\mu}{\Delta y} \left[\frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} - \frac{v_{i+1/2,j} - v_{i+1/2,j-1}}{\Delta y} \right] \\ &\quad + \mu \frac{v_{i+3/2,j} - 2v_{i+1/2,j} + v_{i-1/2,j}}{\Delta x^2} \\ &\quad + \frac{\mu}{\Delta y} \left[\frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} - \frac{u_{i+1,j-1/2} - u_{i,j-1/2}}{\Delta x} \right] \\ &= -\frac{p_{i+1/2,j+1/2} - p_{i+1/2,j-1/2}}{\Delta y} + \mu \frac{v_{i+1/2,j+1} - 2v_{i+1/2,j} + v_{i+1/2,j-1}}{\Delta y^2} \\ &\quad + \mu \frac{v_{i+3/2,j} - 2v_{i+1/2,j} + v_{i-1/2,j}}{\Delta x^2} \\ &\quad + \frac{\mu}{\Delta y} \left[\frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} + \frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} \right] \\ &\quad - \frac{\mu}{\Delta y} \left[\frac{v_{i+1/2,j} - v_{i+1/2,j-1}}{\Delta y} + \frac{u_{i+1,j-1/2} - u_{i,j-1/2}}{\Delta x} \right]. \end{aligned}$$

As duas últimas linhas acima são identicamente nulas em virtude da equação da continuidade (5.12):

$$\begin{aligned} \left[\frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} \right]_{i+1/2,j+1/2} &= \frac{u_{i+1,j+1/2} - u_{i,j+1/2}}{\Delta x} + \frac{v_{i+1/2,j+1} - v_{i+1/2,j}}{\Delta y} = 0, \\ \left[\frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} \right]_{i+1/2,j-1/2} &= \frac{u_{i+1,j-1/2} - u_{i,j-1/2}}{\Delta x} + \frac{v_{i+1/2,j} - v_{i+1/2,j-1}}{\Delta y} = 0. \end{aligned}$$

Portanto, o balanço de tensões na direção y fica

$$\begin{aligned} \frac{1}{\Delta x \Delta y} \oint_{\mathcal{S}} ([\mathbf{n} \cdot \mathbf{T}] \cdot \mathbf{j}) \, dA &= \\ &- \frac{p_{i+1/2,j+1/2} - p_{i+1/2,j-1/2}}{\Delta y} + \mu \left[\frac{v_{i+1/2,j+1} - 2v_{i+1/2,j} + v_{i+1/2,j-1}}{\Delta y^2} + \frac{v_{i+3/2,j} - 2v_{i+1/2,j} + u_{i-1/2,j}}{\Delta x^2} \right] = \\ &\quad - \left. \frac{\delta p}{\delta y} \right|_{i+1/2,j} + \mu \left[\frac{\delta^2 v}{\delta x^2} + \frac{\delta^2 v}{\delta y^2} \right]_{i+1/2,j}. \quad (5.34) \end{aligned}$$

O 2º termo do lado esquerdo de (5.32) é trivial e pode ser escrito como

$$\frac{1}{\Delta x \Delta y} \int_{\mathcal{C}} \rho g_y \, dV = \rho g_y. \quad (5.35)$$

Passamos portanto, agora, ao lado direito de (5.32). O termo transiente também é trivial:

$$\frac{\partial}{\partial t} \int_{\mathcal{C}} \rho v \, dV = \rho \Delta x \Delta y \frac{\delta v}{\delta t};$$

$$\frac{1}{\Delta x \Delta y} \left[\frac{\partial}{\partial t} \int_{\mathcal{C}} \rho v \, dV \right] = \rho \frac{\delta u}{\delta t} \Big|_{i+1/2,j} = \rho \frac{v_{i+1/2,j}^{n+1} - v_{i+1/2,j}^n}{\Delta t}. \quad (5.36)$$

Finalmente, a integral de superfície do lado direito de (5.20) precisa ser calculada para cada uma das 4 faces tomando-se sucessivamente $\mathbf{n} = +\mathbf{i}$, $\mathbf{n} = -\mathbf{i}$, $\mathbf{n} = +\mathbf{j}$ e $\mathbf{n} = -\mathbf{j}$ para as faces $i+1$, i , $j+1/2$ e $j-1/2$:

$$\rho(\mathbf{n} \cdot [\mathbf{v}\mathbf{u}]) = \rho(\mathbf{i} \cdot [\mathbf{v}\mathbf{u}]) = \rho [\widetilde{uv}]_{i+1,j}, \quad (5.37)$$

$$\rho(\mathbf{n} \cdot [\mathbf{v}\mathbf{u}]) = \rho(-\mathbf{i} \cdot [\mathbf{v}\mathbf{u}]) = -\rho [\widetilde{uv}]_{i,j}, \quad (5.38)$$

$$\rho(\mathbf{n} \cdot [\mathbf{v}\mathbf{u}]) = \rho(\mathbf{j} \cdot [\mathbf{v}\mathbf{u}]) = \rho [\widetilde{v}^2]_{i+1/2,j+1/2}, \quad (5.39)$$

$$\rho(\mathbf{n} \cdot [\mathbf{v}\mathbf{u}]) = \rho(-\mathbf{j} \cdot [\mathbf{v}\mathbf{u}]) = -\rho [\widetilde{v}^2]_{i+1/2,j-1/2}. \quad (5.40)$$

Note que nem u nem v estão localizados em nenhuma das 4 faces do volume de controle da figura 5.2 e, da mesma forma que aconteceu com ϕ no balanço material (5.14), precisam ser interpolados. A integral de superfície correspondente é

$$\oint_{\mathcal{S}} \rho(\mathbf{n} \cdot [\mathbf{u}\mathbf{u}]) \, dA = \rho \left\{ \left[\widetilde{v}_{i+1/2,j+1/2}^2 - \widetilde{u}_{i+1/2,j-1/2}^2 \right] \Delta x + \left[\widetilde{u}_{i+1,j} \widetilde{v}_{i+1,j} - \widetilde{u}_{i,j} \widetilde{v}_{i,j} \right] \Delta y \right\},$$

ou

$$\begin{aligned} \frac{1}{\Delta x \Delta y} \oint_{\mathcal{S}} \rho(\mathbf{n} \cdot [\mathbf{u}\mathbf{u}]) \, dA &= \rho \left\{ \frac{\widetilde{v}_{i+1/2,j+1/2}^2 - \widetilde{u}_{i+1/2,j-1/2}^2}{\Delta y} + \frac{\widetilde{u}_{i+1,j} \widetilde{v}_{i+1,j} - \widetilde{u}_{i,j} \widetilde{v}_{i,j}}{\Delta x} \right\} \\ &= \rho \left[\frac{\delta \widetilde{v}^2}{\delta y} + \frac{\delta \widetilde{u} \widetilde{v}}{\delta x} \right]_{i+1/2,j}. \end{aligned} \quad (5.41)$$

As mesmas opções de interpolação descritas para $\widetilde{\phi}$ em (5.17)–(5.19) valem também para \widetilde{u} e \widetilde{v} .

A obtenção da equação de quantidade de movimento para volumes finitos está agora completa e pode ser escrita de forma compacta, utilizando os operadores diferenciais numéricos, como

$$\frac{\delta u}{\delta t} \Big|_{i,j+1/2} + \left[\frac{\delta \widetilde{u}^2}{\delta x} + \frac{\delta \widetilde{u} \widetilde{v}}{\delta y} \right]_{i,j+1/2} = g_x - \frac{1}{\rho} \frac{\delta p}{\delta x} \Big|_{i,j+1/2} + \nu \left[\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} \right]_{i,j+1/2}. \quad (5.42)$$

Para a direção y , nós usamos o volume de controle da figura 5.3. A equação de diferenças é

$$\frac{\delta v}{\delta t} \Big|_{i+1/2,j} + \left[\frac{\delta \widetilde{u} \widetilde{v}}{\delta x} + \frac{\delta \widetilde{v}^2}{\delta y} \right]_{i+1/2,j} = g_y - \frac{1}{\rho} \frac{\delta p}{\delta y} \Big|_{i+1/2,j} + \nu \left[\frac{\delta^2 v}{\delta x^2} + \frac{\delta^2 v}{\delta y^2} \right]_{i+1/2,j}. \quad (5.43)$$

5.2 – Condições de contorno para u

At $x = 0$ and at $x = L_x$, u is the normal component, and it is zero:

$$\begin{aligned} u_{0,j} &= 0, & j &= -1/2, \dots, N_y + 1/2, \\ u_{N_x,j} &= 0, & j &= -1/2, \dots, N_y + 1/2. \end{aligned}$$

At $y = 0$, we must have

$$(u_{i,1/2} + u_{i,-1/2})/2 = 0.$$

Finally, at $y = L_y$, we must have

$$(u_{i,N_y+1/2} + u_{i,N_y-1/2})/2 = U_0.$$

5.3 – Progressão no tempo

Inicialmente fazemos

$$\frac{\delta \mathbf{u}}{\delta t} \equiv \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t}. \quad (5.44)$$

Nós vamos “quebrar” (*split*) o operador introduzindo um campo auxiliar de velocidade \mathbf{u}^* em (5.3):

$$\left[\frac{\mathbf{u}^{n+1} - \mathbf{u}^*}{\Delta t} + \frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} \right] = -\bar{\nabla} \cdot (\mathbf{u}\mathbf{u}) - \frac{1}{\rho} \bar{\nabla} p + \nu \bar{\nabla}^2 \mathbf{u}, \quad (5.45)$$

de tal forma que

$$\frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} = -\bar{\nabla} \cdot (\mathbf{u}\mathbf{u}) + \nu \bar{\nabla}^2 \mathbf{u}, \quad (5.46)$$

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^*}{\Delta t} = -\frac{1}{\rho} \bar{\nabla} p. \quad (5.47)$$

Note que (5.46)+(5.47) recupera (5.45), mas ainda falta impor a condição de incompressibilidade (5.4). Isso pode ser feito tomando-se a divergência numérica de (5.47), e aplicando (5.4) para \mathbf{u}^{n+1} :

$$\begin{aligned} \bar{\nabla} \cdot \frac{\mathbf{u}^{n+1} - \mathbf{u}^*}{\Delta t} &= -\frac{1}{\rho} \bar{\nabla} \cdot \bar{\nabla} p, \\ \frac{\rho}{\Delta t} \bar{\nabla} \cdot \mathbf{u}^* &= \bar{\nabla}^2 p, \end{aligned} \quad (5.48)$$

Isso é tudo de que precisamos por enquanto. A sequência de cálculo é a seguinte: partindo de \mathbf{u}^n calculamos \mathbf{u}^* com (5.46) utilizando um esquema (por enquanto) explícito, preferencialmente de ordem Δt^2 ou maior; em seguida, resolvemos a equação de Poisson (5.48); finalmente, obtemos \mathbf{u}^{n+1} com (5.47).

Para obter a solução completa do problema, precisamos especificar as condições de contorno em x , y . Condições de contorno possíveis são as seguintes:

1. Velocidades especificadas na seção de entrada.
2. Advecção pura na seção de saída (?)
3. pressão nas seções de entrada e saída.
4. derivada normal da pressão em paredes rígidas onde $\mathbf{u} = \mathbf{0}$:

$$\mathbf{n} \cdot \bar{\nabla} p = \mu \mathbf{n} \cdot \bar{\nabla}^2 \mathbf{u}.$$

A solução agora será um vetor \mathbf{u} em função de 2 índices espaciais (i para x e j para y) e um índice temporal n para t . Computacionalmente, o campo de velocidade corresponde a um par de arrays,

$$\begin{aligned} \mathbf{u}[0..T, 0..N, 0..N], \\ \mathbf{v}[0..T, 0..N, 0..N], \end{aligned}$$

e o campo de pressão a um outro array:

$$p[0..T, 0..N, 0..N].$$

A notação introduzida acima procura se aproximar da implementação computacional: enquanto que \mathbf{u} representa *qualquer* elemento $\mathbf{u}_{i,j}^n$ do campo vetorial discreto da solução numérica, u e v representam *todos* os elementos do campo. O mesmo se aplica a p e p .

5.4 – Solução iterativa da equação de Poisson com uma malha geral $\Delta x \neq \Delta y$

A ideia aqui é adaptar a técnica de solução da equação de Laplace com *over-relaxation*, que abordamos nas seções 4.5 e 4.6 para a equação de Poisson. Primeiro, alguma discussão analítica. A equação de Poisson é da forma

$$\nabla^2 \phi = f. \quad (5.49)$$

É relativamente fácil “construir” soluções analíticas simples. Por exemplo, veja <https://farside.ph.utexas.edu/teaching/329/lectures/node71.html>: o problema

$$\nabla^2 \phi = 6xy(1-y) - 2x^3, \quad (5.50)$$

$$\phi(0, y) = f_{0y}(y) = 0, \quad (5.51)$$

$$\phi(1, y) = f_{1y}(y) = y(1-y), \quad (5.52)$$

$$\phi(x, 0) = f_{x0}(x) = 0, \quad (5.53)$$

$$\phi(x, 1) = f_{x1}(x) = 0, \quad (5.54)$$

possui solução analítica

$$\phi(x, y) = y(1-y)x^3 \quad (5.55)$$

(verifique). A média de ϕ no domínio é

$$\bar{\phi} = \int_{x=0}^1 \int_{y=0}^1 \phi(x, y) \, dy \, dx = \frac{1}{24}.$$

A discretização da equação de Poisson é como se segue:

$$\begin{aligned} \bar{\nabla}^2 \phi &= f, \\ \frac{\delta^2 \phi}{\delta x^2} + \frac{\delta^2 \phi}{\delta y^2} &= f, \\ \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y^2} &= f_{i,j} \\ \frac{\phi_{i+1,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} + \phi_{i,j-1}}{\Delta y^2} &= f_{i,j} + 2\phi_{i,j} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right) \\ \frac{\phi_{i+1,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} + \phi_{i,j-1}}{\Delta y^2} &= f_{i,j} + 2\phi_{i,j} \left(\frac{\Delta x^2 + \Delta y^2}{\Delta x^2 \Delta y^2} \right) \end{aligned}$$

Rearranjando,

$$\begin{aligned} 2\phi_{i,j} (\Delta x^2 + \Delta y^2) &= -f_{i,j} \Delta x^2 \Delta y^2 + \Delta y^2 (\phi_{i+1,j} + \phi_{i-1,j}) + \Delta x^2 (\phi_{i,j+1} + \phi_{i,j-1}) \\ \phi_{i,j} &= \frac{1}{2(\Delta x^2 + \Delta y^2)} [-f_{i,j} \Delta x^2 \Delta y^2 + \Delta y^2 (\phi_{i+1,j} + \phi_{i-1,j}) + \Delta x^2 (\phi_{i,j+1} + \phi_{i,j-1})] \end{aligned}$$

Defina

$$A_f = -\frac{(\Delta x \Delta y)^2}{2(\Delta x^2 + \Delta y^2)},$$

$$A_y = \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)},$$

$$A_x = \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)},$$

e obtenha finalmente

$$\phi_{i,j} = A_f f_{i,j} + A_y (\phi_{i+1,j} + \phi_{i-1,j}) + A_x (\phi_{i,j+1} + \phi_{i,j-1}). \quad (5.56)$$

Nós agora vamos adaptar `laplace-sor` para um novo programa, `poisson-sor`; além disso, nós vamos utilizar os mesmos índices para a pressão apresentados na figura 5.1, ou seja: os índices para ϕ correrão de $i = -1/2$ até $i = N_x + 1/2$, e de forma análoga em j .

First ghost point approach Primeiramente, desejamos uma estimativa inicial para ϕ que atenda exatamente às condições de contorno (5.51)–(5.54). Nós vamos fazer isso com uma *análise* aproximada das condições de contorno, para todos os pontos internos. É interessante observar que a grade da figura 5.1 não inclui pontos exatamente sobre o contorno. Para os valores conhecidos nos quatro lados do domínio, devemos ter:

$$\phi(0, y) = 0 \Rightarrow \frac{1}{2} [\phi_{-1/2,j} + \phi_{+1/2,j}] = 0, \quad j = 1/2, \dots, N_y - 1/2 \quad (5.57)$$

$$\phi(1, y) = y(1 - y) \Rightarrow \frac{1}{2} [\phi_{N_x-1/2,j} + \phi_{N_x+1/2,j}] = j\Delta y (L_y - j\Delta y), \quad j = 1/2, \dots, N_y - 1/2 \quad (5.58)$$

$$\phi(x, 0) = 0 \Rightarrow \frac{1}{2} [\phi_{i,-1/2} + \phi_{i,+1/2}] = 0, \quad i = 1/2, \dots, N_x - 1/2 \quad (5.59)$$

$$\phi(x, 1) = 0 \Rightarrow \frac{1}{2} [\phi_{i,N_y-1/2} + \phi_{i,N_y+1/2}] = 0, \quad i = 1/2, \dots, N_x - 1/2. \quad (5.60)$$

Para garantir a validade estrita das condições de contorno em nossa estimativa inicial, nós vamos analisar os pontos internos próximos aos contornos, e utilizar as relações acima para definir os valores dos pontos externos (“pontos-fantasma”).

Each of these things needs to be treated separately from (5.56), and that is what we are going to do now, until lunch.

Putting (5.56) and (5.67) together for $i = 1/2$,

$$\begin{aligned} \phi_{-1/2,j} &= -\phi_{1/2,j}, \\ \phi_{1/2,j} &= A_f f_{1/2,j} + A_y (\phi_{3/2,j} + \phi_{-1/2,j}) + A_x (\phi_{1/2,j+1} + \phi_{1/2,j-1}), \\ \phi_{1/2,j} &= A_f f_{1/2,j} + A_y (\phi_{3/2,j} - \phi_{1/2,j}) + A_x (\phi_{1/2,j+1} + \phi_{1/2,j-1}), \\ (1 + A_y) \phi_{1/2,j} &= A_f f_{1/2,j} + A_y \phi_{3/2,j} + A_x (\phi_{1/2,j+1} + \phi_{1/2,j-1}), \\ \phi_{1/2,j} &= \frac{1}{1 + A_y} [A_f f_{1/2,j} + A_y \phi_{3/2,j} + A_x (\phi_{1/2,j+1} + \phi_{1/2,j-1})]. \end{aligned}$$

Putting (5.56) and (5.68) together for $i = N_x - 1/2$,

$$\begin{aligned} y &= j\Delta y, \quad j = 1/2, \dots, N_y - 1/2; \\ 2y(1 - y) &= \phi_{N_x-1/2,j} + \phi_{N_x+1/2,j}, \\ \phi_{N_x+1/2,j} &= 2y(1 - y) - \phi_{N_x-1/2,j}, \\ \phi_{N_x-1/2,j} &= A_f f_{N_x-1/2,j} + A_y (\phi_{N_x+1/2,j} + \phi_{N_x-3/2,j}) + A_x (\phi_{N_x-1/2,j+1} + \phi_{N_x-1/2,j-1}), \\ \phi_{N_x-1/2,j} &= A_f f_{N_x-1/2,j} + A_y (2y(1 - y) - \phi_{N_x-1/2,j} + \phi_{N_x-3/2,j}) \\ &\quad + A_x (\phi_{N_x-1/2,j+1} + \phi_{N_x-1/2,j-1}), \\ (1 + A_y) \phi_{N_x-1/2,j} &= A_f f_{N_x-1/2,j} + A_y (2y(1 - y) + \phi_{N_x-3/2,j}) + A_x (\phi_{N_x-1/2,j+1} + \phi_{N_x-1/2,j-1}), \\ \phi_{N_x-1/2,j} &= \frac{1}{1 + A_y} [A_f f_{N_x-1/2,j} + A_y (2y(1 - y) + \phi_{N_x-3/2,j}) + A_x (\phi_{N_x-1/2,j+1} + \phi_{N_x-1/2,j-1})]. \end{aligned}$$

Putting (5.56) and (5.69) together for $j = +1/2$,

$$\phi_{i,-1/2} = -\phi_{i,+1/2},$$

$$\begin{aligned}\phi_{i,1/2} &= A_f f_{i,1/2} + A_y (\phi_{i+1,1/2} + \phi_{i-1,1/2}) + A_x (\phi_{i,3/2} + \phi_{i,-1/2}), \\ \phi_{i,1/2} &= A_f f_{i,1/2} + A_y (\phi_{i+1,1/2} + \phi_{i-1,1/2}) + A_x (\phi_{i,3/2} - \phi_{i,1/2}), \\ (1 + A_x) \phi_{i,1/2} &= A_f f_{i,1/2} + A_y (\phi_{i+1,1/2} + \phi_{i-1,1/2}) + A_x \phi_{i,3/2}, \\ \phi_{i,1/2} &= \frac{1}{1 + A_x} [A_f f_{i,1/2} + A_y (\phi_{i+1,1/2} + \phi_{i-1,1/2}) + A_x \phi_{i,3/2}].\end{aligned}$$

Putting (5.56) and (5.69) together for $j = N_y - 1/2$,

$$\begin{aligned}\phi_{i,N_y+1/2} &= -\phi_{i,N_y-1/2}, \\ \phi_{i,N_y-1/2} &= A_f f_{i,N_y-1/2} + A_y (\phi_{i+1,N_y-1/2} + \phi_{i-1,N_y-1/2}) + A_x (\phi_{i,N_y+1/2} + \phi_{i,N_y-3/2}), \\ \phi_{i,N_y-1/2} &= A_f f_{i,N_y-1/2} + A_y (\phi_{i+1,N_y-1/2} + \phi_{i-1,N_y-1/2}) + A_x (-\phi_{i,N_y-1/2} + \phi_{i,N_y-3/2}), \\ (1 + A_x) \phi_{i,N_y-1/2} &= A_f f_{i,N_y-1/2} + A_y (\phi_{i+1,N_y-1/2} + \phi_{i-1,N_y-1/2}) + A_x \phi_{i,N_y-3/2}, \\ \phi_{i,N_y-1/2} &= \frac{1}{1 + A_x} [A_f f_{i,N_y-1/2} + A_y (\phi_{i+1,N_y-1/2} + \phi_{i-1,N_y-1/2}) + A_x \phi_{i,N_y-3/2}].\end{aligned}$$

The 4 corners, however, are still troublesome. Note that, if $i = 1/2$ and $j = 1/2$, there are two ghost points. In this case we need to combine the two cases above into

$$\phi_{i,j} = \frac{1}{1 + A_x + A_y} [A_f f_{i,j} + A_y \phi_{i+1,j} + A_x \phi_{i,j+1}].$$

By the same token, if $i = 1/2$ and $j = N_y - 1/2$, we need to have

$$\phi_{i,j} = \frac{1}{1 + A_x + A_y} [A_f f_{i,j} + A_y \phi_{i+1,j} + A_x \phi_{i,j-1}].$$

We can proceed along the same lines (intuitively) for $i = N_x - 1/2$ and $j = 1/2$:

$$\phi_{i,j} = \frac{1}{1 + A_x + A_y} [A_f f_{i,j} + A_y (2y(1-y) + \phi_{i-1,j}) + A_x \phi_{i,j+1}].$$

Finally, for $i = N_x - 1/2$ and $j = N_y - 1/2$,

$$\phi_{i,j} = \frac{1}{1 + A_x + A_y} [A_f f_{i,j} + A_y (2y(1-y) + \phi_{i-1,j}) + A_x \phi_{i,j-1}].$$

Of course, all of that needs to be programmed very carefully. A direct attack implies separate re-calculation of each case inside the iterative procedure, which is how I first implemented it. But with sufficient memory available, we can do better. We re-write (5.56) as

$$\phi_{i,j} = B_{i,j}^f f_{i,j} + B_{i,j}^{i+} \phi_{i+1,j} + B_{i,j}^{i-} \phi_{i-1,j} + B_{i,j}^{j+} \phi_{i,j+1} + B_{i,j}^{j-} \phi_{i,j-1} + C_{i,j}, \quad (5.61)$$

where $C_{i,j}$ is needed to accomodate the boundary conditions, and now we calculate only once:

1) ($i = 1/2$) \vee ($j = 1/2$):

$$D = 1 + A_x + A_y,$$

$$B_{i,j}^f = \frac{A_f}{D},$$

$$B_{i,j}^{i+} = \frac{A_y}{D},$$

$$B_{i,j}^{i-} = 0,$$

$$B_{i,j}^{j+} = \frac{A_x}{D},$$

$$B_{i,j}^{j-} = 0,$$

$$C_{i,j} = 0.$$

2) ($i = 1/2$) \vee ($1/2 < j < N_y - 1/2$):

$$\begin{aligned} D &= 1 + A_y, \\ B_{i,j}^f &= \frac{A_f}{D}, \\ B_{i,j}^{i+} &= \frac{A_y}{D}, \\ B_{i,j}^{i-} &= 0, \\ B_{i,j}^{j+} &= \frac{A_x}{D}, \\ B_{i,j}^{j-} &= \frac{A_x}{D}, \\ C_{i,j} &= 0. \end{aligned}$$

3) ($i = 1/2$) \vee ($j = N_y - 1/2$):

$$\begin{aligned} D &= 1 + A_x + A_y, \\ B_{i,j}^f &= \frac{A_f}{D}, \\ B_{i,j}^{i+} &= \frac{A_y}{D}, \\ B_{i,j}^{i-} &= 0, \\ B_{i,j}^{j+} &= 0, \\ B_{i,j}^{j-} &= \frac{A_x}{D}, \\ C_{i,j} &= 0. \end{aligned}$$

4) ($1/2 < i < N_x - 1/2$) \vee ($j = 1/2$):

$$\begin{aligned} D &= 1 + A_x, \\ B_{i,j}^f &= \frac{A_f}{D}, \\ B_{i,j}^{i+} &= \frac{A_y}{D}, \\ B_{i,j}^{i-} &= \frac{A_y}{D}, \\ B_{i,j}^{j+} &= \frac{A_x}{D}, \\ B_{i,j}^{j-} &= 0, \\ C_{i,j} &= 0. \end{aligned}$$

5) ($1/2 < i < N_x - 1/2$) \vee ($1/2 < j < N_y - 1/2$) (and $D = 1$):

$$\begin{aligned} B_{i,j}^f &= A_f, \\ B_{i,j}^{i+} &= A_y, \\ B_{i,j}^{i-} &= A_y, \\ B_{i,j}^{j+} &= A_x, \\ B_{i,j}^{j-} &= A_x, \\ C_{i,j} &= 0. \end{aligned}$$

6) $(1/2 < i < N_x - 1/2) \vee (j = N_y - 1/2)$:

$$\begin{aligned} D &= 1 + A_x, \\ B_{i,j}^f &= \frac{A_f}{D}, \\ B_{i,j}^{i+} &= \frac{A_y}{D}, \\ B_{i,j}^{i-} &= \frac{A_y}{D}, \\ B_{i,j}^{j+} &= 0, \\ B_{i,j}^{j-} &= \frac{A_x}{D}, \\ C_{i,j} &= 0. \end{aligned}$$

7) $(i = N_x - 1/2) \vee (j = 1/2)$:

$$\begin{aligned} D &= 1 + A_x + A_y, \\ B_{i,j}^f &= \frac{A_f}{D}, \\ B_{i,j}^{i+} &= 0, \\ B_{i,j}^{i-} &= \frac{A_y}{D}, \\ B_{i,j}^{j+} &= \frac{A_x}{D}, \\ B_{i,j}^{j-} &= 0, \\ C_{i,j} &= \frac{A_y(2y(1-y))}{D}. \end{aligned}$$

8) $(i = N_x - 1/2) \vee (1/2 < j < N_y - 1/2)$:

$$\begin{aligned} D &= 1 + A_y, \\ B_{i,j}^f &= \frac{A_f}{D}, \\ B_{i,j}^{i+} &= 0, \\ B_{i,j}^{i-} &= \frac{A_y}{D}, \\ B_{i,j}^{j+} &= \frac{A_x}{D}, \\ B_{i,j}^{j-} &= \frac{A_x}{D}, \\ C_{i,j} &= \frac{A_y(2y(1-y))}{D}. \end{aligned}$$

9) $(i = N_x - 1/2) \vee (j = N_y - 1/2)$:

$$\begin{aligned} D &= 1 + A_y + A_x, \\ B_{i,j}^f &= \frac{A_f}{D}, \\ B_{i,j}^{i+} &= 0, \\ B_{i,j}^{i-} &= \frac{A_y}{D}, \end{aligned}$$

$$\begin{aligned} B_{i,j}^{j+} &= 0, \\ B_{i,j}^{j-} &= \frac{A_x}{D}, \\ C_{i,j} &= \frac{A_y(2y(1-y))}{D}. \end{aligned}$$

5.5 – Poisson equation with Neumann BCs

Let us now try to solve Poisson's equation (5.49) as in with the same forcing as before, but with Neumann BCs

$$\nabla^2 \phi = 6xy(1-y) - 2x^3, \quad (5.62)$$

$$\frac{\partial \phi(0, y)}{\partial x} = 0, \quad (5.63)$$

$$\frac{\partial \phi(1, y)}{\partial x} = 0, \quad (5.64)$$

$$\frac{\partial \phi(x, 0)}{\partial y} = 0, \quad (5.65)$$

$$\frac{\partial \phi(x, 1)}{\partial y} = 0. \quad (5.66)$$

Ghost point approach for the derivative in the normal direction Primeiramente, desejamos uma estimativa inicial para ϕ que atenda exatamente às condições de contorno (5.51)–(5.54). Nós vamos fazer isso com uma *análise* aproximada das condições de contorno, para todos os pontos internos. É interessante observar que a grade da figura 5.1 não inclui pontos exatamente sobre o contorno. Para os valores conhecidos nos quatro lados do domínio, devemos ter:

$$\frac{\partial \phi(0, y)}{\partial x} = 0 \Rightarrow \phi_{-1/2,j} = \phi_{+1/2,j}, \quad j = 1/2, \dots, N_y - 1/2 \quad (5.67)$$

$$\frac{\partial \phi(1, y)}{\partial x} = 0 \Rightarrow \phi_{N_x-1/2,j} = \phi_{N_x+1/2,j}, \quad j = 1/2, \dots, N_y - 1/2 \quad (5.68)$$

$$\frac{\partial \phi(x, 0)}{\partial y} = 0 \Rightarrow \phi_{i,-1/2} = \phi_{i,+1/2}, \quad i = 1/2, \dots, N_x - 1/2 \quad (5.69)$$

$$\frac{\partial \phi(x, 1)}{\partial y} = 0 \Rightarrow \phi_{i,N_y-1/2} = \phi_{i,N_y+1/2}, \quad i = 1/2, \dots, N_x - 1/2. \quad (5.70)$$

Para garantir a validade estrita das condições de contorno em nossa estimativa inicial, nós vamos analisar os pontos internos próximos aos contornos, e utilizar as relações acima para definir os valores dos pontos externos (“pontos-fantasma”).

Each of these things needs to be treated separately from (5.56), and that is what we are going to do now, until lunch.

Putting (5.56) and (5.67) together for $i = 1/2$,

$$\begin{aligned} \phi_{-1/2,j} &= \phi_{1/2,j}, \\ \phi_{1/2,j} &= A_f f_{1/2,j} + A_y(\phi_{3/2,j} + \phi_{-1/2,j}) + A_x(\phi_{1/2,j+1} + \phi_{1/2,j-1}), \\ \phi_{1/2,j} &= A_f f_{1/2,j} + A_y(\phi_{3/2,j} + \phi_{1/2,j}) + A_x(\phi_{1/2,j+1} + \phi_{1/2,j-1}), \\ (1 - A_y)\phi_{1/2,j} &= A_f f_{1/2,j} + A_y\phi_{3/2,j} + A_x(\phi_{1/2,j+1} + \phi_{1/2,j-1}), \\ \phi_{1/2,j} &= \frac{1}{1 - A_y} [A_f f_{1/2,j} + A_y\phi_{3/2,j} + A_x(\phi_{1/2,j+1} + \phi_{1/2,j-1})]. \end{aligned}$$

Putting (5.56) and (5.68) together for $i = N_x - 1/2$,

$$\phi_{N_x+1/2,j} = \phi_{N_x-1/2,j},$$

$$\begin{aligned}\phi_{N_x-1/2,j} &= A_f f_{N_x-1/2,j} + A_y (\phi_{N_x+1/2,j} + \phi_{N_x-3/2,j}) + A_x (\phi_{N_x-1/2,j+1} + \phi_{N_x-1/2,j-1}), \\ \phi_{N_x-1/2,j} &= A_f f_{N_x-1/2,j} + A_y (\phi_{N_x-1/2,j} + \phi_{N_x-3/2,j}) + A_x (\phi_{N_x-1/2,j+1} + \phi_{N_x-1/2,j-1}), \\ (1 - A_y) \phi_{N_x-1/2,j} &= A_f f_{N_x-1/2,j} + A_y \phi_{N_x-3/2,j} + A_x (\phi_{N_x-1/2,j+1} + \phi_{N_x-1/2,j-1}), \\ \phi_{N_x-1/2,j} &= \frac{1}{1 - A_y} [A_f f_{N_x-1/2,j} + A_y \phi_{N_x-3/2,j} + A_x (\phi_{N_x-1/2,j+1} + \phi_{N_x-1/2,j-1})].\end{aligned}$$

Putting (5.56) and (5.69) together for $j = +1/2$,

$$\begin{aligned}\phi_{i,-1/2} &= \phi_{i,+1/2}, \\ \phi_{i,1/2} &= A_f f_{i,1/2} + A_y (\phi_{i+1,1/2} + \phi_{i-1,1/2}) + A_x (\phi_{i,3/2} + \phi_{i,-1/2}), \\ \phi_{i,1/2} &= A_f f_{i,1/2} + A_y (\phi_{i+1,1/2} + \phi_{i-1,1/2}) + A_x (\phi_{i,3/2} + \phi_{i,1/2}), \\ (1 - A_x) \phi_{i,1/2} &= A_f f_{i,1/2} + A_y (\phi_{i+1,1/2} + \phi_{i-1,1/2}) + A_x \phi_{i,3/2}, \\ \phi_{i,1/2} &= \frac{1}{1 - A_x} [A_f f_{i,1/2} + A_y (\phi_{i+1,1/2} + \phi_{i-1,1/2}) + A_x \phi_{i,3/2}].\end{aligned}$$

Putting (5.56) and (5.69) together for $j = N_y - 1/2$,

$$\begin{aligned}\phi_{i,N_y+1/2} &= \phi_{i,N_y-1/2}, \\ \phi_{i,N_y-1/2} &= A_f f_{i,N_y-1/2} + A_y (\phi_{i+1,N_y-1/2} + \phi_{i-1,N_y-1/2}) + A_x (\phi_{i,N_y+1/2} + \phi_{i,N_y-3/2}), \\ \phi_{i,N_y-1/2} &= A_f f_{i,N_y-1/2} + A_y (\phi_{i+1,N_y-1/2} + \phi_{i-1,N_y-1/2}) + A_x (\phi_{i,N_y-1/2} + \phi_{i,N_y-3/2}), \\ (1 - A_x) \phi_{i,N_y-1/2} &= A_f f_{i,N_y-1/2} + A_y (\phi_{i+1,N_y-1/2} + \phi_{i-1,N_y-1/2}) + A_x \phi_{i,N_y-3/2}, \\ \phi_{i,N_y-1/2} &= \frac{1}{1 - A_x} [A_f f_{i,N_y-1/2} + A_y (\phi_{i+1,N_y-1/2} + \phi_{i-1,N_y-1/2}) + A_x \phi_{i,N_y-3/2}].\end{aligned}$$

The 4 corners, however, are still troublesome. Note that, if $i = 1/2$ and $j = 1/2$, there are two ghost points. In this case we need to combine the two cases above into

$$\phi_{1/2,j} = \frac{1}{1 - A_x - A_y} [A_f f_{1/2,j} + A_y \phi_{3/2,j} + A_x \phi_{1/2,j+1}].$$

By the same token, if $i = 1/2$ and $j = N_y - 1/2$, we need to have

$$\phi_{1/2,j} = \frac{1}{1 - A_x - A_y} [A_f f_{1/2,j} + A_y \phi_{3/2,j} + A_x \phi_{1/2,j-1}].$$

We can proceed along the same lines (intuitively) for $i = N_x - 1/2$ and $j = 1/2$:

$$\phi_{N_x-1/2,j} = \frac{1}{1 - A_x - A_y} [A_f f_{N_x-1/2,j} + A_y \phi_{N_x-3/2,j} + A_x \phi_{N_x-1/2,j+1}].$$

Finally, for $i = N_x - 1/2$ and $j = N_y - 1/2$,

$$\phi_{N_x-1/2,j} = \frac{1}{1 - A_x - A_y} [A_f f_{N_x-1/2,j} + A_y \phi_{N_x-3/2,j} + A_x \phi_{N_x-1/2,j-1}].$$

Of course, all of that needs to be programmed very carefully.

5.6 – Poisson with FFTs?

Consider the same Poisson 2D PDE again,

$$\nabla^2 \phi = f(x, y), \quad (5.71)$$

$$\frac{\partial \phi(0, y)}{\partial x} = 0, \quad (5.72)$$

$$\frac{\partial \phi(1, y)}{\partial x} = 0, \quad (5.73)$$

$$\frac{\partial \phi(x, 0)}{\partial y} = 0, \quad (5.74)$$

$$\frac{\partial \phi(x, 1)}{\partial y} = 0. \quad (5.75)$$

and adjust as we progress. Let us discretize this. Notation, keeping ultimately 3D operations in view, is very important here. Let us try to keep the usual physical space variables x, y, z discretized as

$$\begin{aligned} x_i &= i\Delta x = \frac{iL_x}{N_x}, \\ y_j &= j\Delta y = \frac{jL_y}{N_y}, \\ z_k &= k\Delta z = \frac{kL_z}{N_z}. \end{aligned}$$

Cyclic frequency f_l in one dimension is

$$f_l = \frac{l}{L_x} = \frac{l}{N_x \Delta x},$$

but I need 3 letters for cyclic frequencies in three different dimensions. I can think of greek letters, like λ, μ , and ν . Then,

$$\begin{aligned} \lambda_l &= l\Delta\lambda = \frac{l}{L_x} = \frac{l}{N_x \Delta x}, \\ \mu_m &= m\Delta\mu = \frac{m}{L_y} = \frac{m}{N_y \Delta y}, \\ \nu_n &= n\Delta\nu = \frac{n}{L_z} = \frac{n}{N_z \Delta z}. \end{aligned}$$

There is a difference between the mathematical Fourier Transform and the discrete, computer versions that are calculated over finite arrays. Given a function $\phi(x)$ we have the following Fourier transform pair:

$$\widehat{\phi}(\lambda) = \int_{x=-\infty}^{+\infty} \phi(x) e^{-2\pi i \lambda x} dx, \quad (5.76)$$

$$\phi(x) = \int_{\lambda=-\infty}^{+\infty} \widehat{\phi}(\lambda) e^{+2\pi i \lambda x} d\lambda. \quad (5.77)$$

Without a lot of rigor, let

$$\phi_i = \phi(x_i), \quad (5.78)$$

$$\widehat{\phi}_l = \widehat{\phi}(\lambda_l); \quad (5.79)$$

replace the infinite domain by

$$L_x = N_x \Delta x, \quad (5.80)$$

$$\Lambda_x = N_x \Delta \lambda, \quad (5.81)$$

and now calculate the discrete Fourier Transform pair:

$$\widehat{\phi}(\lambda_l) = \sum_{i=0}^{N_x-1} \phi(x_i) e^{-2\pi i \frac{l}{L_x} \frac{iL_x}{N_x} \Delta x},$$

$$\begin{aligned}\widehat{\phi}_l &= \Delta x \sum_{i=0}^{N_x-1} \phi_i e^{-\frac{2\pi i l i}{N_x}}, \\ \widehat{\phi}_l &= \Delta x \widehat{\Phi}_l.\end{aligned}$$

This defines the Numerical Direct Fourier Transform of $\phi(x)$ that is calculated by the computer in one dimension:

$$\widehat{\Phi}_l \equiv \sum_{i=0}^{N_x-1} \phi_i e^{-\frac{2\pi i l i}{N_x}}.$$

By the same token, the inverse is

$$\begin{aligned}\phi(x_i) &= \sum_{l=0}^{N_x-1} \widehat{\phi}(\lambda_l) e^{+2\pi i \frac{l}{N_x} \frac{i L_x}{N_x} \Delta \lambda}, \\ \phi_i &= \sum_{l=0}^{N_x-1} \widehat{\phi}_l e^{+\frac{2\pi i l i}{N_x} \Delta \lambda}, \\ \phi_i &= \sum_{l=0}^{N_x-1} \widehat{\phi}_l e^{+\frac{2\pi i l i}{N_x}} \frac{1}{N_x \Delta x}, \\ \phi_i &= \frac{1}{N_x} \sum_{l=0}^{N_x-1} \widehat{\Phi}_l e^{+\frac{2\pi i l i}{N_x}}, \\ \phi_i &= \frac{1}{N_x} \Phi_i.\end{aligned}$$

This on the other hand defines the Inverse Numerical Fourier Transform,

$$\Phi_i = \sum_{l=0}^{N_x-1} \widehat{\Phi}_l e^{+\frac{2\pi i l i}{N_x}}.$$

$$\begin{aligned}\widehat{\phi}(k, l) &= \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} \phi_{i,j} e^{-i(kx_i + ly_j)}, \\ &= \sum_{i=1}^{N_x} e^{-ikx_i} \underbrace{\left[\sum_{j=1}^{N_y} \phi_{i,j} e^{-il y_j} \right]}_{\widehat{\phi}_i(l)}, \\ &= \sum_{i=1}^{N_x} e^{-ikx_i} \widehat{\phi}_i(l), \\ \mathcal{F}\{\nabla^2 \phi\}(k_x, k_y) &= \sum_{i=1}^{N_x} \sum_{j=1}^{N_x} f_{i,j} e^{-i(k_x x_i + k_y y_j)}\end{aligned}$$

5.7 – Índices

Vamos imaginar (talvez isso mude) que precisamos de pontos-fantasma para u , v e p . Para os pontos da malha de p (e de escalares), precisamos de

$$p_{i,j}, \quad i = -1, \dots, N_x + 1, \quad j = -1, \dots, N_y + 1.$$

Para os pontos da malha de u , precisamos de

$$u_{i+1/2,j}, \quad i = -1, \dots, N_x, \quad j = -1, \dots, N_y + 1.$$

Para os pontos da malha de v , precisamos de

$$v_{i,j+1/2}, \quad i = -1, \dots, N_x + 1, \quad j = -1, \dots, N_y.$$

Mas isso pode mudar!

Eu creio que já tenho elementos suficientes para começar a programar.

5.8 – Marcha no tempo

Considere a componente x de (5.46) e um método explícito de marcha no tempo; olhando para (5.42) devemos ter

$$\begin{aligned} u_{i+1/2,j}^* &= u_{i+1/2,j}^n + \text{adv} + \nu \Delta t \left[\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} \right]_{i+1/2,j}^n \\ &= u_{i+1/2,j}^n + \text{adv} + \nu \Delta t \left[\frac{u_{i+3/2,j}^n - 2u_{i+1/2,j}^n + u_{i-1/2,j}^n}{\Delta x^2} + \frac{u_{i+1/2,j+1}^n - 2u_{i+1/2,j}^n + u_{i+1/2,j-1}^n}{\Delta y^2} \right] \end{aligned}$$

Analogamente, olhando para (5.43), devemos ter

$$\begin{aligned} v_{i,j+1/2}^* &= \text{adv} + \nu \Delta t \left[\frac{\delta^2 v}{\delta x^2} + \frac{\delta^2 v}{\delta y^2} \right]_{i,j+1/2}^n \\ &= u_{i,j+1/2}^n + \text{adv} + \nu \Delta t \left[\frac{v_{i+1,j+1/2}^n - 2v_{i,j+1/2}^n + v_{i-1,j+1/2}^n}{\Delta x^2} + \frac{v_{i,j+3/2}^n - 2v_{i,j+1/2}^n + v_{i,j-1/2}^n}{\Delta y^2} \right] \end{aligned}$$

Bibliography

- Boussinesq, J. (1903). Sur le débit, en temps de sécheresse, d'une source alimentée par une nappe d'eaux d'infiltration. *C. R. Hebd. Séances Acad. Sci.*, 136:1511–1517.
- Boussinesq, M. J. (1904). Recherches théoriques sur l'écoulement des nappes d'eau infiltrées dans le sol sur le débit des sources. *J des Mathématiques Pures et Appliquées*, 5éme Sér., 10:5–78.
- Briggs, W. L., Henson, V. E., e McCormick, S. F. (2000). *A multigrid tutorial*. SIAM.
- Brutsaert, W. (2005). *Hydrology. An introduction*. Cambridge University Press, Cambridge, UK.
- Brutsaert, W. e Lopez, J. P. (1998). Basin-Scale Geohydrologic Drought Flow Features of Riparian Aquifers in the Southern Great Plains. *Water Resour. Res.*, 34(2):233–240.
- Chen, Y. e Falconer, R. A. (1992). Advection-diffusion modelling using the modified QUICK scheme. *International Journal for Numerical Methods in Fluids*, 15(10):1171–1196.
- Chor, T. L. e Dias, N. L. (2015). Technical Note: A simple generalization of the Brutsaert and Nieber analysis. *Hydrol. Earth Syst. Sci.*, 19(6):2755–2761.
- Chow, V. T. (1959). *Open-Channel Hydraulics*. McGraw-Hill, New York.
- Chow, V. T., Maidment, D. R., e Mays, L. W. (1988). *Applied Hydrology*. McGraw-Hill, New York.
- Dias, N. L. (1995). Obtenção de curvas de remanso pelo método de Runge-Kutta. Em *XI Simpósio Brasileiro de Recursos Hídricos, Recife*, volume 4, páginas 277–285, Recife. Associação Brasileira de Recursos Hídricos.
- Dias, N. L. (2003). Obtenção de uma solução analítica da equação de difusão-advecção com decaimento de 1^a ordem pelo método da transformação de similaridade generalizada. *Revista Brasileira de Recursos Hídricos*, 8:181–188.
- Fisher, E. (1954). The period and amplitude of the van der Pol limit cycle. *J. Appl. Phys.*, 25(3):273–274.
- Fox, R. W., McDonald, A. T., e Pitchard, P. J. (2006). *Introduction to Fluid Mechanics, 2004*. John Wiley & Sons, Inc.
- French, R. H. (1986). *Open-Channel Hydraulics*. McGraw-Hill, New York.
- Hansen, P. B. (1992). Numerical solution of Laplace's equation. Electrical Engineering and Computer Science — Technical Reports 168, Syracuse University, College of Engineering and Computer Science. Retrieved in 2021-03-09T17:25:14.
- Hovland, P. D. e Heath, M. T. (1997). Adaptive SOR: A case study in automatic differentiation of algorithm parameters. *Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rep. ANL/MCS-P673-0797*.

- Jaluria, Y. e Torrance, K. E. (1986). *Computational Heat Transfer*. Hemisphere Publishing Corporation, New York.
- Kreider, D. L., Kuller, R. G., Ostberg, D. R., e Perkins, F. W. (1966). *An Introduction to Linear Analysis*. Addison-Wesley Publishing Company, Reading, Massachusetts.
- Kundu, P. K. (1990). *Fluid Mechanics*. Academic Press, San Diego.
- Leonard, B. P. (1979). A stable and accurate convective modelling procedure based on quadratic upstream interpolation. *Computer methods in applied mechanics and engineering*, 19(1):59–98.
- Lin, W. e Gray, D. M. (1971). Calculation of Backwater Curves by the Runge-Kutta Method. Obtido de http://www.usask.ca/hydrology/papers/Lin_1971.pdf, em 27/02/2017.
- Ludwig, D., Jones, D. D., e Holling, C. S. (1978). Qualitative analysis of insect outbreak systems: the spruce budworm and forest. *J. Anim. Ecol.*, 47(1):315–332.
- Nishikawa, H. (2021). The QUICK scheme is a third-order finite-volume scheme with point-valued numerical solutions. *International Journal for Numerical Methods in Fluids*, 93(7):2311–2338.
- O'Loughlin, E. M. e Bowmer, K. H. (1975). Dilution and decay of aquatic herbicides in flowing channels. *J. Hydrol.*, 26:217–235.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., e Flannery, B. P. (1992). *Numerical Recipes in C; The Art of Scientific Computing*. Cambridge University Press, Cambridge, 2^a edição.
- Prosperetti, A. e Tryggvason, G., editores (2007). *Computational Methods for Multiphase Flow*. Cambridge University Press, Cambridge.
- Versteeg, H. K. e Malalasekera, W. (2007). *An Introduction to Computational Fluid Dynamics*. Pearson Prentice-Hall.
- Yang, S. e Gobbert, M. K. (2009). The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions. *Applied mathematics letters*, 22(3):325–331.