Chapel as an alternative for teaching numerical metods

Nelson Luís Dias[1]

[1]Department of Environmental Engineering, Federal University of Paraná, Brazil
: nelsonluisdias@gmail.com : www.nldias.github.io

Meet-ups about teaching Chapel, Oct 9 2024

Opportunity and Motivation

This semester I had the opportunity to offer a graduate course in Numerical Methods (at an introductory/intermediate level).

The course draws material from a undergraduate course formerly taught using Python.

I took the opportunity to translate material to Chapel, expand it, and promote the language.

Students

- Few in number (5).
- Most without previous experience in numerical methods.
- Different previous programming languages (Python:2, C:1, Fortran:1, Matlab: 1).

What they know about programming

- **`if`**, **`then`**, **`for`**, ...
- functions/procedures/subroutines
- text files

What they don't seem to know so well

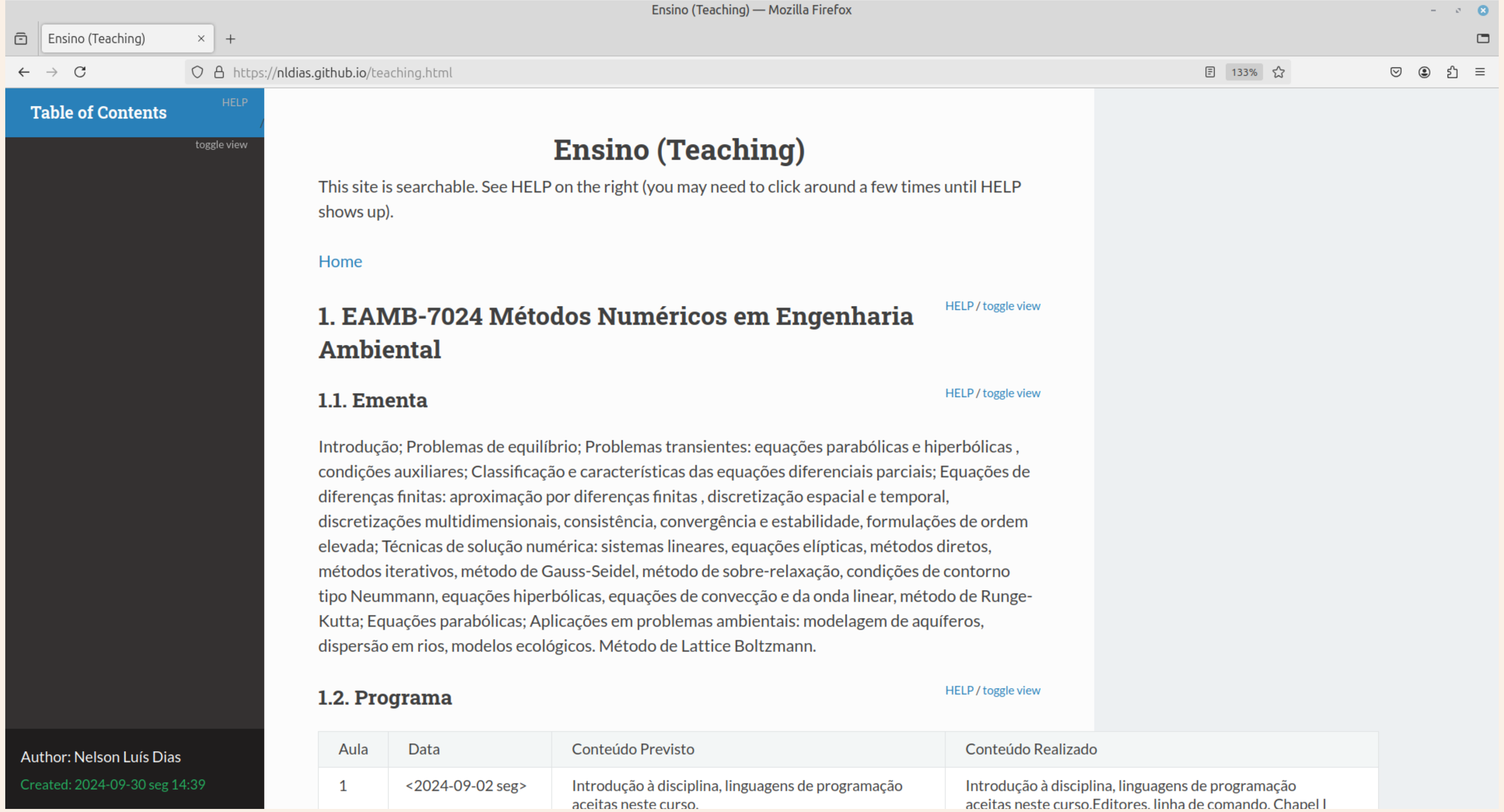
- binary files
- call by value/call by reference
- pointers in general

The instructor (myself :-))

- Numerical metods ***is not*** my main research area/experience.
- But I do data processing all the time, and often make incursions into numerical methods (ODEs and PDEs) for different purposes.
- Because I process relatively large amounts of turbulence data, speed of processing is an important factor. This (among other things) has drawn me to Chapel.

The course

Course material (in Portuguese) is posted at `nldias.github.io/teaching.html`.



Ensino (Teaching) — Mozilla Firefox

Ensino (Teaching)

https://nldias.github.io/teaching.html

Author: Nelson Luís Dias

Created: 2024-09-30 seg 14:39

1.4.1. Linguagens que podem ser utilizadas nesta disciplina

HELP / toggle view

A disciplina será lecionada com exemplos em Chapel, que é uma linguagem intrinsicamente paralela, com recursos de programação semelhantes a Python, e eficiência igual a Fortran. No entanto, os alunos poderão escolher várias linguagens para fazer seus trabalhos (veja a seguir).

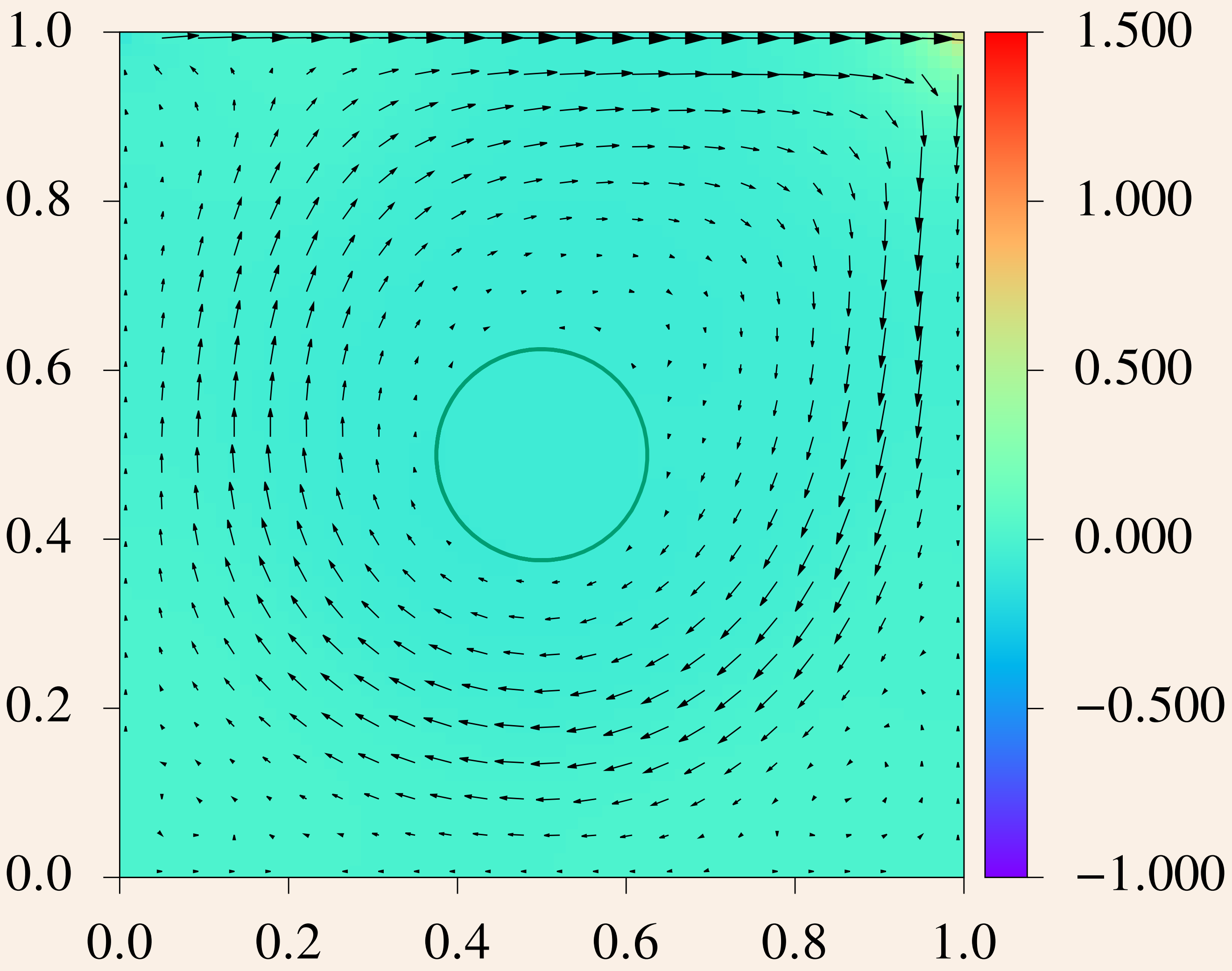
Por exemplo, eis aqui um programa que resolve uma equação diferencial com o método de Runge-Kutta:

```
// ----------------------------------------------------------------------------
// rungek4: resolve a equação diferencial
// dy/dx + y/x = sen(x)
// usando o método de Runge-Kutta de ordem 4
// ----------------------------------------------------------------------------
use Math only sin, cos;
use IO only openWriter;
config const h = 0.1;                // passo em x
const n = round(50/h):int;     // número de passos
var
   x,                          // variável independente
   y:                          // variável dependente
   [0..n] real;
x[0] = 0.0;                    // x inicial
y[0] = 0.0;                    // y inicial
// ----------------------------------------------------------------------------
```

Highlights

- “Standard” course on numerical solutions of ordinary and partial differential equations …
- …with finite differences, and a little bit of finite volumes.
- Examples (in Chapel) of Runge-Kutta, kinematic wave, diffusion, and Laplace/Poisson.
- Maybe there will be time for some material on simple approaches to the solution of the (primitive) Navier-Stokes equations.
- Two programming assignments, to be made in any procedural programming language that the student feels comfortable with (alas, no one opted to do it in Chapel).

Navier-Stokes with Immersed Boundary Elements done in Chapel



Promoting Chapel (to my students): how?

Promoting Chapel (to my students): how?

I don't really know, here is my approach:

- Let them choose the language for the assignments.
- Show examples in Chapel.
- Emphasize that 8 $\times$ faster with 8 cores is significant for some tasks.
- Exhibit Chapel's strenghts vis-à-vis established languages (my humble opinion only):

  Python, Matlab, … Much faster. Freedom to choose first and last indices in a range.

  Fortran Smaller, more elegant, easier to program, intrinsically parallel (Fortran supporters will say that it is too).

  C Much safer, much less prone to programming errors, freedom to choose first and last indices in a range, code is more clear.

- Class notes in Portuguese (LaTeX $\rightarrow$ pdf) available at `nldias.github.io/teaching` with many examples.

Examples of examples

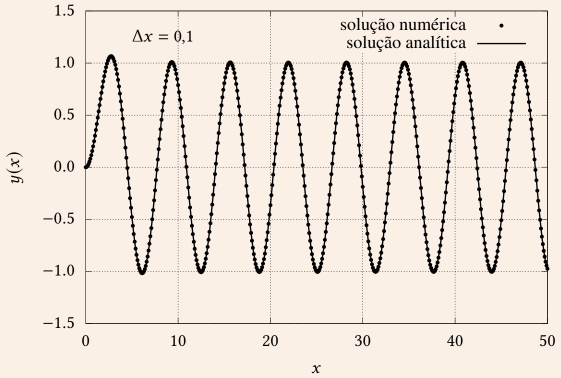


Figura 2.8: Comparação da solução analítica da equação (2.7) com a saída de `adbash.chpl`, para $\Delta x = 0{,}5$.

```
}
// ----------------------------------------------------------------------------
// agora a solução com passo h
// ----------------------------------------------------------------------------
x[0] = 0.0;                    // x inicial
y[0] = 0.0;                    // y inicial
x[1] = h;
y[1] = yq[10];
// ----------------------------------------------------------------------------
// os demais com Adams-Bashforth
// ----------------------------------------------------------------------------
for i in 2..n do {           // loop da solução numérica
   var ynp1 = adbash(x[i-1],y[i-1],x[i-2],y[i-2],h,ff);
   x[i] = i*h;
   y[i] = ynp1;
}
var erro = 0.0;                // calcula o erro relativo médio
   for i in 1..n do {
   var yana = sin(x[i])/x[i] - cos(x[i]);
   erro += abs( (y[i] - yana)/yana );
}
erro /= n ;
writef("erro relativo médio = %10.5dr",erro);
writeln();
const fou = openWriter("adbash.out",locking=false);
for i in 0..n do {             // imprime o arquivo de saída
   fou.writef("%12.6dr %12.6dr\n",x[i],y[i]);
}
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).

In each iteration (counted by nc) ω is updated differently for the first and second half-steps (steps between consecutive half-sweeps), which is decided in lines 44–52. These two half-sweeps occur inside the **for** in line 43. The heart of the "staggering" update is programmed in procedure `halfsw` in line 112: it boils down to starting the `j` index either on 1 or 2, and incrementing it by 2. These indices are declared in line 40, and swapped in line 54.

The new approach is short of miraculous; here we compare in table 3.4the performance of the serial programs `laplace-sor.chpl` and `laplace-asor.chpl`.

Tabela 3.4: Grid size N_n, number of iterations to convergence n_c, estimated $\overline{u}$, MAD and relative run-time t_r for the serial (`laplace-sor`) and accelerated serial (`laplace-asor`) versions of the solution of Laplace's equation with SOR.

	serial				accelerated serial			
N_n	n_c	$\overline{u}$	MAD	t_r	n_c	$\overline{\phi}$	MAD	t_r (s)
128	431	0.7500	2.5625×10^{-7}	0.0174	364	0.7500	3.5163×10^{-8}	0.0041
256	1934	0.7500	1.5653×10^{-6}	0.3223	714	0.7500	9.3864×10^{-8}	0.0267
512	6947	0.7500	6.6456×10^{-6}	4.8341	1404	0.7500	1.5544×10^{-7}	0.2587
1024	23955	0.7500	2.7032×10^{-5}	67.6316	2759	0.7500	2.1904×10^{-7}	2.0131
2048	80310	0.7499	1.0864×10^{-4}	916.1190	5208	0.7500	7.7417×10^{-7}	18.8378

What about parallelization? This is actually straightforward: comparing with the code from `laplace-sor-p.chpl`, we only need to change the procedure `halfsw`; therefore, we show only the modified part of `laplace-asor-p.chpl`:

Listagem 3.23: `laplace-asor-p` — Parallel solution of Laplace's equation with accelerated successive over relaxation.

```
111 inline proc halfsw(                  // beginning of halfsw
112    const in js: int
113    ) {
114    var deltaxm: [1..Nn-1] real;
115    forall i in 1..Nn-1 do {
116       var dj = {js..Nn-1 by 2};
117       var deltaym: [dj] real;
118       foreach j in dj do {
119          var phiavg = (phi[i+1,j]+phi[i-1,j]+phi[i,j-1]+phi[i,j+1])/4.0;
120          var deltaphi = omega*(phiavg - phi[i,j]);
121          phi[i,j] += deltaphi;
122          deltaphi = abs(deltaphi);
123          deltaym[j] += deltaphi;
124       }
125       deltaxm[i] = (+ reduce deltaym);
126    }
127    deltam += (+ reduce deltaxm);
128 }                                    // end of halfsw
```

And now we compare the serial and parallel implementations of the accelerated version in table 3.5.

Final thoughts

- Some beginning/intermediate students expect the "goodies" that come with Python, R, etc.. The fact that Chapel does not come with a module to calculate mean and standard deviations surprised one of them. Would a central repository — does it exist already? — help promote the language?
- Even without a formal introduction, students seem to understand Chapel code easily.
- As we know, Chapel is mature to be used in teaching and in "production" environments.

Lastly, Chapel is ***elegant***. The ability to express mathematical concepts and algorithms succintly and with clarity, albeit of an aesthetical/subjective nature, is, in my opinion, a big part of the fun that comes with programming in it.

Thanks for the attention.