ALTERNATIVAS DE PROCESSAMENTO NUMÉRICO EFICIENTE PARA A ENGENHARIA DO SÉCULO XXI

Nelson Luís Dias[1]

RESUMO

Neste trabalho, nós comparamos (do ponto de vista de desempenho computacional) a linguagem de programação interpretada Python com a linguagem de programação compilada Fortran. Ambas são extremamente populares na comunidade de cientistas e engenheiros, e ambas possuem prós e contras. Nós mostramos que, devidamente ampliadas com extensões adequadas (NumPy e Numba no caso de Python; BLAS no caso de Fortran – e, implicitamente, também no de Python), elas geram código que roda muitas vezes com a mesma ordem de magnitude de velocidade, havendo casos em que a implementação em Python com NumPy chega a ser mais rápida – embora, em geral, ainda haja uma clara vantagem para Fortran. Entretanto, tais diferenças têm pouca importância prática, exceto no caso de programas cujo tempo de processamento é extremamente longo (da ordem de horas ou mais), e a vantagem em termos de velocidade de *desenvolvimento* e em termos de *clareza* e de *concisão* do código proporcionada por Python não é desprezível. Conclui-se que, atualmente, não é mais importante escolher a "melhor" linguagem; tal conceito simplesmente não faz mais sentido. Muito pelo contrário, nós procuramos ressaltar a importância de engenheiros (computacionalmente) "poliglotas", já que a melhor escolha da ferramenta de computação frequentemente depende do problema em questão. Esperamos, assim, que os exemplos utilizados sejam úteis em si para estimular o aprendizado e a adoção desse tipo de ferramenta pelos estudantes de engenharia.

Palavras-chave: Fortran; Python; processamento científico.

ABSTRACT

ALTERNATIVES FOR EFFICIENT NUMERICAL PROCESSING FOR ENGINEERING OF XXI CENTURY

In this work, we compare the interpreted programming language Python to the compiled programming language Fortran from the point of view of computational performance. Both are very popular amidst scientists and engineers, and both have pros and cons. We show that duly extended by external libraries, both generate executable code that runs at speeds of the same order of magnitude, there being cases when Python with Numpy is actually faster, although, in general, there is a clear advantage for Fortran. Except in the case of programs that take very long to run (of the order of hours of CPU time or more), however, such differences are of little practical significance, and the advantage in terms of development speed and clarity and conciseness of the code afforded by Python can be important. We conclude that, nowadays, it no longer matters to choose a "best" overall programming language: this concept simply does not apply anymore. On the contrary, we emphasize the importance of "multilingual" engineers (from a computational perspective), since the best choice of computing tool often depends on the problem concerned. Finally, we hope that the examples provided here would be in themselves useful to stimulate the learning and adoption of such programming tools by engineering students.

Keywords: Fortran; Python; scientific processing.

1 Professor Associado, Universidade Federal do Paraná, Departamento de Engenharia Ambiental; nldias@ufpr.br

1- INTRODUÇÃO

Programas de computador (em geral) são *meios* para se fazer cálculos de engenharia. Portanto, quando se trata de decidir sobre uma linguagem de programação, seja para aprendê-la em um curso de graduação, seja para aplicá-la em um projeto de engenharia, a pergunta principal que devemos fazer é:

Quais são as melhores ferramentas para cada tarefa?

Não há uma única resposta, porque, como sempre, o resultado depende do conhecimento que o usuário tem da ferramenta. A época em que não havia praticamente nenhuma opção, exceto o compilador de Fortran, disponível no único computador da empresa ou da universidade já passou há muito tempo. Hoje em dia, existem inúmeras opções, desde planilhas de cálculo até aplicações específicas, tais como programas de cálculo estrutural ou programas de mecânica dos fluidos computacional, e a diversidade de escolhas é facilitada pelo fato de que os computadores são realmente *individuais*.

Por outro lado, linguagens de programação *gerais*, capazes de implementar um grande número de algoritmos, continuam a representar um papel importante no desenvolvimento de programas de computadores para engenharia. Talvez a principal mudança das últimas duas décadas seja o fato de que muitas dessas linguagens, agora, contam com um número muito grande de bibliotecas de aplicações, que tornam supérflua a implementação pelo usuário de tarefas específicas (tais como integração numérica ou a solução de sistemas de equações lineares). Exemplos de linguagens com tais extensões são Matlab© e Python.

O custo de se adotá-las é que, via de regra, elas são linguagens interpretadas, de forma que seu desempenho para tarefas que usam intensivamente operações de ponto flutuante deixa muito a desejar. Para tais tarefas, até bem pouco tempo, o único recurso viável era a implementação dos algoritmos de interesse diretamente em linguagens compiladas, tais como C, C++, e Fortran.

Neste trabalho, nós exploraremos algumas técnicas para fazer bom uso de Fortran e Python. Isso será feito por meio de alguns exemplos reais (com diferentes níveis de complexidade) implementados tanto em Fortran quanto em Python. Alguns resultados são surpreendentes:

- a diferença entre as duas linguagens pode ser bem menor que uma ordem de magnitude na velocidade de processamento, se ambas utilizarem as melhores ferramentas possíveis de implementação. No caso de Python, as ferramentas que exploraremos neste trabalho são NumPy e Numba;
- da mesma forma, muitas vezes é importante compilar (do Inglês *to compile*) e linqueditar (do Inglês *to link*) Fortran com bibliotecas de processamento básico de álgebra linear (BLAS), sem o que, em alguns casos, Fortran chega a ficar mais lento que Python otimizado com as ferramentas acima (sendo que, elas mesmas, usam BLAS).

Nosso foco não é indicar uma linguagem "vencedora": – de fato, a ideia de que exista uma única linguagem de programação aplicável a todos os casos que o engenheiro encontrará em sua vida profissional parece fazer cada vez menos sentido. Ao contrário, nós vamos procurar enfatizar a importância de utilizar a "ferramenta certa" em cada linguagem. Neste sentido, a relevância dos resultados aqui apresentados para o ensino de engenharia está em apresentar diversas opções aos alunos ao longo do curso, e enfatizar os aspectos algorítmicos e de eficiência computacional.

2- PACOTES COMPUTACIONAIS UTILIZADOS

Todos os exemplos mostrados neste trabalho foram feitos em sistema operacional GNU-Linux, distribuição Xubuntu 13.04. Todos os pacotes necessários foram instalados a partir do repositório padrão da distribuição, e não foram recompilados para tirar partido da máquina utilizada. Os pacotes instalados e utilizados foram:

Fortran	`gfortran (4.7.3)`
BLAS	`libatlas-base-dev, libatlas-dev, libatlas3-base, libblas-dev (3.8.4-9)`
Python	`python (2.7.4)`
NumPy	`python-numpy (1.7.1)`
SciPy	`python-scipy (0.11)`
pip	`python-pip (1.3.1)`
git	`git (1.8.1.2)`

Além disso, foi necessário instalar Numba manualmente (o significado dos termos Numba e NumPy é dado na seção 3), já que não encontramos, ou ainda não há, um pacote **`.deb`** no repositório padrão para o Xubuntu. A melhor forma que encontramos para instalar Numba foi rodar um *script* em *bash*, obtido em <https://groups.google.com/a/continuum.io/forum/#!msg/numba-users/JWUIGi_BD0Y/ lf56HxPMM0IJ>.

3- ASPECTOS GERAIS DE FORTRAN, PYTHON, E EXTENSÕES UTILIZADAS

Fortran é hoje, claramente, um termo genérico, que pode significar muitas coisas. Observamos, inicialmente, que a evolução da linguagem, desde os seus primórdios (BACKUS *et al.*, 1956), criou a necessidade de compatibilidade com o passado: de certa forma, cada nova versão da linguagem criava uma "nova" linguagem, que passava a conviver com uma linguagem "antiga". Nesse sentido, podemos dizer que, atualmente, Fortran são várias linguagens em uma. As diversas linguagens são muitas, mas, grosso modo, elas podem ser resumidas em: Fortran-66, Fortran-77, Fortran-90, Fortran-95, Fortran-2003 e Fortran-2008.

O resultado do requisito de compatibilidade com o passado é que, hoje, Fortran é uma linguagem grande, e relativamente mais difícil de aprender e dominar que linguagens mais modernas, tais como Matlab© e Python. Entretanto, Fortran goza da vantagem de permanecer extremamente popular entre cientistas e engenheiros e também de ser, provavelmente, a linguagem de programação que gera o código mais rápido – é possível que este último fato seja muito mais o resultado de décadas de investimento em bons compiladores do que de virtudes específicas da linguagem.

Ao contrário de Fortran, Python (python.org; LUTZ, 2008), que é muito mais jovem, permanece claramente como uma única linguagem, embora tenham ocorrido diversas "bifurcações" ao longo de sua história, com o resultado inevitável de algumas duplicações de funcionalidade (em outras palavras: assim como em Fortran, também em Python há mais de uma maneira para se fazer a mesma coisa na linguagem – mas não tantas). De certa forma, Python 3.*x* tenta lidar com esse problema ao abrir mão, deliberadamente, da "compatibilidade com o passado": em Python 3.*x*, diversas características das versões anteriores foram declaradas obsoletas e retiradas da linguagem, o que obriga os programadores a modificar o código fonte para que os programas em Python das versões anteriores possam rodar em Python 3.*x*. Apesar de ser desejável (do ponto de vista da comunidade de programadores em Python) atualizar todos os códigos para Python 3.*x*, neste trabalho, nós utilizamos (ainda) Python 2.7, por ser essa a versão implementada por *default* no sistema operacional em que os programas foram rodados.

Uma das maiores vantagens reconhecidas pelos programadores em Python é o número de bibliotecas padrão que permitem que uma grande variedade de tarefas possa ser executada de forma quase trivial; e a existência de extensões, além das bibliotecas padrão, que ampliam ainda mais a gama de aplicações possíveis. Além disso, a linguagem é pequena e simples, e (na opinião do autor) muito fácil de aprender.

No caso específico de programação científica, as seguintes extensões são dignas de nota: NumPy (www.numpy.org; OLIPHANT, 2006), Numba (numba.pydata.org), SciPy (www.scipy.org), SymPy (sympy.org) e Sage (www.sagemath.org); elas consistem de bibliotecas numéricas e simbólicas pré-compiladas (em Fortran ou em C!) e amplamente disponíveis. Neste trabalho, nós prestaremos particular atenção aos dois primeiros, para os quais as seguintes descrições podem ser encontradas:

NumPy: (OLIPHANT, 2006) "Numpy baseia-se no (e é um sucessor do) bem-sucedido objeto cujo nome é "Numeric array". Seu objetivo é criar a pedra fundamental de um ambiente útil para a programação científica" [tradução nossa].

Numba: (numba.pydata.org) "Numba é um compilador em tempo real especializado que compila código Python com anotações (por meio de "decorações") em LLVM. Seu objetivo é uma integração natural com as rotinas científicas em Python e

> proudzir código de máquina otimizado, assim como proporcionar integração com outras linguagens de programação" [tradução nossa].

Na prática, NumPy adiciona um novo tipo a Python, o *array*, e numerosas rotinas para calcular (por exemplo, entre muitas outras coisas) produtos escalares, autovalores e autovetores, soluções de sistemas lineares de equações, etc. Além disso, NumPy permite o uso de uma sintaxe vetorial extremamente concisa e poderosa, como teremos a oportunidade de ver na seção 6.

Numba, por outro lado, adiciona a capacidade de *compilar* Python, utilizando uma tecnologia denominada compilação *just-in-time*. Numba requer que sejam incluídas informações adicionais sobre o tipo de argumentos que uma determinada função recebe; essas informações são então utilizadas pelo compilador *just-in-time* para gerar código compilado. Com isso, Numba tem o potencial de tornar Python tão rápida quanto linguagens compiladas. Neste trabalho, teremos a oportunidade de testar o quanto isso é atingível em diversos exemplos.

BLAS (http://www.netlib.org/blas), por sua vez, proporciona as bibliotecas que tornam Fortran (e muitas outras linguagens – inclusive Python) ainda mais rápidas, ao implementar operações básicas de álgebra linear de forma a tirar o máximo de proveito possível das instruções de máquina disponíveis em cada processador. As ideias fundamentais das implementações de BLAS podem ser encontradas em Lawson *et al.* (1979); as implementações detalhadas de numerosos algoritmos são dadas, por exemplo, por Golub e van Loan (1996).

4- FORTRAN *x* PYTHON, SEM BLAS

Consideremos, inicialmente, alguns exemplos em que Fortran e Python são utilizados *sem* o "auxílio" (explícito, porque Python+NumPy podem fazê-lo independentemente de nossa vontade) de BLAS. A Listagem 1 mostra um primeiro programa muito simples, escrito em Fortran. Ele simplesmente calcula os elementos a_{ij} de uma matriz, de acordo com $a_{ij} = i + j$; para isso, é preciso fazer dois laços, em *i* e em *j*. Para demorar um pouco mais, nós dimensionamos uma matriz grande, de 5000 × 5000 elementos. O programa utiliza a rotina padrão de Fortran **`cpu_time`** para calcular o tempo de processamento. A compilação e a execução são feitas com

```
% gfortran -o3 cacumij.f90 -o cacumij
% ./cacumij
% 0.280000001
```

Um programa equivalente em Python é mostrado na Listagem 2, e utiliza a rotina **`time`**, do módulo **`time`**, para calcular o tempo de processamento. É importante observar que os tempos de processamento reportados aqui são obtidos a partir de rodadas individuais: quando o mesmo programa é rodado diversas vezes, os tempos de processamento variam ligeiramente. Estritamente falando, seria necessário reportar *médias* sobre várias repetições (um exemplo disso é dado na penúltima seção deste trabalho). No entanto, para os nossos propósitos, os tempos de processamento reportados são suficientes para avaliarmos a eficiência computacional das diversas alternativas computacionais que nós estudamos aqui.

Em **`cacum.py`**, na Listagem 2, observe-se a utilização de NumPy: a matriz **`a`** é criada na linha 9. Os dois *arrays* de inteiros **`ix`** e **`iy`** são agora "blocos" para construir a matriz **`a`** em uma declaração de uma única linha (10).

Listagem 1: cacumij.f90

```
01 program cacumij
02 integer, parameter :: n = 5000
03 real (kind=8) :: a(0:n-1,0:n-1)
04 integer (kind=4) :: i,j
05 call cpu_time(t1)
06 do i = 0,n-1
07    do j = 0,n-1
08       a(i,j) = i+j
09    end do
10 end do
11 call cpu_time(t2)
12 write(*,*) t2 - t1
13 end program
```

Rodando o programa, obtemos:

```
% ./cacum.py
% 0.112325191498
```

Listagem 2: cacum.py

```
01 #!/usr/bin/python
02 from numpy import arange,
array,int32,\
03   float64,zeros
04 from time import time
05 n = 5000
06 t1 = time()
07 ix = arange(n,dtype=int32)
08 iy = arange(n,dtype=int32)
09 a = zeros((n,n),float64)
10 a = ix[:,None] + iy[None,:]
11 t2 = time()
12 print t2 - t1
```

O resultado não deixa de ser impressionante: nosso primeiro programa em Python rodou mais rápido que seu equivalente em Fortran. Os leitores mais acostumados com computação científica e computação em geral reconhecerão, entretanto, que (como sempre) a grande questão é a quantidade de otimizações que estão sendo aplicadas em cada caso. O programa em Python, ao utilizar NumPy, evita quaisquer laços e defere todas as operações às rotinas subjacentes de NumPy, que são todas compiladas de forma otimizada em C ou em Fortran. Já o programa em Fortran está percorrendo a matriz **a** por *linha*, enquanto Fortran, na verdade, armazena *arrays* por *coluna*. Existe, portanto, terreno para melhorar nosso programa em Fortran e tornar a "competição" mais justa.

Note-se, de passagem, que o algoritmo $a_{ij} = i + j$ aparentemente não é vetorizável: BLAS (cuja utilização surgirá daqui a pouco) não se aplica. Mesmo assim, **cacum.py** pôde ser escrito de forma vetorial, dispensando o uso de laços para o cálculo individual dos elementos de **a**.

Listagem 3: cacumji.f90

```
01 program cacumji
02 integer, parameter :: n = 5000
03 real (kind=8) :: a(0:n-1,0:n-1)
04 integer (kind=4) :: i,j
05 call cpu_time(t1)
06 do j = 0,n-1
07    do i = 0,n-1
08       a(i,j) = i+j
09    end do
10 end do
11 call cpu_time(t2)
12 write(*,*) t2 - t1
13 end program
```

Como já observamos, **cacum.py** é cerca de duas vezes mais rápido que **cacumij.f90**. O que está errado? A solução (para Fortran) é relativamente simples: trocando-se o laço de **i** com o laço de **j**, a matriz **a** passará a ser percorrida por colunas, como mostra a Listagem 3.

O novo programa é compilado e rodado com:

```
% gfortran -o3 cacumji.f90 -o cacumji
./cacumji
% 0.104000002
```

A diferença entre as listagens 1 e 3 é que, no segundo caso, o laço interno percorre os elementos adjacentes, na memória, da estrutura de dados. Esse é um "truque" bem conhecido de programadores Fortran tradicionais. É reconfortante que o truque continue funcionando muito bem no século XXI. Agora, Fortran voltou a ficar (um pouco) mais rápido que Python. Esse tipo de exemplo "simples" pode ser usado para motivar a compreensão, pelos alunos de engenharia, quanto à importância de se implementar algoritmos capazes de tirar proveito das características de cada linguagem de programação.

Existem outras maneiras de programar a mesma coisa em Python, e a Listagem 4 mostra o que, talvez, seja a forma mais ineficiente. Rodando esse programa, obtemos:

```
% ./cacumij-slow.py
% 3.71550297737
```

Trata-se, definitivamente, de um retrocesso: note a perda de velocidade de processamento quando calculamos a matriz **a** sem o auxílio explícito das bibliotecas de NumPy.

Listagem 4: cacumij-slow.py

```
01 #!/usr/bin/python
02 from numpy import arange, array,zeros
03 from time import time
04 n = 5000
05 t1 = time()
06 a = zeros((n,n),float)
07 def aom(a):
08    for i in range(n):
09       for j in range(n):
10          a[i,j] = i+j
11 aom(a)
12 t2 = time()
13 print t2 - t1
```

Finalmente, a Listagem 6 mostra uma última opção para se fazer a mesma coisa. Ela é rodada com

```
% ./cacumij.py
% 0.197266101837
```

A grande novidade da Listagem 5 é o aparecimento de Numba, na linha 4 (onde são importados **jit** e **float64**), e na linha 8 (que "decora" – no sentido de "enfeitar", e não no sentido de "memorizar" – a função **aom**). A linha 8 informa a Numba que a rotina **aom** tem como argumento uma matriz cujos elementos são números de ponto flutuante de 64 bits. Compare os tempos de execução das listagens 4 e 5: com o uso de Numba, o programa, agora, roda uma ordem de magnitude mais rapidamente. Ele ainda é cerca de duas vezes mais *lento* que o correspondente programa em Fortran, **cacumji.f90**; no entanto, o ganho de velocidade com Numba (assim como o uso de sintaxe vetorial com NumPy na Listagem 2) é relevante. Mesmo sem o uso da sintaxe vetorial, Numba permitiu rodar um programa em Python em uma velocidade da mesma ordem que a obtida com Fortran.

Listagem 5: cacumij.py

```
01 #!/usr/bin/python
02 from numpy import arange, array,zeros
03 from time import time
04 from numba import jit,float64
05 n = 5000
06 t1 = time()
07 a = zeros((n,n),float)
08 @jit(argtypes=[float64[:,:]])
09 def aom(a):
10   for i in range(n):
11      for j in range(n):
12         a[i,j] = i+j
13 aom(a)
14 t2 = time()
15 print t2 - t1
```

Até aqui, algumas breves conclusões:

- um bom conhecimento dos detalhes de uma linguagem (no caso, como Fortran armazena matrizes) é muitas vezes necessário para otimizar um programa;
- a ordem em que uma matriz é percorrida (*ij,ji*) importa muito, porque a CPU é muito mais rápida do que qualquer memória. A diferença de velocidade está *aumentando*: a tecnologia está tornando as CPUs relativamente mais e mais rápidas que as memórias;
- depois de otimizado, Fortran ainda é a linguagem mais rápida;
- no entanto, obtivemos a mesma ordem de grandeza de tempo de CPU com Python, de duas maneiras diferentes (sem e com Numba);
- compiladores *just-in-time* (Numba) estão chegando próximo do desempenho de Fortran.

5- O USO EFICAZ DE BLAS

Existem atualmente várias implementações possíveis para BLAS (<http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms>). Algumas delas são: *Accelerate, ACML, C++ AMP BLAS, ATLAS, ESSL, Eigen BLAS, Goto BLAS, HP MLIB, Intel MKL, MathKeisan, Netlib BLAS, Netlib CBLAS, PDLIB/SX, SCSL, Sun Performance Library, SurviveGotoBLAS2, OpenBLAS, cuBLAS*.

Conforme observamos acima, nossa escolha neste trabalho foi por ATLAS, embora isso não signifique nenhuma vantagem específica, nem que ATLAS venha a produzir necessariamente velocidades maiores de processamento em relação às demais implementações de BLAS.

Listagem 6: cabum.f90

```
01 program cabum
02 integer, parameter :: size = 1000
03 real(kind=8)::a(size,size),
b(size,size), &
04 c(size,size)
05 integer (kind=4) :: seed
06 call random_seed()
07 call random_number(a)
08 call random_number(b)
09 call cpu_time(t1)
10 c = matmul(a,b)
11 call cpu_time(t2)
12 write(*,*) t2 - t1
13 end program
```

O efeito que BLAS provoca sobre a velocidade de processamento de alguns programas pode ser considerável. Nesta seção, portanto, vamos verificar qual é o *incremento* na velocidade de processamento de um programa quando o compilamos com BLAS. Novamente, os exemplos são baseados em códigos

muito simples e fáceis de entender. O programa em Fortran que vamos utilizar encontra-se na Listagem 6. Ele simplesmente preenche duas matrizes grandes, **a** e **b**, com números aleatórios, e calcula o seu produto.

Para compilar e rodar **cabum.f90** sem BLAS, fazemos:

```
% ./gfortran -o3 cabum.f90 -o cabum-noblas
% ./cabum-noblas
% 0.727999985
```

Em seguida, repetimos a compilação do mesmo programa, mas, agora, nós o linqueditamos com BLAS:

```
% ./gfortran -o3 -fexternal-blas cabum.f90
-lblas -o cabum-blas
% ./cabum-blas
% 0.167999998
```

Portanto, a simples compilação utilizando BLAS multiplicou a velocidade de processamento em cerca de cinco vezes.

Listagem 7: cabum.py

```
01 #!/usr/bin/python
02 import numpy
03 n = 1000
04 a = numpy.random.rand(n,n)
05 b = numpy.random.rand(n,n)
06 from time import time
07 t1 = time()
08 c = numpy.dot(a, b)
09 t2 = time()
10 print t2-t1
```

Naturalmente, é possível fazer essencialmente a mesma coisa com NumPy, como mostra a Listagem 7. Lembremo-nos de que Python não requer compilação; para rodar **cabum.py** fazemos:

```
% ./cabum.py
% 0.0847289562225
```

Nesse caso, é Python que "ganha", rodando cerca de duas vezes mais rapidamente que o código correspondente em Fortran. Eis aqui nossas rápidas conclusões a respeito de BLAS:

- BLAS faz toda a diferença na geração de códigos computacionalmente eficientes, sendo mandatório em qualquer trabalho numérico com objetivos profissionais;
- BLAS torna um programa Fortran otimizado para multiplicar matrizes cerca de cinco vezes mais rápido;
- mas NumPy também tira toda a vantagem possível de BLAS: nesse exemplo, Python/NumPy é duas vezes mais rápido que Fortran.

6- SOLUÇÃO NUMÉRICA DE UMA EQUAÇÃO DIFERENCIAL PARCIAL

Nesta seção, nós resolvemos um problema consideravelmente mais realista, do tipo que aparece muitas vezes em aplicações de engenharia. Considere-se a equação de difusão-advecção unidimensional com velocidade constante *U*, e difusividade constante *D*,

$$\frac{\partial \phi}{\partial t} - U\frac{\partial \phi}{\partial x} = D\frac{\partial^2 \phi}{\partial x^2} \qquad (1)$$

com condições de contorno

$$\phi(x, 0) = 1 \qquad (2)$$

$$\phi(0, t) = 0 \qquad (3)$$

$$\frac{\partial \phi}{\partial x}(1, t) = 0 \qquad (4)$$

com $U = 1$, $D = 2$.

A solução numérica de equações diferenciais é parte do programa das disciplinas lecionadas pelo autor na graduação de Engenharia Ambiental da UFPR; nelas, adotou-se o uso de Python pela simplicidade e rapidez com que a linguagem pode ser aprendida, sobrando mais tempo para a compreensão de métodos numéricos e de algoritmos. O material lecionado, incluindo uma breve introdução a Python, pode ser encontrado em Dias (2013).

Uma discretização simples de diferenças finitas para esse problema é usar um esquema *downwind* implícito para $\partial \phi / \partial x$ e um esquema totalmente implícito no lado direito. A discretização resultante é:

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

com $x = i\Delta x$ e $t = n\Delta t$. Passando todos os termos em n+1 para o lado esquerdo, e todos os termos em n para o lado direito, e rearrumando, tem-se:

$$-Fo\phi_{i-1}^{n+1} + (1 + 2Fo + Co)\phi_i^{n+1} - (Co + Fo)\phi_{i+1}^{n+1} = \phi_i^n \quad (5)$$

em que $Co = U\Delta t/\Delta x$ é o número de Courant, e $Fo = D\Delta t/\Delta x^2$ é o número de Fourier.

Como sempre, as condições de contorno (3)–(4) produzem linhas especiais: para $i = 1$, $\phi_0^{n+1} = 0$ e para $i = N - 1$, $\phi_N^{n+1} = \phi_{N-1}^{n+1}$ (derivada nula). Então, (5) torna-se, para as primeira e última linhas, respectivamente:

$$(1 + 2Fo + Co)\phi_1^{n+1} - (Co + Fo)\phi_2^{n+1} = \phi_1^n \quad (6)$$

$$-Fo\phi_{N-2}^{n+1} + (1 + Fo)\phi_{N-1}^{n+1} = \phi_{N-1}^n \quad (7)$$

Listagem 8: lindad.py: tridag

```
01 #!/usr/bin/python
02 # -*- coding: iso-8859-1 -*-
03 from __future__ import print_function
04 from __future__ import division
05 from time import time
06 from deltas import dx,dt
07 from sys import argv, exit
08 from numpy import zeros
09 from numba import jit,float64
10 @jit(argtypes=[float64[:,:],float64[:]])
11 def tridag(A,y):
12     m = A.shape[0]
13     n = A.shape[1]
14     if m != 3 :
15         exit("m != 3 em tridag")
16     pass
17     o = y.shape[0]
18     if o != n :
19         exit("n != 0")
20     pass
21     x = zeros(n,float)
22     gam = zeros(n,float)
23     if A[1,0] == 0.0 :
24         exit("Erro 1 em tridag")
25     pass
26     bet = A[1,0]
27     x[0] = y[0]/bet
28     for j in range(1,n):
29         gam[j] = A[2,j-1]/bet
30         bet = A[1,j] - A[0,j]*gam[j]
31         if (bet == 0.0):
32             exit("Erro 2 em tridag")
33         pass
34         x[j] = (y[j] - A[0,j]*x[j-1])/
bet
35     pass
36     for j in range(n-2,-1,-1):
37         x[j] -= gam[j+1]*x[j+1]
38     pass
39     return x
40 pass
```

Segue-se que é necessário resolver uma matriz tridiagonal a cada passo de tempo n. Um algoritmo clássico para a solução de sistemas com matrizes tridiagonais é apresentado em Press *et al.* (1992), com o nome de **`tridag`**. A Listagem 7 mostra uma **`tridag`** modificada para o "estilo" de Python e NumPy. O arquivo **`deltas.py`** simplesmente define **`dx=0.0001`** e **`dt=0.0001`**. A Listagem 8 mostra o restante do mesmo arquivo (**`lindad.py`**) com o programa que resolve a equação diferencial com um esquema de diferenças finitas implícito utilizando (5)–(7).

Do ponto de vista educacional, há dois pontos importantes a serem enfatizados:

1. **`tridag`** não é facilmente *vetorizável*: não é possível especificar operações que se apliquem a linhas e colunas inteiras das matrizes e vetores envolvidos de uma vez só. Por isso, como veremos a seguir, as linhas 9–10 da Listagem 8 (que invocam Numba) são fundamentais, proporcionando um ganho significativo de velocidade de processamento;
2. já a montagem da matriz do sistema é *altamente vetorizável*: nas linhas 48–53 da Listagem 9; note-se a facilidade de montar a matriz tridiagonal de acordo com as equações (5)–(7), e a grande expressividade de NumPy para permitir uma programação ao mesmo tempo sucinta e clara. Como veremos, Fortran possui essencialmente a mesma capacidade.

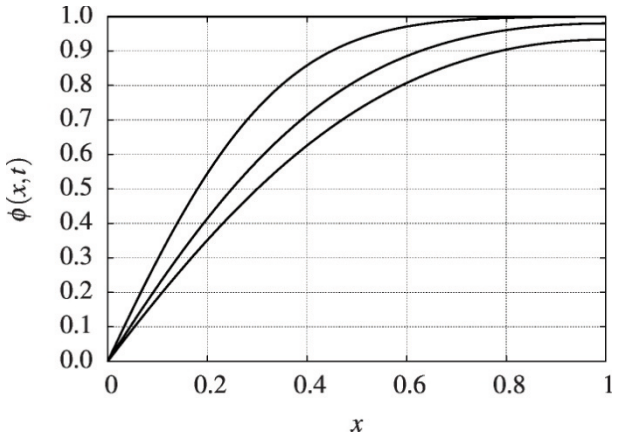
O Gráfico 1 mostra a solução para alguns valores de t. O mais interessante, porém, é comparar o *tempo* de solução quando as linhas 9–10 da Listagem 8, que aplicam Numba a **`tridag`**, são utilizadas ou não, conforme mostrado na Tabela 1. Novamente, Numba faz uma significativa diferença, acelerando o código Python a um preço muito baixo, de 2 linhas

adicionais. Além disso, ambas as linhas obedecem estritamente à sintaxe de Python. Com isso, evita-se o uso de uma sintaxe "mista" ou estendida para melhorar a performance.

Tabela 1: Tempo de processamento da solução da equação (1) em Python e Fortran, com diversas opções.

Implementação	Tempo de processamento
Python sem Numba	355.95154405
Python com Numba	6.83969092
Python com SciPy	24.99252605
Fortran sem BLAS	2.90800000
Fortran com BLAS	2.91600000

Gráfico 1: Solução por diferenças finitas da equação (1), para t = 0, t = 0,02, t = 0,04 e t = 0,06.



Listagem 9: lindad.py: programa principal

```
41 t1 = time()
42 fou = open('lindad.dat','wb')
43 nx = int(round(1.0/dx,0))
44 nt = int(round(1.0/dt,0))
45 A = zeros((3,nx-1),float)
46 Cou = 1*dt/dx
47 Fon = 2*dt/(dx*dx)
48 A[0,0] = 0.0
49 A[0,1:nx-1] = -Fon
50 A[1,0:nx-2] = 1.0 + 2*Fon + Cou
51 A[1,nx-2] = 1.0 + Fon
52 A[2,0:nx-2] = -(Cou + Fon)
53 A[2,nx-2] = 0.0
54 phi = zeros((2,nx+1),float)
55 phi[0,0:nx+1] = 1.0
56 phi[0].tofile(fou)
57 b = zeros(nx-1,float)
58 old = False
59 new = True
60 for n in range(nt):
61     b[0:nx-1] = phi[old,1:nx]
62     phi[new,1:nx] = tridag(A,b)
63     phi[new,0] = 0
64     phi[new,nx] = phi[new,nx-1]
65     phi[new].tofile(fou)
66    (old,new) = (new,old)
67 fou.close()
69 t2 = time()
69 print('%16.8f' % (t2 - t1) )
```

Uma terceira implementação é possível e de interesse aqui, com SciPy. Nesse caso, nós utilizamos uma rotina pré-compilada para resolver o sistema tridiagonal. A forma de armazenamento é *diferente* daquela utilizada por **`tridag`**: o leitor é aconselhado a procurar os detalhes em <www.scipy.org>, onde é possível encontrar as especificidades da rotina **`solve_banded`**.

A implementação correspondente, no programa, é mostrada na Listagem 10. Novamente, o tempo de processamento, bastante decepcionante, consta na Tabela 1. Nossa **`tridag`** com Numba é significativamente melhor. Portanto, nem sempre pacotes pré-compilados produzem os melhores resultados.

Listagem 10: lindas.py: Solução da equação de difusão-advecção utilizando uma rotina para solução de matrizes esparsas de SciPy

```
01 #!/usr/bin/python
02 # -*- coding: iso-8859-1 -*-
03 from __future__ import print_function
04 from __future__ import division
05 from scipy.linalg import solve_banded
06 from time import time
07 from deltas import dx,dt
08 from sys import argv, exit
09 from numpy import zeros
10 t1 = time()
11 fou = open('lindad.dat','wb')
12 nx = int(round(1.0/dx,0))
13 nt = int(round(1.0/dt,0))
14 A = zeros((3,nx-1),float)
15 Cou = 1*dt/dx
16 Fon = 2*dt/(dx*dx)
17 A[0,0] = 0.0
18 A[0,1:nx-1] = -(Cou+Fon)
19 A[1,0:nx-2] = 1.0 + 2*Fon + Cou
20 A[1,nx-2] = 1.0 + Fon
21 A[2,0:nx-2] = -Fon
22 A[2,nx-2] = 0.0
23 phi = zeros((2,nx+1),float)
24 phi[0,0:nx+1] = 1.0
25 phi[0].tofile(fou)
26 b = zeros(nx-1,float)
27 old = False
28 new = True
29 for n in range(nt):
30     b[0:nx-1] = phi[old,1:nx]
```

```
31    phi[new,1:nx] = solve_
banded((1,1),A,b)
32    phi[new,0] = 0
33    phi[new,nx] = phi[new,nx-1]
34    phi[new].tofile(fou)
35    (old,new) = (new,old)
36 fou.close()
37 t2 = time()
38 print('%16.8f' % (t2 - t1) )
```

É interessante comparar o resultado com o obtido com Fortran. As listagens 11 e 12 mostram a implementação em Fortran da solução do mesmo problema. De forma fiel às tradições da linguagem, o programa principal aparece na primeira listagem (linhas 1–44), enquanto que as sub-rotinas utilizadas – incluindo uma implementação em Fortran de **`tridag`** – aparecem na segunda listagem, linhas 45–95. É possível compilar e rodar o programa correspondente, **`linfad.f90`**, com e sem BLAS. Com as opções de compilação a seguir, obtêm-se dois programas executáveis, sem e com BLAS, respectivamente:

```
% gfortran -o3 linfad.f90 -o linfad-noblas

% gfortran -o3 -fexternal-blas linfad.f90
-lblas -o linfad-blas
```

Conforme comentamos, a sub-rotina **`tridag`** não parece ser intrinsecamente vetorizável.

Listagem 11: linfad.f90: programa principal

```
01 program linfad
02 implicit none
03 real(kind=8),dimension(:,:),&
04 allocatable::A
05 real(kind=8),dimension(:,:),&
06 allocatable::phi
07 real(kind=8),dimension(:),allocatable
::b
08 real(kind=8)::dx,dt,Cou,Fon,t1,t2
09 integer(kind=4)::k,n,nx,nt,iold,inew
10    call cpu_time(t1)
11    open(unit=1,file='linfad.dat',&
12    form='unformatted',status='replac
e',&
13    access='stream')
14    dx = 0.0001
15    dt = 0.0001
16    nx = nint(1.0/dx)
17    nt = nint(1.0/dt)
18    allocate(A(0:nx-2,0:2))
19    A = 0.0
20    Cou = 1*dt/dx
21    Fon = 2*dt/(dx*dx)
22    A(0,0) = 0.0
23    A(1:nx-2,0) = -Fon
24    A(0:nx-3,1) = 1.0 + 2*Fon + Cou
25    A(nx-2,1) = 1.0 + Fon
26    A(0:nx-3,2) = -(Cou + Fon)
27    A(nx-2,2) = 0.0
28    allocate(phi(0:nx,0:1))
29    phi(0:nx,0) = 1.0
30    write(1) phi(0:nx,0)
31    allocate(b(0:nx-2))
32    iold = 0
33    inew = 1
34    do n = 0,nt-1
35       b(0:nx-2) = phi(1:nx-1,iold)
36       call tridag(A,b,phi(1:nx-
1,inew))
37       phi(0,inew) = 0
38       phi(nx,inew) = phi(nx-1,inew)
39       write(1) phi(0:nx,inew)
40       call swap(iold,inew)
41    end do
42    close(1)
43    call cpu_time(t2)
44    write(*,'(F16.8)') t2 - t1
```

A Tabela 1 mostra os tempos de processamento obtidos com cada uma das opções. Nesse caso, em que as operações básicas de Álgebra Linear não parecem ser particularmente úteis em **`tridag`**, a opção de compilar o código Fortran com BLAS não foi vantajosa. O melhor tempo de processamento foi obtido com Fortran: a melhor implementação de Fortran é mais rápida que a melhor implementação com Python, mas da mesma ordem de grandeza. A relação dos tempos de processamento com Fortran *versus* Python "puro" (com NumPy, sem Numba) – terceira linha da Tabela 1 – ilustra um fato que é bem sabido: Fortran é cerca de cem vezes mais rápido que Python "puro". Note-se, entretanto, a drástica mudança no estado das coisas quando utilizamos Numba.

Listagem 12: linfad.f90: sub-rotinas

```
45 contains
46    subroutine swap(ix,iy)
47    integer (kind=4),intent(inout)::ix
48    integer (kind=4),intent(inout)::iy
49    integer (kind=4)::iz
50       iz = ix
51       ix = iy
52       iy = iz
53       return
54    end subroutine swap
55    subroutine tridag(A,y,x)
56    real(kind=8),intent(inout),&
57       dimension(0:,0:)::A
58    real(kind=8),intent(inout),&
59       dimension(0:)::y
60    real(kind=8),intent(inout),&
```

```
61        dimension(0:)::x
62     real(kind=8),allocatable,&
63        dimension(:)::gam
64     real(kind=8)::bet
65     integer (kind=4)::j,m,n,o
66        m = size(A,2)
67        n = size(A,1)
68        if ( m /= 3) then
69           stop("m != 3 em tridag")
70        endif
71        o = size(y,1)
72        if ( o /= n ) then
73           stop("n != 0")
74        endif
75        allocate(gam(0:n-1))
76        if (A(0,1) == 0.0) then
77           stop("Erro 1 em tridag")
78        endif
79        bet = A(0,1)
80        x(0) = y(0)/bet
81        do j = 1,n-1
82           gam(j) = A(j-1,2)/bet
83           bet = A(j,1) - A(j,0)*gam(j)
84           if (bet == 0.0) then
85              stop("Erro 2 em tridag")
86           endif
87           x(j) = (y(j) -
A(j,0)*x(j-1))/bet
88        end do
89        do j = n-2,0,-1
90           x(j) = x(j) -
gam(j+1)*x(j+1)
91        end do
92        deallocate(gam)
93        return
94     end subroutine tridag
95 end program linfad
```

7- COMPARAÇÕES ACURADAS ENTRE TEMPOS DE PROCESSAMENTO

Uma vez que os tempos de processamento apresentados até agora foram obtidos em rodadas individuais de programas (por simplicidade), faremos a seguir uma comparação breve, porém acurada, das diferenças entre Python puro (P), Python com Numba (N), e Fortran (F). Um conjunto de quatro rotinas estatísticas foi programado tanto em Python quanto em Fortran: **`stat1`** (média), **`stat2`** (média e desvio-padrão), **`covar`** (covariância) e **`reglin`** (regressão linear). **`stat1`** calcula a média de um vetor; **`stat2`** calcula a média e a variância de um vetor; **`covar`** calcula a covariância entre dois vetores; e **`reglin`** calcula uma regressão linear por mínimos quadrados entre dois vetores. Por economia de espaço, não mostramos os códigos. Todo esforço foi feito para que os algoritmos fossem os mesmos nas duas linguagens, e para que os recursos de programação também fossem os mesmos, de tal forma que a comparação fosse a mais "justa" possível.

Foram gerados dois vetores de 100.000 valores com as rotinas de geração de números aleatórios de cada linguagem, e um terceiro vetor foi produzido como uma combinação linear dos dois primeiros. O primeiro vetor foi utilizado nas rotinas **`stat1`** e **`stat2`**. O primeiro e o terceiro vetor foram utilizados nas rotinas **`covar`** e **`reglin`**. As rotinas estatísticas foram chamadas 1.000 vezes para a geração de estatísticas confiáveis, e o tempo médio de processamento e seu desvio-padrão foram calculados.

Tabela 2: Estatísticas acuradas de tempo de processamento (média ± desvio-padrão) de quatro rotinas estatísticas implementadas em Python puro (P), Python com Numba (N) e em Fortran (F).

Linguagem→ Rotina↓	P	N	F
stat1	5.0362E-02 ±7.0196E-04	1.6555E-04 ±1.9668E-05	0.4260E-03 ±0.5264E-05
stat2	2.3806E-01 ±3.5661E-03	4.1168E-04 ±5.5795E-06	0.9406E-03 0.6962E-05
covar	2.9162E-01 ±3.5048E-03	8.7996E-04 ±2.8391E-05	0.1613E-02 ±0.1752E-04
reglin	6.6030E-01 ±7.3532E-03	1.3765E-03 ±1.7173E-05	0.2537E-02 ±0.5547E-04

8- CONCLUSÕES

Neste trabalho nós apresentamos diversos exemplos de computação científica em Fortran e em Python. Os exemplos das seções 4 e 5 são muito simples, e mostram como implementar operações básicas (como por exemplo o produto de matrizes) de maneira eficiente nas duas linguagens. Na seção 6 nós mostramos um caso mais sofisticado.

Seguem-se algumas das principais conclusões:

- não é possível ainda prescindir totalmente de Fortran ou C: vale a pena manter compiladores atualizados em todos os sistemas operacionais em que for necessário trabalhar;

- todas as vezes em que operações vetoriais e matriciais forem uma parte significativa (em tempo de máquina) do programa, Fortran sem BLAS não é competitivo com Python/NumPy, *porque uma instalação competente de NumPy já vem "linqueditada" com BLAS*;
- portanto, é fundamental possuir BLAS no sistema, tão otimizado quanto possível;
- Python/NumPy/Numba possuem desempenho comparável a Fortran, pelo menos em ordem de grandeza (digamos, entre 50% e 100% da velocidade de Fortran).

É preciso, desse modo, ter uma visão de conjunto no que se refere à linguagem de programação a ser utilizada. A realidade de hoje é mais *complexa* e as demandas de aprendizado são *maiores* do que há 30 ou 40 anos. Por um lado, para a solução de problemas que exigem resposta rápida e nos quais o "gargalo" é o tempo de *desenvolvimento* do programa, ferramentas do tipo de Python são inestimáveis, pois permitem um ciclo rápido de desenvolvimento e maior facilidade em corrigir ("debugar") o código. Por outro lado, quando o problema em questão for muito intensivo de operações de ponto flutuante (tal como é o caso em muitas aplicações bi e tridimensionais de mecânica dos fluidos computacional), uma implementação competente em Fortran, provavelmente linqueditada com BLAS, será a melhor opção.

Sendo assim, precisamos ensinar *mais*, e não *menos*, opções de programação aos alunos de engenharia. A opção do autor tem sido ensinar programação científica usando Python pela facilidade de aprendizado e de implementação, mas sempre fazendo fortes referências às similaridades com Fortran: um aluno que aprende a programar competentemente em Python usando NumPy e Numba e utilizando expressões vetoriais do tipo encontrado nas linhas 48–53 da Listagem 10 não deve ter dificuldade em adaptar os seus conhecimentos para programar de forma similar em Fortran, como pode ser verificado pela similaridade entre as listagens 9 e 13 e entre as listagens 10 e 12.

REFERÊNCIAS

BACKUS, J. W.; MITCHELL, L. B.; BEEB, R. J.; NELSON, A.; BEST, S.; NUTT, R.; GOLDBERG, R. **The Fortran automatic coding system for the IBM 704 EDPM.** 590 Madison Ave, New York, NY. IBM, 1956.

DIAS, N. L. **Métodos numéricos em matemática aplicada à engenharia**. Departamento de Engenharia Ambiental, Universidade Federal do Paraná. 2013. Disponível em: <www.lemma.ufpr.br//wiki/index.php/Prof._Nelson_Luís_Dias>.

GOLUB, G. H.; VAN LOAN, C. F. **Matrix computations** (Johns Hopkins Studies in Mathematical Sciences). 3rd ed.: Johns Hopkins University Press, 1996.

LAWSON, C. L.; HANSON, R. J.; KINCAID, D.; KROGH, F. T. Basic Linear Algebra subprograms for FORTRAN usage. **ACM Trans. Math. Software**, v. 5, p. 308-323, 1979.

LUTZ, M. **Learning Python**. O'Reilly, 2008.

OLIPHANT, T. E. **Guide to Numpy**. Trelgol Publishing, 2006.

PRESS, W. H.; TEUKOLSKY, S. A.; VETTERLING, W. T.; FLANNERY, B. P. **Numerical recipes in C**. Cambridge University Press, Cambridge, UK, 1992.

DADOS DO AUTOR

Nelson Luís da Costa Dias – Possui graduação em Engenharia Civil pela Universidade Federal do Rio de Janeiro (1983), mestrado em Engenharia Civil pela Universidade Federal do Rio de Janeiro (1986), doutorado em Engenharia Civil e Ambiental pela Universidade Cornell (1993) e pós-doutorado em Física Ambiental e da Atmosfera pela Universidade da Geórgia (2007). Atualmente é do Ministério da Educação e do Desporto, sendo Professor Associado do Departamento de Engenharia Ambiental da Universidade Federal do Paraná, atuando no Curso de Graduação de Engenharia Ambiental, no Programa de Pós-Graduação em Métodos Numéricos em Engenharia e no Programa de Pós-Graduação em Engenharia Ambiental. Tem experiência nas áreas de Hidrologia e Micrometeorologia, atuando principalmente nos seguintes temas: evaporação, micrometeorologia e qualidade do ar, hidrologia, e turbulência.