Introdução à eficiência de algoritmos

[Precisão | Velocidade | Cosseno | Solução ineficiente | Solução | Solução eficiente]

Nesta seção examinaremos uma questão essencial à computação, se o algoritmo é ou não eficiente para resolver o problema de interesse. Sobre isso existem vários aspectos que podem ser considerados, aqui examinaremos dois: a **precisão** e a **velocidade**.

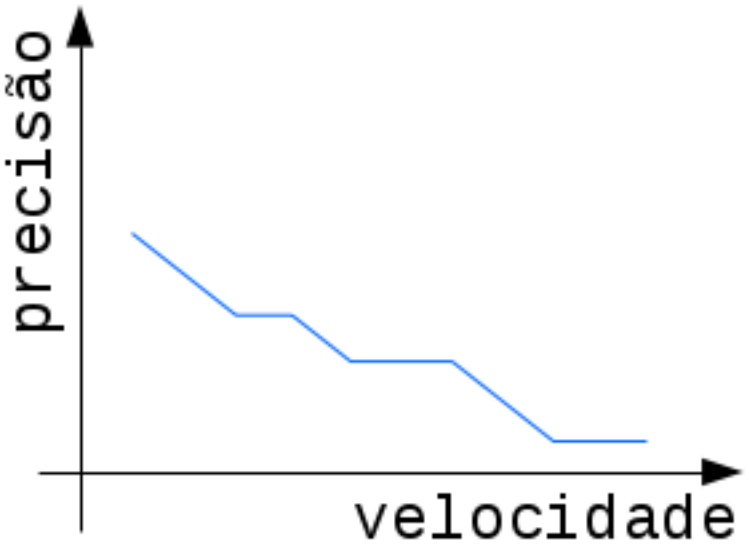


Fig. 1. Representação de que maior velocidade de execução geralmente implica em menor precisão numérica.

A **precisão** diz respeito à "qualidade" da resposta, é necessário que ela seja **suficientemente próxima da real solução** para o problema considerado. Para ilustrar podemos considerar um exemplo *numérico*, por exemplo, tentar encontrar a raíz para determinada função *f(.)*, ou seja, encontrar (se existir) algum *x* para o qual *f(x)=0*. Nesse caso, se o interesse estiver em pontos dentro do intervalo *[0, 1]*, seria inútil se o ponto devolvido tivesse um *erro maior que 0,5* (i.e., $|f(x)| > 0,5$).
Outro exemplo numérico é computar uma função transcendente como o *cosseno* de valor real. Uma vez as operações que um computador digital consegue efetivamente realizar são aquelas diretamente derivadas da soma, é preciso encontrar um algoritmo que compute uma *boa aproximação* para o *cosseno(x)*, qualquer que seja o valor real *x*. Neste texto examinaremos as implementações para as funções *seno* e *cosseno* empregando o conceito matemático da **série de Taylor**.

Quanto à **velocidade**, ela diz respeito ao *tempo que o algoritmo leva para computar a resposta*. É preciso que o tempo seja viável para o contexto, quer dizer que ao obter a resposta, essa ainda seja útil. Para ilustrar podemos considerar novamente o exemplo *numérico* da determinação da raíz da função *f(.)* (se existir, encontrar algum *x* para o qual *f(x)=0*). Digamos que o interessado seja uma industria de produção de tintas e que a raíz representa o melhor momento para interromper a produção de cada um de seus 300 pigmentos distintos. Nesse caso, se a

resposta para cada função/pigmento demorar cerca de 10 minutos, então o método empregado é inviável, pois seria *necessário cerca de 50 horas para obter as 300 raízes* (*50 x 10 min = 3000 min*).
Um exemplo de outra natureza seria o contexto de planejamentos de pousos e aterrissagens em um grande aeroporto. Após a ocorrência de algum evento não esperado, deve-se gerar uma nova escala para as próximas aterrissagens e decolagens muito rapidamente, antes que acabe o combustível de alguma aeronave em sobrevoo.

1. Problema de precisão numérica

Como discutido no texto sobre *ponto flutuante*, o problema de precisão numérica inicia-se na impossibilidade de representar valores *irracionais* e mesmo *dizimas periódicas* que precisam ser **truncados**. Mas existe outra questão de eficiência, que é a precisão numérica do algoritmo utilizado para o cômputo. Um exemplo ilustrativo é acumular valores que crescem muito rapidamente (como funções exponenciais ou fatoriais) ou dividir por valores que aproximam-se de zero, ambos podem produzir resultados "ruins": **estouro** (ou *transbordamento*).

Como citado acima, um problema computacional é conseguir implementar funções que não podem ser escritas a partir de operações elementares de soma e multiplicação, como é o caso da função *cosseno* citada anteriormente (uma função *transcendental*). Hoje, este não parece ser um problema, pois qualquer computador de bolso tem uma calculadora implementada que é capaz de computar boas aproximações para a função *cosseno*. Entretanto, isso só é possível por implementarem algum algoritmo eficiente, a partir de somas e multiplicações. No caso das *funções periódicas*, isso é feito usando a técnica matemática da *série de Taylor*. Isso será ilustrado nas tabelas 1 (implementação ineficiente) e 2 (implementação eficiente).

2. Eficiência em relação ao tempo de execução

Existem problemas que sabidamente são intratáveis no sentido de sabermos, matematicamente, que é impossível existir um algoritmo que consegue resolvê-lo (como o *problema da parada*: não existe algoritmo que possa decidir se um algoritmo qualquer pára ou após um tempo finito de execução).

Existem outros tipos de problemas que supomos **não** ser possível resolvê-los eficientemente, ou seja, todas suas soluções conhecidas são ineficientes (tempo proporcional à uma função exponencial, de base maior que a unidade). Um exemplo nessa categoria, que denominada NP-Completo, é o *problema da satisfatibilidade*: encontrar valores para cada uma de suas variáveis, que são *booleanas*, de modo a tornar a expressão lógica verdadeira, como em $f(b_1, b_2, b_3) = b_1$ *E* $(b_2$ *OU não* $b_3)$. Para esse caso, por ser uma instância pequena do problema, é fácil perceber que a resposta é *sim*, basta tomar b_1 e b_2 como *verdadeiro* (não importando o valor de b_3 pode ser *verdadeiro* ou *falso*). A solução conhecida para a *satisfatibilidade* é tentar todas as 2^n possibilidades para as n variáveis! Se o n for grande, é inviável esperar por uma resposta. Exemplo para $n=3$: testar com *f(verdadeiro,verdadeiro,verdadeiro)*, *f(verdadeiro,verdadeiro,falso)*, *f(verdadeiro,falso, verdadeiro)*, *f(verdadeiro,falso, falso)*, *f(falso, verdadeiro,verdadeiro)*, *f(falso, verdadeiro,falso)*, *f(falso, falso, verdadeiro)*, *f(falso, falso, falso)*.

Na seção seguinte examinaremos um particular problema, conseguir um valor aproximado para a função trigonométrica *cosseno*, explorando tanto a questão da eficiência numérica (precisão) quanto a de tempo de execução (velocidade).

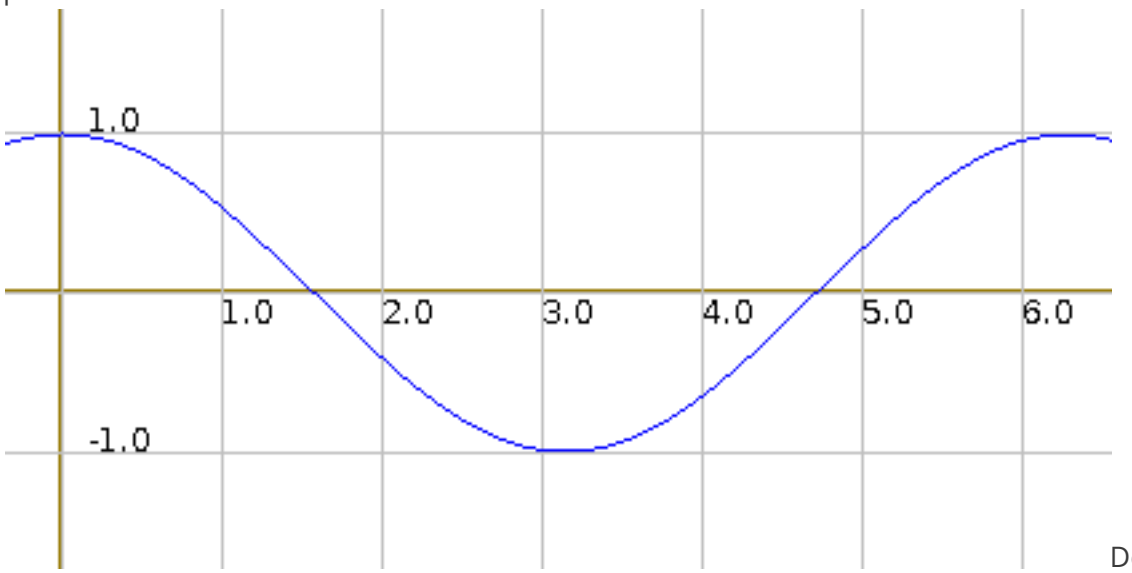
3. *Série de Taylor* e a função *cosseno*

A função *cosseno*, assim como várias outras, não pode ser computada de modo exato em um computador digital, em razão de ser uma função *transcendente* (basicamente, significa que não pode ser computada a partir de somas). Então usamos uma teoria matemática apresentada em 1715, aproximá-la por uma função polinomial que tem como propriedade ter todas as derivadas coincidindo com as derivadas da função a ser aproximada, considerando determinado ponto fixado. Esse polinômio de infinitos termos, recebeu o nome de **série de Taylor**, devido seu proponente Brook Taylor.

O *polinômio de Taylor* que aproxima a função cosseno quando aplicada ao ponto *x* é:

$$cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! + ...$$

Se estiver interessado entender a Matemática para se construir a demostração desse polinômio, estude o bloco abaixo.
Como a função *cosseno* é periódica, de período *2*Π* e além disso os valores em cada quadrante coincidem em módulo, podemos computar apenas para primeiro caso,
para *x* entre *0* e *Π/2*.



Desse modo, podemos usar um caso particular da **série de Taylor**, a séria aplicada ao ponto *x=0*, que neste caso é chamada de **série de MacLaurin**. Fixado um valor real *x* entre *0* e *Π/2*, a *série de MacLaurin* que aproxima o cosseno aplicado ao ponto *x=0*, *cos(x)*, é o polinômio (de infinitos termos) $p(x)=a_0+a_1x^1+a_2x^2+a_3x^3+...$ de tal forma que a *k*-ésima derivada de *p(0)* coincide com a *k*-ésima derivada de *cos(0)*, ou seja, *cos(0)=p(0)*, *cos'(0)=p'(0)*, *cos''(0)=p''(0)* e assim por diante.

Assim, no caso da função cosseno, escrevendo o polinômio candidato $p(x) = a_0 x^0 + a_1 x^1 + a_2 x^2 + ...$ e estabelecendo as identidades para cada derivada e usando *x=0* e sabendo que *cos(0)=1*, *sen(0)=0*, que *cos'(x)=-sen(x)* e que *sen'(x)=cos(x)=*, obtém-se

$p(0)=a_0 \Rightarrow a_0 = 1$

$p'(0)=a_1+2a_2x^1+3a_3x^2+... \mid x=0 \Rightarrow p'(0)=a_1 \Rightarrow a_1 = p'(0)= -sen(0) = 0 \Rightarrow a_1 = 0$

$p''(0)=2a_2+6a_3x^1+12a_4x^2+... \mid x=0 \Rightarrow (1/2)p''(0)=a_2 \Rightarrow a_2 = (1/2)(-cos(0)) = -1 \Rightarrow a_2 = -1/2$

$p'''(0)=6a_3+24a_4x^1+... \mid x=0 \Rightarrow (1/6)p'''(0)=a_3 \Rightarrow a_3 = (1/6)(cos'''(0)) = (1/3!)(-sen''(0)) = (1/3!)(-cos'(0)) = (1/3!)sen(0) = (1/3!)0 \Rightarrow a_3 = 0$

e assim, por diante, obtendo-se o polinômio aproximador:

$cos(x) = 1 - x^2 /2! + x^4 /4! - x^6 /6! + x^8 /8! + ...$

4. Eficiência: examinando a função transcendente *cosseno*

A implementação da aproximação da função *cosseno*, próximo do valor nulo, por *série de Taylor* produz um bom exemplo para ilustrar *ineficiência* e *eficiência* de algoritmos, quanto ao tempo de execução e quanto à precisão numérica. Uma primeira observação é que é necessário truncar a série por duas razões: é impossível computar infinitos termos; além disso a partir de determinado termo, todos os seguintes resultam zero (devido à precisão finita de qualquer computador). Portanto, devemos *truncar* a *série de Taylor*.

Analisando o polinômio de Taylor para o *cosseno*, $cos(x) = 1 - x^2 /2! + x^4 /4! - x^6 /6! + x^8 /8! + ...$, talvez um aprendiz na arte da programação perceba que poderia computar cada termo do somatório (*1*, depois $- x^2 /2!$, depois $x^4 /4!$ e assim por diante) e ir acumulando-os.
Neste caso, examinando com cuidado poderia deduzir que seu *k*-ésimo termo tem a forma: $(-1)^k x^{2k}/(2k!)$, $k=0,1,2,...$.
Ou seja, cada termo do somatório, tem em seu numerador um cálculo de exponencial (x^{2k}) e no denominador um cálculo de fatorial ($2k!$) e poderia desenhar um laço acumulando cada um desses termos.
Entretanto, essa solução tem um claro problema (que aparecerá no primeiro teste): o computo de exponencial (se maior que um) e de fatorial, ambos crescem muito rápido! Isso resultará em valores nada próximos do esperado. Isso é ilustrado abaixo.

5. Uma primeira solução nada eficiente para a função *cosseno*

A partir da série de Taylor para o cosseno, percebe-se que pode-se elaborar um laço que a cada passo computa um dos termos da série, primeiro $1=x^0/0!$, depois $x^2/2!$ e assim por diante, para um passo genérico *k* seria, $-1^k x^{2k}/(2k!)$.

Desse modo uma primeira solução, seria implementar duas funções auxiliares, uma para computar a potência (*pot*) e uma para o fatorial (*fat*), a cada passo invocando as funções auxiliares. Vamos simplificar a condição de parada, fixando a *priori* o número de termos calculados, digamos *NT=10*.

Tab. 1. Uma implementação muito ineficiente para a Série de Taylor que aproxima cosseno.

Algoritmo muito ineficiente para cômputo do cosseno	
C	**Python**

```
i = 0;
while (i<2*NT) { // Ineficiente! Nao implemente
assim!
     soma = soma + pot(-1, 2*i) * pot(x, 2*i) /
fat(2*i);
     i += 2;
     }
```

```
i = 0;
while (i<2*NT) : # Ineficiente! Nao implemente
assim!
     soma = soma + pot(-1, 2*i) * pot(x, 2*i) /
fat(2*i);
     i += 2;

  return soma;
```

Testando-se o algoritmo acima, percebe-se rapidamente que o método não é eficiente. Por exemplo, examinando a tabela 1, na qual listamos os resultados para vários valores de *x* empregando o método acima, um método eficiente (visto abaixo) e a função *cosseno* da biblioteca da linguagem. Nota-se que já na segunda linha, para *x=0.05*, o método ineficiente gera o valor *1.0012* enquanto o eficiente e a biblioteca *0.9988*, uma diferença nada desprezível. Isso vai piorando com o aumento de *x*, para o último tabelado, *x=0.45*, a diferença já está em quase *0.2*!

Ineficiência 1. Fazendo um rápido exame da solução notamos um primeiro item de ineficiência: computar *pot(-1, 2*i)* a cada passo. Isso é completamente desnecessário, pois de um passo ao outro deve-se apenas inverter o sinal, então bastaria usar uma variável `sinal`, que a cada passo fizesse `sinal = -sinal`.

Ineficiência 2. Note que a mesma ineficiência do sinal ocorre com `pot` e `fat`, pois a cada passo ignora-se que uma potência e um fatorial parcial já foi computado, calculando tudo desde o primeiro valor. Se ainda não está claro que pode ser melhorado, vamos examinar os passos `k=2` e `k=3`:
- `k=2`: computa-se `pot(x, 2*2)=pot(x, 4)` e `fat(2*2)=fat(4)`; e
- `k=3`: computa-se `pot(x, 2*3)=pot(x, 6)` e `fat(2*3)=fat(6)`.

Mas, se usarmos uma variável `pot1` para manter a potência e `fat1` para manter o fatorial, então, ao final do passo `k=2`, `pot1` estaria com o valor $x^{2k}=x^4$ e `fat1` com o valor *2k!=4!=24*.
Portanto, no passo `k=3`, bastaria fazer:

`pot1 = pot1 * x * x` e `fat1 = fat1 * 5 * 6`,

pois dessa forma, conseguiríamos

$$\texttt{pot1} \leftarrow \texttt{pot1} * x * x \Rightarrow \texttt{pot1} \leftarrow x^4\ x^2 = x^6 \quad \text{e}$$
$$\texttt{fat1} \leftarrow \texttt{fat1} * 5 * 6 \Rightarrow \texttt{fat1} \leftarrow 4! * 5 * 6 = 6!.$$

Outro problema da solução da tabela 1 é que potência e fatorial crescem muito rápido (se *x > 1*, mas se *x < 1*, então decresce muito rápido). Isso também pode resultar ineficiência numérica.

Portanto, esta solução 1 é muito ineficaz, tanto em termos de velocidade quanto de precisão. A discussão da razão de sua ineficácia já indica um caminho de melhoria, examinado a seguir.

6. Uma segunda solução para a função *cosseno*: ainda ineficaz numericamente

Uma solução um pouco melhor, mas numericamente ainda ingênua, seria eliminar a grande ineficiência citada acima, a cada passo aproveitando a potência e o fatorial anterior. Usando

como critério de parada que as diferenças entre o cômputo do termo da série seja menor que um dado limiar *Lim*, o código pode ficar como o mostrado abaixo. Suporemos a existência de uma função que devolve o módulo de um valor "flutuante", de nome *valor_absoluto*:

Tab. 2. Uma implementação não tão ineficiente para a Série de Taylor que aproxima cosseno.

Algoritmo ineficiente para cômputo do cosseno

C

```
float cossInef (float x) {
  float pot = 1, soma = 0;
  float termo0 = 2, termo = 1;
  int fat = 1, i = 1, sinal = 1;
  while (valor_absoluto(termo0-termo)>Lim) {
     soma = soma + termo;
     pot *= x * x; // compute potencia 2i de x
     fat *= (i+1) * (i+2); // compute fatorial de
2i
     termo0 = termo;
     termo = sinal * pot / fat;
     i += 2;
     sinal = -sinal;
     }
  return soma;
  }
```

Python

```
def cossInef (x) :
  pot = 1; soma = 0;
  termo0 = 2; termo = 1;
  fat = 1; i = 0; sinal = 1;
  while (valor_absoluto(termo0-termo)>Lim) :
     soma = soma + termo;
     pot *= x * x; # compute potencia 2i de x
     fat *= (i+1) * (i+2); # compute fatorial de
2i
     termo0 = termo;
     termo = sinal * pot / fat;
     i += 2;
     sinal = -sinal;
  return soma;
```

Rodando este algoritmo e comparando-o com a implementação interna das linguagens **C** e **Python**, mesmo para valores próximos de zero (como *0.1*) nota-se uma diferença significativa de *0.01* (veja a tabela abaixo onde comparamos o algoritmo acima, sua versão mais eficiente e a implementação interna da linguagem). Isso indica que o método acima **não** é eficiente.

Analisando-se o algoritmo, percebe-se que o potencial problema é que a variável *termo* é resultado da divisão de 2 valores que tem tem potencial de crescimento enorme (as variáveis *pot* e *fat* - com valor *x* próximo de zero, *pot* na verdade aproxima-se muito rápido é do zero).

Outra observação importante: a diferença entre 2 passos no laço para a variável *termo* é multiplicar por *x * x / (i+1)*(i+2)*. Assim, poderia-se tentar a cada passo multiplicar diretamente por esta "diferença" e não acumular a potência e o fatorial!

7. Uma solução eficiente para a função *cosseno*

Usando esta ideia produz-se o algoritmo seguinte que é bem mais eficiente. Nele aproveitamos para ilustrar o uso de medição de tempo de execução tanto em **C** quanto em **Python**,

Tab. 3. Uma implementação bem eficiente para a Série de Taylor que aproxima cosseno.

Algoritmo eficiente para cômputo do cosseno - comparação com o ineficiente e o interno

C

```
// Leo^nidas O. Brandao - 2017/06/19
// cos(x) = 0! x^0 - x^2 /2! + x^4 /4! - x^6 /6!
+ x^8 /8! + ...
```

Python

```
# Leo^nidas O. Brandao - 2017/06/19
# cos(x) = 0! x^0 - x^2 /2! + x^4 /4! - x^6 /6! +
x^8 /8! + ...
#        = 1 - x^2 /2! + x^4 /4! - x^6 /6! + x^8
/8! + ...
```

```c
//        = 1 - x^2 /2! + x^4 /4! - x^6 /6! + x^8
/8! + ...
// gcc -lm -o
introducao_eficiencia_algoritmos_cosseno
introducao_eficiencia_algoritmos_cosseno.c

#include <stdio.h>
#include <math.h> // cos(x)
#include <time.h> // clock_t, clock()

#define Lim 0.001

float valor_absoluto (float x) {
  if (x>=0) return x;
  return -x;
  }

// Cosseno implementado de modo ineficiente
float cossInef (float x) { ... } // copiar aqui o
codigo C do ineficiente acima

// Cosseno implementado de modo mais eficiente
float cossEfic (float x) {
  float pot = 1, soma = 0;
  float termo0 = 2, termo = 1;
  int i = 0, sinal = 1;
  while (valor_absoluto(termo0-termo)>Lim) {
     soma = soma + termo;
     termo0 = termo;
     termo *= -1 * x * x / ((i+1) * (i+2)); //
compute potencia 2i de x e fatorial de 2i
     i += 2;
     }
  return soma;
  }

int main (void) {
  float x, cosx, auxcI, auxcE;
  int i;
  clock_t tempo_inicial = clock(); // "dispara o
cronometro"...
  printf("    x | ineficiente | eficiente | math
cos(x) | dist inef. | dist efic.\n");
  x = 0.0;
  for (i=0; i<10; i++) {
    cosx = cos(x);
    auxcI = cossInef(x);
    auxcE = cossEfic(x);
    printf(" %3.2f | %11.4f | %9.4f | %11.4f
| %10.4f | %10.4f\n",
           x, auxcI, auxcE, cosx,
valor_absoluto(auxcI-cosx), valor_absoluto(auxcE-
cosx));
    x += 0.05;
    }
  clock_t tempo_final = clock();
  printf("Tempo em segundos: %f\n",
(double)(tempo_final - tempo_inicial) /
CLOCKS_PER_SEC);

  return 1;
  }
```

```python
# Invocar execucao com linha de comando: python
introducao_eficiencia_algoritmos_cosseno.py

import math; # para cos(x)
import time; # para tempo 'time.time()'

Lim = 0.001

def valor_absoluto (x) :
  if (x>=0) : return x;
  return -x;

# Cosseno implementado de modo mais eficiente
def cossInef (x) : # Copiar aqui a implementacao
de cosseno ineficiente
  ...

# Cosseno implementado de modo mais eficiente
def cossEfic (x) :
  pot = 1; soma = 0;
  termo0 = 2; termo = 1;
  i = 0; sinal = 1;
  while (valor_absoluto(termo0-termo)>Lim) :
     soma = soma + termo;
     termo0 = termo;
     termo *= -1 * x * x / ((i+1) * (i+2)); #
compute potencia 2i de x e fatorial de 2i
     i += 2;
  return soma;

def main () :
  tempo_inicial = time.time(); # "dispara o
cronometro"...
  print("    x | ineficiente | eficiente | math
cos(x) | dist inef. | dist efic.");
  x = 0.0;
  for i in range(0, 10) :
    cosx = math.cos(x);
    auxcI = cossInef(x);
    auxcE = cossEfic(x);
    print(" %3.2f | %11.4f | %9.4f | %11.4f
| %10.4f | %10.4f" %
         ( x, auxcI, auxcE, cosx,
valor_absoluto(auxcI-cosx), valor_absoluto(auxcE-
cosx) ));
    x += 0.05;

  tempo_final = time.time();
  print("Tempo em segundos: %f\n" % (tempo_final
- tempo_inicial));

main();
```

Experimente rodar em sua máquina o código acima (completando com o código da função *cossInef*, implementada na tab. 1). Na máquina usada em 2017 para o primeiro teste desse código, tanto rodando *C* quanto *Python*, produzem como resultado a tabela abaixo. Note que a medida que o *x* se afasta da origem, aparece mais erro, entretanto os erros são consistentemente maiores ao usar a implementação ingênua.

Tab. 4. As saídas geradas pelos códigos ineficientes e eficientes (da tabela 3) para aproximar cosseno.

x	ineficiente	eficiente	math cos(x)	dist inef.	dist efic.
0.00	1.0000	1.0000	1.0000	0.0000	0.0000
0.05	1.0012	0.9988	0.9988	0.0025	0.0000
0.10	1.0050	0.9950	0.9950	0.0100	0.0000
0.15	1.0112	0.9888	0.9888	0.0225	0.0000
0.20	1.0199	0.9801	0.9801	0.0399	0.0000
0.25	1.0311	0.9689	0.9689	0.0622	0.0000
0.30	1.0447	0.9553	0.9553	0.0893	0.0000
0.35	1.0606	0.9394	0.9394	0.1213	0.0000
0.40	1.0789	0.9211	0.9211	0.1579	0.0000
0.45	1.0996	0.9004	0.9004	0.1991	0.0000

Os resultados numéricos são os mesmos comparando o executável a partir do código *C* ou a interpretação do *Python*, Entretanto, os tempos de execução diferem, sendo significativamente menores utilizando a versão compilada (compilador **Gnu GCC**). Usando a mesma máquina (um PC com 2 processadores AMD Phenom II rodando Linux/Debian 8.8), os menores resultados de tempo de execução (dentre 100 testes) para o executável via *C* e o interpretador *Python*:

	C	Python
Tempo em segundos:	0.000079	0.000220

Em novos testes, desta vez usando um computador de dois núcleos com processador *Intel Core i5-6400 CPU @ 2.70GHz*, com sistema *Debian GNU/Linux 9.12*, rodando 400 vezes, eliminando-se as linhas com comando de impressão (pois as saídas poderiam uniformizar os resultados por serem significativamente mais lentas), os resultados numéricos do executável (compilação *Gnu GCCC*) mostra-se bastante superior ao interpretador *Python*. Os tempos médios das 400 rodadas, do menor tempo de execução e do maior tempo, para a versão compilada e interpretada foram (em segundos):

	C	Python
Tempo medio:	0.000520	0.008276
Tempo minimo:	0.000016	0.000076
Tempo maximo:	0.000945	0.016031

Este é apenas um resumo para apresentar o assunto de eficiência de algoritmos, seu aprofundamento demanda muito mais tempo. Por exemplo, no IME-USP existe um curso (usualmente de segundo ano) cujo objeto de estudo é precisamente discutir efeciência de algoritmos, o curso MAC0122 - Princípios de Desenvolvimento de Algoritmos. No programa de Verão do IME-USP temos o curso Tópicos de Programação.

Leônidas de Oliveira Brandão
http://line.ime.usp.br

Alterações