Um milhão de dígitos de π

Fernando Mário de Oliveira Filho
20/3/2015

1. Introdução. Muitos cálculos e ciclos de computação já foram gastos para calcular aproximações cada vez mais precisas do número π. O matemático grego Arquimedes de Siracusa (287–212 AC) foi o primeiro a descrever um método para calcular aproximações de π e o primeiro a fornecer limitantes para π (cf. Arndt e Haenel [1], seção 13.2).

O método de Arquimedes consiste em considerar um círculo de raio 1 e dois polígonos regulares, um inscrito no círculo e outro circunscrito ao círculo, como mostra a Figura 1. O perímetro do polígono inscrito é menor que o perímetro do círculo; o perímetro do polígono circunscrito é maior que o do círculo. Assim, do polígono inscrito obtemos um limitante inferior para π e do circunscrito um limitante superior.

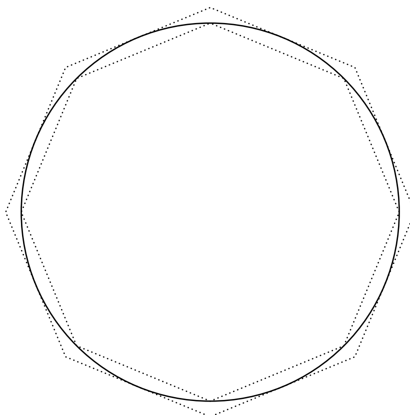

Figura 1. Círculo com octógono inscrito e circunscrito.

Para calcular sua aproximação, Arquimedes começou com hexágonos, polígonos bem compreendidos à época. Ele então considerou uma seqüência de polígonos, cada qual com o dobro de lados do anterior. Assim, partiu para dodecágonos, depois para polígonos de 24 lados, 48 lados e finalmente 96 lados. Dos polígonos de 96 lados ele obteve os limitantes

$$3.1408 \approx 3 + \frac{10}{71} < \pi < 3 + \frac{1}{7} \approx 3.1428.$$

Assim, Arquimedes obteve dois dígitos de precisão na expansão de π na base 10.

Com o advento do cálculo, várias séries foram encontradas para calcular π. Por exemplo, podemos usar a série de Taylor da função arcosseno ao redor de 0,

$$\operatorname{arcosseno} x = \sum_{k=0}^{\infty} \frac{(2k)!}{4^k (k!)^2 (2k+1)} x^{2k+1},$$

já que $\pi/2 = \operatorname{arcosseno} 1$.

Quanto mais termos da série calculamos, mais próximos chegamos de π. Você pode tentar calcular os primeiros termos e ver quantos dígitos consegue. Isaac Newton (cf. Arndt e Haenel [1], seção 13.3) usou uma modificação da série acima para calcular 15 dígitos decimais de π. Newton depois escreveu: “I am ashamed to tell you to how many figures I carried these computations, having no other business at the time”. Todos os métodos modernos para o cálculo de π são baseados em séries.

Em 1949, o ENIAC, considerado o primeiro computador digital, foi programado para calcular uma aproximação de π. Após 70 horas, foi possível calcular 2037 dígitos decimais de π. O recorde atual é de 2014, quando foram calculados 13300000000000 dígitos decimais de π; o cálculo durou 208 dias (cf. Yee [4]).

O objetivo deste exercício é gastar mais alguns ciclos de computação para calcular muitos dígitos da expansão de π na base 10. Você escreverá um programa simples capaz de calcular um grande número de casas decimais de π. Certamente seu programa será capaz de calcular 10000 casas decimais, mas quem sabe com algumas modificações não seja possível calcular 1 milhão ou 10 milhões de casas decimais.

2. Será preciso ter certa familiaridade com a forma de expressar números nas bases 2 e 16. Você já deve ter-se familiarizado com a base 2. Em base 16, ou base *hexadecimal*, temos 16 dígitos, denotados por

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f.`

O dígito `a` corresponde ao número 10 na base 10, o dígito `b` ao número 11, e assim por diante.

É muito fácil converter números da base 2 para a base 16 e vice-versa: cada dígito hexadecimal corresponde a 4 dígitos binários. Assim, na base 16 apenas agrupamos os dígitos binários em grupos de 4. Por exemplo:

Base 2	`0001`	`1111`	`0101`	`1100`
Base 16	`1`	`f`	`5`	`c`

Também é importante saber que números racionais podem ser representados em base 2 ou 16, assim como na base 10 com a notação decimal. Por exemplo, em base 16 a seqüência

`5f.2cb`

representa o número

$$5 \times 16^1 + 15 \times 16^0 + 2 \times 16^{-1} + 12 \times 16^{-2} + 11 \times 16^{-3}.$$

(Um número racional que pode ser representado exatamente na base 10 pode ser uma dízima periódica em base 16. Você consegue pensar num exemplo?)

Finalmente, é possível representar diretamente números hexadecimais num programa C. Fazemos isso como a seguir:

```
int a = 0x23f;
```

O que diz que o número está na base hexadecimal é o `0x` inicial. Note entretanto que funções de leitura como *scanf* lêem números na base 10, não na base 16. Em C, apenas números inteiros podem ser escritos na base 16 como no exemplo acima.

3. π na base 16. Algumas séries modernas para calcular π fornecem vários dígitos novos a cada termo. O grande problema é que precisamos trabalhar com números enormes para usá-las. Veja por exemplo a série da função arcosseno e o tamanho dos números envolvidos.

Neste exercício usaremos uma fórmula devida a Bailey, Borwein e Plouffe [2]. Dado $n \geq 0$, essa fórmula pode ser utilizada para calcular o n-ésimo dígito hexadecimal (depois do ponto) da expansão de π (nós começamos a contar do zero; para $n = 0$, temos o primeiro dígito depois do ponto). Mesmo para valores de n muito grandes (da ordem de bilhões) é possível fazer o cálculo usando pouca precisão, de modo que podemos usar tipos comuns de C como **int**, **long** e **double**.

A fórmula de Bailey, Borwein e Plouffe (fórmula BBP) é a seguinte:

$$\pi = \sum_{k=0}^{\infty} 16^{-k}\Big(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\Big).$$

Talvez você queira escrever um programa que calcula essa série para verificar experimentalmente que ela converge para π. Na §16 você encontra uma prova da identidade acima.

Não é imediatamente claro que a fórmula acima possa ser usada para calcular o n-ésimo dígito hexadecimal de forma eficiente. Para ver isso, temos de fazer algumas transformações.

4. Antes de começar é preciso definir algumas coisas. Seja x um número real. Denotamos por $\lfloor x \rfloor$ seu *chão*, que é o maior inteiro menor ou igual a x. Por $\mathrm{int}(x)$ denotamos a *parte inteira* de x, que é $\lfloor |x| \rfloor$. Por $\mathrm{frac}(x)$ denotamos a *parte fracionária* de x, que é igual a $|x| - \mathrm{int}(x)$. Note que nem a parte inteira nem a parte fracionária carregam o sinal de x, i.e., ambas são sempre números não-negativos. A tabela abaixo mostra alguns exemplos.

x	$\lfloor x \rfloor$	$\mathrm{int}(x)$	$\mathrm{frac}(x)$
0	0	0	0
0.2	0	0	0.2
−0.2	−1	0	0.2
3.14	3	3	0.14
−3.14	−4	3	0.14
17	17	17	0
−17	−17	17	0

Algumas propriedades da parte fracionária serão utilizadas a seguir. Em particular, se x, $y \geq 0$, então

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

Uma propriedade um pouco mais complicada é seguinte: se $x \geq y \geq 0$, então

$$\mathrm{frac}(x-y) = \mathrm{frac}(1 + \mathrm{frac}(x) - \mathrm{frac}(y)).$$

5. Se queremos calcular o n-ésimo dígito hexadecimal de π, então queremos calcular

$$\mathrm{int}(16\,\mathrm{frac}(16^n \pi)).$$

Podemos também calcular os k dígitos hexadecimais começando com o n-ésimo, calculando

$$\mathrm{int}(16^k\,\mathrm{frac}(16^n \pi)).$$

Não precisamos necessariamente operar acima com o número π. Qualquer aproximação de $\mathrm{frac}(16^n \pi)$ cujos bits mais significativos estejam corretos é suficiente. Por exemplo, para encontrar o primeiro dígito hexadecimal de π, temos

$$\mathrm{int}(16\,\mathrm{frac}(3.1415926535)) = 2 = \mathrm{int}(16\,\mathrm{frac}(3.14)).$$

Obviamente, quanto maior o número k de dígitos que queremos calcular a partir do n-ésimo, maior deve ser a precisão de $\mathrm{frac}(16^n \pi)$.

Resta então saber como calcular uma boa aproximação de $\mathrm{frac}(16^n \pi)$. Para tanto usamos a fórmula BBP. Da fórmula vemos que

$$16^n \pi = \sum_{k=0}^{\infty} 16^{n-k}\Big(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\Big).$$

Vejamos como calcular uma boa aproximação para a parte fracionária do lado direito. Para $n \geq 0$ e $c \geq 1$, defina

$$\sigma(n,c) = \sum_{k=0}^{\infty} \frac{16^{n-k}}{8k+c}.$$

Então

$$16^n \pi = 4\sigma(n,1) - 2\sigma(n,4) - \sigma(n,5) - \sigma(n,6).$$

6. Nosso problema então é calcular uma boa aproximação para $\mathrm{frac}(\sigma(n,c))$. Observe que

$$\sigma(n,c) = \sum_{k=0}^{n} \frac{16^{n-k}}{8k+c} + \sum_{k=n+1}^{\infty} \frac{16^{n-k}}{8k+c}. \tag{$*$}$$

A idéia agora é a seguinte: a primeira soma acima é uma soma finita que envolve números enormes. Mas, como estamos interessados apenas na parte fracionária, cálculos com números grandes podem ser evitados. A segunda soma é uma soma infinita, mas todos os termos são menores que 1. Não só isso: a soma converge para zero rapidamente e portanto se consideramos apenas alguns poucos termos já obtemos uma boa aproximação da parte fracionária.

Vejamos primeiro como lidar com a segunda soma. É fácil calcular um limitante superior para qualquer cauda dessa soma. De fato, para $l \geq 1$, usando a fórmula para a soma de uma progressão geométrica, temos

$$\sum_{k=n+l}^{\infty} \frac{16^{n-k}}{8k+c} \leq \sum_{k=n+l}^{\infty} 16^{n-k} = \frac{16^{-l}}{1-16^{-1}}.$$

Para $l = 10$, temos por exemplo que

$$\sum_{k=n+10}^{\infty} \frac{16^{n-k}}{8k+c} \leq \frac{1}{1030792151040} \leq 10^{-10}.$$

Assim, se consideramos apenas os 9 primeiros termos da segunda soma em $(*)$, cometemos um erro de no máximo 10^{-10}. Como estamos interessados apenas nos bits mais significativos da parte fracionária, não perdemos nada com isso. (Bem, isso não é necessariamente verdade! Veja a §14 para mais detalhes.)

A primeira soma em $(*)$ é finita, mas como dissemos os números envolvidos parecem enormes. Estamos apenas interessados na parte fracionária, logo podemos usar o seguinte fato:

$$\mathrm{frac}\Big(\frac{16^{n-k}}{8k+c}\Big) = \mathrm{frac}\Big(\frac{16^{n-k} \bmod 8k+c}{8k+c}\Big).$$

Agora, para calcular $16^{n-k} \bmod 8k+c$ podemos usar apenas números menores que $8k+c$. De fato, basta usar que para inteiros a, $b \geq 0$ e $c > 0$ temos

$$ab \bmod c = (a \bmod c)(b \bmod c) \bmod c.$$

Assim, para calcular $16^{n-k} \bmod 8k+c$ podemos acumular o resultado dos produtos, sempre reduzindo módulo $8k+c$.

Finalmente, após obter a parte fracionária da primeira soma em $(*)$ e uma boa aproximação da parte fracionária da segunda soma em $(*)$, basta somar os resultados e tomar sua parte fracionária. Conseguimos assim uma boa aproximação da parte fracionária de $\sigma(n,c)$.

Para obter uma aproximação da parte fracionária de $16^n\pi$, basta observar que $\sigma(n,c) \geq 0$ para todo n e $c > 0$ e que $\sigma(n,c) \geq \sigma(n,c')$ se $c \leq c'$. Assim, temos que

$$4\sigma(n,1) \geq 2\sigma(n,4) + \sigma(n,5) + \sigma(n,6).$$

Usando então o fato de que se $x \geq y \geq 0$ temos

$$\text{frac}(x-y) = \text{frac}(1 + \text{frac}(x) - \text{frac}(y)),$$

podemos calcular $\text{frac}(16^n\pi)$.

7. Na primeira parte do seu programa, você deve implementar uma função capaz de calcular o n-ésimo dígito hexadecimal de π, ou alguns poucos dígitos (até quatro) a partir do n-ésimo. Uma sugestão de funções a implementar é a seguinte (abaixo, usa-se **ulong** em vez de **unsigned long**, **uint** em vez de **unsigned int**, etc., por questão de espaço):

uint *mod_power*(**uint** n, **uint** k, **uint** d)

Para n, $k \geq 0$ e $d > 0$, devolve $n^k \bmod d$.

double *sigma*(**uint** n, **uint** c)

Para n, $c \geq 0$, devolve uma aproximação da parte fracionária de $\sigma(n,c)$.

uint *bbp*(**uint** n, **int** k)

Para $n \geq 0$ e $k \geq 1$, devolve os k primeiros dígitos hexadecimais da expansão de π a partir do n-ésimo dígito. Lembre-se de que k deve ser um número pequeno. Os dígitos são devolvidos como um só número inteiro, sendo que o primeiro dígito é o mais significativo e o último o menos significativo. Assim, se $k = 3$ e os três dígitos computados são `a`, `2` e `c`, a função devolve o numero `a2c`. Note que o tamanho de variáveis do tipo **uint** em seu computador efetivamente limita o valor máximo de k. Em computadores modernos, variáveis do tipo **uint** geralmente ocupam 4 bytes de memória, podendo portanto armazenar 8 dígitos hexadecimais. Para o nosso programa, usar $k \geq 5$ já seria um tanto imprudente.

Uma dica para sua implementação: use variáveis do tipo **ulong** e **double** para suas contas. Assim, você evita overflow e ganha em precisão.

8. Quanto tempo leva seu programa para $n = 1000$, 10000, 100000? Meu programa leva os seguintes tempos num processador de 2.6GHz e consegue os seguintes resultados, calculando 4 dígitos a partir do n-ésimo:

n	Dígitos	Tempo
10^4	`8ac8`	0.013s
10^5	`35ea`	0.100s
10^6	`6c65`	1.230s
10^7	`7af5`	13.858s

Se seu programa é muito mais devagar, então a culpa provavelmente está na função *mod_power*. Você deve ter implementado o algoritmo mais direto e ingênuo para calcular a exponenciação modular. Para calcular $n^k \bmod d$, esse algoritmo realiza um número de operações aritméticas proporcional a k. Para valores grandes de n, o expoente k pode ser bem grande e, como usamos freqüentemente a função *mod_power*, o tempo de execução cresce rapidamente.

Existe uma forma mais eficiente de calcular $n^k \bmod d$. Para entender como, imagine que k é escrito na base 2, ou seja, temos

$$k = a_s 2^s + a_{s-1} 2^{s-1} + \cdots + a_1 2 + a_0.$$

Note então que

$$n^k = n^{a_s 2^s} n^{a_{s-1} 2^{s-1}} \cdots n^{a_1 2} n^{a_0}.$$

Isso sugere um algoritmo para calcular n^k. O algoritmo é o seguinte:

1. Faça $pow := n$ e $ret := 1$;
2. Se $k = 0$, devolva ret;
3. Se k é ímpar, faça $ret := ret * pow$;
4. Faça $pow := pow * pow$ e $k := \lfloor k/2 \rfloor$. Vá para o passo 2.

O algoritmo acima calcula n^k. Para calcular $n^k \bmod d$, basta reduzir módulo d sempre que fazemos uma multiplicação. Assim, os números nunca ficam maiores do que $d - 1$. É preciso, entretanto, cuidar para que não ocorra overflow ao fazer um produto. A função *mod_power* recebe argumentos do tipo **uint**. Se você utilizar internamente variáveis do tipo **ulong**, não deve ter problemas com overflow, já que uma variável do tipo **ulong** deve conseguir guardar o produto de duas variáveis do tipo **uint** sem problemas.

O algoritmo ingênuo executa um número de operações aritméticas proporcional a k. O número de iterações do algoritmo acima, para $k > 0$, é igual ao número de bits de k e, para cada iteração, executamos um número constante de operações aritméticas. O número de bits de k é igual a $1 + \lfloor \lg k \rfloor$, onde $\lg k$ é o logaritmo de k na base 2. Esse número é muito menor que k.

9. π na base 10. Calcular π na base 16 é interessante, mas quando pensamos em aproximações de π pensamos imediatamente na base 10. Assim, depois de calcular muitos dígitos hexadecimais de π, você deve querer transformá-los em dígitos decimais. Felizmente não é muito difícil fazer isso, embora agora você precise trabalhar com números grandes.

A idéia fundamental é a seguinte. Suponha que você tenha calculado os primeiros n dígitos hexadecimais de π, ou seja, você tem o seguinte número inteiro imenso:

$$a = \mathrm{int}(16^n \, \mathrm{frac}(\pi)).$$

Suponha que você queira os primeiros m dígitos decimais de π. Se $m \leq n$, então você quer expressar o número

$$\mathrm{int}(10^m a / 16^n)$$

na base 10. Para tanto precisamos saber (i) guardar números bem grandes, potencialmente com milhões de dígitos hexadecimais, na memória; (ii) multiplicar esses números por potências de 10; (iii) dividir esses números por potências de 10 e potências de 16. Vejamos como executar essas tarefas.

Uma forma simples de guardar um número grande na memória é usar vetores de variáveis inteiras. O tamanho de uma variável **uint** varia de computador para computador. Você pode descobrir o tamanho de um tipo de dado usando o operador **sizeof**. Tente executar por exemplo o seguinte código em seu computador:

```
printf("%lu %lu", sizeof(uint), sizeof(ulong));
```

Em meu computador, a saída é `4 8`, indicando que uma variável do tipo **uint** ocupa 4 bytes e uma do tipo **ulong** ocupa 8 bytes. Esses são os tamanhos atualmente usuais.

Cada byte contém *pelo menos* 8 bits. O número de bits num byte pode variar de sistema para sistema, embora na grande maioria dos computadores modernos 1 byte contenha 8 bits. Para saber quantos bits há num byte, você pode verificar o valor de `CHAR_BIT`, que é uma macro definida no arquivo `limits.h`.

Suponhamos daqui em diante que **uint** ocupe 4 bytes, que **ulong** ocupe 8 bytes e que 1 byte contenha 8 bits. Então o maior número que uma variável do tipo **uint** pode guardar é $2^{32} - 1$. Assim, podemos usar um vetor de variáveis **uint** para representar um número na base 2^{32}.

Por exemplo, o vetor

```
uint a[1000]
```

pode guardar um número com 1000 dígitos na base 2^{32}, ou em outras palavras um número de 32000 bits ou 8000 dígitos hexadecimais. Nossa convenção será a seguinte: o dígito menos significativo do número é guardado na posição 0, o seguinte na posição 1, e assim por diante, de modo que o dígito mais significativo esteja na última posição diferente de zero.

10. Para lidar com números grandes, precisaremos saber acessar bits individuais de variáveis do tipo **uint** ou **ulong**, por exemplo. Para tanto, a linguagem C oferece operadores bit-a-bit, ou *bitwise operators*, e operadores de deslocamento, ou *shift operators*.

Precisamos de três dos operadores bit-a-bit, o operador `&`, ou *bitwise and*, o operador `|`, ou *bitwise or*, e o operador $\sim$, ou *bitwise negation*.

Se a e b são números, o resultado de $a \,\&\, b$ é assim obtido. Alinham-se os bits de a e b. Cada bit do resultado é igual a 1 se e somente se ambos os bits correspondentes em a e b forem 1. O resultado de $a \mid b$ é obtido de forma semelhante: cada bit do resultado é 0 se e somente se ambos os bits correspondentes em a e b forem 0. Finalmente, o resultado de $\sim a$ é o número no qual cada bit tem o valor inverso de seu bit correspondente em a. Por exemplo:

a	0101
b	1101
$a \,\&\, b$	0101
$a \mid b$	1101
$\sim a$	1010

O operador `<<` é o *left shift*. Se a é um número e $k \geq 0$, então $a \ll k$ é o número obtido de a adicionando-se k zeros à esquerda. O operador `>>` é o *right shift*. O número $a \gg k$ é obtido desprezando-se os k bits menos significativos de a. Por exemplo:

a	001001
$a \ll 2$	100100
$a \gg 2$	000010

Com isso, você já pode imaginar como guardar o número $\text{int}(16^{8n}\,\text{frac}(\pi))$. Podemos fazê-lo da seguinte forma:

```
uint a[MAX_DIGITS];
int i = 0, j;
for (n--; n >= 0; n--) {
   a[n] = 0;
   for (j = 0; j < 8; j++)
      a[n] = (a[n] << 16) + bbp(i++, 1);
}
```

Este pequeno trecho calcula um dígito de cada vez. Cada posição do vetor a carrega 8 dígitos hexadecimais. Observe também que o primeiro dígito é o mais significativo, logo o colocamos por último no vetor. Acima, supomos que `MAX_DIGITS` é o número máximo de dígitos comportados. Esse pode ser um valor fixo, digamos 10^7, mas deve ser suficiente para que o vetor a comporte os resultados das operações que desejamos realizar.

11. Como uma sugestão, para converter números grandes para a base 10 você pode implementar as seguintes funções.

int *multiply*(**int** $*n$, **uint** $*a$, **uint** b)

A função recebe um número na base 2^{32} contido em a. O dígito mais significativo desse número está na posição $*n$, ou seja, é $a[*n]$. A função substitui o número em a por seu produto pelo número b e coloca em $*n$ a posição do dígito mais significativo do produto. Ela devolve 0 se ocorreu overflow, ou seja, se o número resultante teria mais de `MAX_DIGITS` dígitos; caso contrário devolve 1.

uint *divide*(**int** $*n$, **uint** $*a$, **uint** b)

Recebe um número na base 2^{32} como a função *multiply*. Substitui a pelo quociente da divisão de a por b e coloca em $*n$ a posição do dígito mais significativo do quociente. Devolve o resto da divisão.

void *print_base_10*(**int** n, **uint** $*a$)

Imprime o número em a, cujo dígito mais significativo está na posição n, na base 10.

Para você entender melhor como as funções acima podem ser implementadas, vejamos uma implementação da função *multiply*. A função *divide* você deve implementar sozinho, depois de aprender mais adiante o algoritmo de divisão.

O algoritmo para multiplicar um número a por um número b composto por um único dígito é bem simples. Nós multiplicamos cada dígito de a, do menos significativo para o mais significativo, por b. Por exemplo, na base 16, temos:

```
  67
×  2
----
  ce
```

No exemplo acima, cada multiplicação resultou num número de um só dígito, portanto o resultado. Nem sempre acontece assim; por exemplo, podemos ter

```
   1
  67
×  3
----
 135
```

Aqui temos que $3 \times 7 = 15$. Assim, temos o efeito "vai um", ou *carry*. O dígito mais significativo do produto avança para o próximo produto e é somado ao resultado. A multiplicação termina quando fazemos $3 \times 6 + 1 = 13$.

Você deve ser capaz de entender a seguinte implementação da função *multiply*:

```
int multiply(int *n, uint *a, uint b) {
  int i;
  ulong tmp, carry = 0;
  for (i = 0; i <= *n; i++) {
    tmp = (ulong) b * a[i] + carry;
    a[i] = tmp & 0xffffffff;
    carry = tmp >> 32;
  }
  if (carry && *n == MAX_DIGITS - 1) return 0;
  if (carry) a[++(*n)] = carry;
  return 1;
}
```

A implementação acima supõe que uma variável do tipo **uint** contenha 32 bits e que uma variável do tipo **ulong** contenha 64 bits. Na prática, para que sua implementação seja independente do hardware, você deve dar uma olhada no arquivo `limits.h`, que

define macros contendo o maior valor que pode ser colocado numa variável do tipo **uint**, etc.

O ponto mais importante é entender como lidar com o *carry*. Como **ulong** tem o dobro do tamanho de **uint**, quando fazemos a atribuição de *tmp* não ocorre overflow. Daí colocamos em $a[i]$ o dígito menos significativo do resultado, usando para isso uma operação *bitwise and*. Depois, *carry* torna-se o dígito mais significativo do resultado.

Ao final, se *carry* é não-zero, então a ganhou um dígito a mais. Se não há mais espaço para guardar esse dígito, devolvemos 0 para indicar overflow. Se há espaço, guardamos o dígito e aumentamos o número de dígitos em 1.

12. A implementação de *multiply* acima deve servir para você se acostumar com a forma de representar números e como operar com eles. Vamos ver como funciona o algoritmo de divisão; depois, você deverá implementar a função *divide* sozinho.

O algoritmo de divisão é muito simples na base 2. Suponha que temos um número representado na base 2,

$$a = a_s 2^s + \cdots + a_1 2 + a_0.$$

Queremos dividir esse número por b. Para $k > 0$, suponha que conheçamos o quociente q e o resto r da divisão de

$$2^{-k}(a_s 2^s + \cdots + a_k 2^k)$$

por b. Nosso objetivo é encontrar o quociente q' e o resto r' da divisão de

$$2^{-(k-1)}(a_s 2^s + \cdots + a_{k-1} 2^{k-1})$$

por b.

Como

$$2^{-k}(a_s 2^s + \cdots + a_k 2^k) = qb + r,$$

temos que

$$2^{-(k-1)}(a_s 2^s + \cdots + a_{k-1} 2^{k-1}) = 2qb + 2r + a_{k-1}.$$

Se $2r + a_{k-1} < b$, então $q' = 2q$ e $r' = 2r + a_{k-1}$. Senão, então note que como $r < b$, temos que $2r + a_{k-1} < 2b$. Podemos escrever

$$2qb + 2r + a_{k-1} = (2qb + b) + (2r + a_{k-1} - b) = (2q + 1)b + (2r + a_{k-1} - b).$$

Como $2r + a_{k-1} - b < b$, vemos que $q' = 2q + 1$ e $r' = 2r + a_{k-1} - b$.

Isso sugere o seguinte algoritmo para calcular o quociente q e o resto r da divisão de a por b.

1. Faça $q := r := 0$;
2. Percorra os bits de a do mais significativo para o menos significativo. Ao processar o i-ésimo bit de a, faça $r := 2r + s$, onde s é o i-ésimo bit de a. Se $r \geq b$, então faça $r := r - b$ e troque o i-ésimo bit de q para 1.

Você pode usar esse algoritmo em sua implementação da função *divide*. A função será usada para dividir por 10 repetidamente, mas como o dividendo fornecido para a função é um **uint**, podemos dividir por potências maiores de 10, economizando assim algum trabalho.

Também precisamos dividir números por 16, mas isso é muito mais fácil e para tanto não precisamos da função *divide*: basta fazer um shift do número correto de bits.

13. Para calcular m dígitos decimais de π, não é necessário ter calculado m dígitos hexadecimais, mas um número menor.

Um número inteiro $a > 0$ requer $1 + \lfloor \log_b a \rfloor$ dígitos para ser representado na base b. Suponha agora que temos um número a de n dígitos quando representado na base b. Para representá-lo na base c precisamos de

$$1 + \lfloor \log_c a \rfloor \geq \frac{\log_b a}{\log_b c} \geq \frac{n-1}{\log_b c}$$

dígitos. Portanto, se queremos m dígitos de π na base 10, é suficiente calcular

$$1 + m \log_{16} 10 \leq 1 + 0.84m.$$

14. Erros e verificação. Foi dito anteriormente que se cometemos um erro pequeno, digamos da ordem de 10^{-10}, ao calcular a segunda soma em $(*)$, então esse erro não afeta os bits mais significativos do resultado. Isso não é necessariamente sempre verdade. Por exemplo, quando somamos 10^{-10} ao número 0.0999999999, obtemos 0.1 e vemos que o resultado da soma foi propagado até o dígito mais significativo. Isso poderia acontecer em nossa implementação, invalidando o resultado.

Um programa bem feito tentaria verificar se esse tipo de efeito ocorre. Uma forma simples de aumentar nossa confiança nos resultados obtidos é a seguinte. Para $n \geq 1$, podemos calcular o n-ésimo dígito hexadecimal de π de duas formas. Podemos tanto fazer a chamada $bbp(n, 1)$ quanto fazer a chamada $bbp(n-1, 2)$ e considerar o dígito hexadecimal menos significativo do resultado. As operações aritméticas executadas por essas chamadas são completamente diferentes, portanto se os dígitos calculados forem os mesmos podemos ter um alto grau de confiança no resultado.

15. Paralelização. Seu programa calcula alguns dígitos hexadecimais de π de cada vez, mas os cálculos de grupos de dígitos diferentes são totalmente independentes. Por isso, é muito fácil paralelizá-lo. Você pode escrever um programa que calcule todos os dígitos num determinado intervalo fornecido pelo usuário e grave o resultado num arquivo. Daí é possível executar o programa diversas vezes, para intervalos diferentes, em computadores diferentes. Na verdade, você pode executar seu programa várias vezes no mesmo computador. Se seu computador possui n cores e você executar seu programa n vezes ao mesmo tempo, cada execução será feita num core diferente em paralelo. Depois, tudo o que você precisa fazer é juntar os resultados para transformar o número da base 16 para a base 10, uma operação que não é tão fácil de paralelizar.

Também é interessante observar que calcular o n-ésimo dígito toma mais tempo quanto maior for n. Isso pode influenciar a forma como você vai dividir as tarefas entre processadores diferentes, de modo a obter uma divisão equilibrada.

16. Uma prova da fórmula BBP. O interessante sobre a fórmula BBP é como ela foi encontrada, não exatamente a prova de que ela é verdadeira. Mas vejamos a prova de qualquer maneira, já que ela é simples de entender para quem conhece um pouco de cálculo.

É um exercício não muito complicado de integração mostrar que

$$\pi = \int_0^{1/\sqrt{2}} \frac{4\sqrt{2} - 8x^3 - 4\sqrt{2}x^4 - 8x^5}{1 - x^8}\, dx. \tag{$**$}$$

Você pode tentar calcular a primitiva da função acima, ou usar um software de álgebra computacional como o Sage [3] para fazê-lo.

Uma vez convencido da identidade acima, note que a expansão de Taylor para $1/(1-x^8)$ ao redor de 0 é

$$1 + x^8 + x^{16} + x^{24} + \cdots = \sum_{k=0}^{\infty} x^{8k}.$$

Trocando $1/(1-x^8)$ em $(**)$ pela série acima obtemos

$$\pi = \int_0^{1/\sqrt{2}} (4\sqrt{2} - 8x^3 - 4\sqrt{2}x^4 - 8x^5) \sum_{k=0}^{\infty} x^{8k}\, dx.$$

Agora, podemos quebrar a integral acima em quatro integrais e trocar a soma infinita com a integral (pois a série de Taylor converge absoluta e uniformemente em $[0,1)$). Para o primeiro termo, obtemos

$$\int_0^{1/\sqrt{2}} 4\sqrt{2} \sum_{k=0}^{\infty} x^{8k}\, dx = \sum_{k=0}^{\infty} 4\sqrt{2} \int_0^{1/\sqrt{2}} x^{8k}\, dx = 4 \sum_{k=0}^{\infty} \frac{16^{-k}}{8k+1}.$$

Fazendo o mesmo com as outras três integrais obtemos $(*)$.

17. Acúmulo de erros. Olhe novamente para a fórmula

$$\sigma(n,c) = \sum_{k=0}^{n} \frac{16^{n-k}}{8k+c} + \sum_{k=n+1}^{\infty} \frac{16^{n-k}}{8k+c}.$$

Suponha que para $n > 0$ nós já tenhamos calculado $\sigma(n,c)$ e que portanto conheçamos o valor

$$x = \operatorname{frac}\Bigl(\sum_{k=0}^{n} \frac{16^{n-k}}{8k+c}\Bigr).$$

Mas então

$$\operatorname{frac}\Bigl(\sum_{k=0}^{n+1} \frac{16^{(n+1)-k}}{8k+c}\Bigr) = \operatorname{frac}\Bigl(16x + \frac{1}{8(n+1)+c}\Bigr).$$

Isso sugere um algoritmo mais simples para calcular a parte fracionária de $\sigma(n,c)$, sem usar exponenciação modular: basta guardar valores já computados e aplicar a fórmula acima. Você deve tentar fazer isso e ver até que valor de n você obtém resultados corretos. O problema da abordagem acima é a multiplicação por 16. Nós não conhecemos x, mas sim uma aproximação de x com um certo número de bits significativos. Agora, os 4 bits menos significativos de $16x$ são iguais a zero. Assim, perdemos em precisão a cada multiplicação. Os erros então se acumulam até que obtemos resultados totalmente errados.

18. Bibliografia.

[1] J. Arndt e C. Haenel, π *Unleashed*, Springer-Verlag, Berlin, 2001.

[2] D. Bailey, P. Borwein e S. Plouffe, On the rapid computation of various polylogarithmic constants, *Mathematics of Computation* 66 (1997) 903–913.

[3] W. Stein *et al.*, Sage Mathematics Software (Version 6.3), The Sage Development Team, 2014, `http://www.sagemath.org`.

[4] A.J. Yee, `http://www.numberworld.org/y-cruncher`, 2015.

F.M. de Oliveira Filho
Instituto de Matemática e Estatística
Universidade de São Paulo
`fmario@gmail.com`