Esperienze di Programmazione
Algoritmo BBP per π

Antonio Pitasi

03/06/2017

1 Introduzione al problema

In questo progetto, si cerca di dare una buona implementazione della formula di Bailey–Borwein–Plouffe, che modificata in modo appropriato, consente di calcolare l'n-esima cifra esadecimale di π senza conoscerne le precedenti.
La formula originale, denominata *BPP formula*, è la seguente:

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

Pur rappresentando un buon metodo per il calcolo di π, non è il più veloce conosciuto oggi (record attualmente detenuto dall'algoritmo dei fratelli Chudnovsky). Tuttavia, come già accennato, sarà possibile ricavarne un modo per calcolare una cifra arbitraria di π. Un vantaggio di questo approccio, consiste nella possibilità di avere macchine che calcolano cifre diverse parallelamente, poiché il risultato di un calcolo non influenza gli altri.

La formula scoperta da Plouffe è stata ispirata da una formula già nota, anch'essa in forma di sommatoria:

$$\ln(2) = \sum_{k=1}^{\infty} \frac{1}{2^k \cdot k}$$

All'interno del programma viene implementata anche quest'ultima, utilizzabile per calcolare l'n-esima cifra binaria di $\log(2)$, ancora una volta senza conoscerne le precedenti.

2 Algoritmi

2.1 Il calcolo di log(2)

Borwein e Plouffe osservarono come partendo dalla formula per il calcolo di $\log(2)$ potessero calcolarne le cifre binarie a partire da una posizione arbitraria.

Riprendendo quindi la formula scritta precedentemente:

$$\ln(2) = \sum_{k=1}^{\infty} \frac{1}{2^k \cdot k}$$

Vogliamo calcolare le cifre *binarie* che partono alla posizione $d+1$. Si può notare come questo sia equivalente a calcolare $\{2^d \cdot \log(2)\}$, dove $\{\}$ indica la parte frazionaria[1].

Quindi possiamo scrivere:

$$\begin{aligned}
\left\{2^d \cdot \log(2)\right\} &= \left\{\sum_{k=1}^{\infty} \frac{2^{d-k}}{k}\right\} \\
&= \left\{\left\{\sum_{k=1}^{d} \frac{2^{d-k}}{k}\right\} + \sum_{k=d+1}^{\infty} \frac{2^{d-k}}{k}\right\} \\
&= \left\{\left\{\sum_{k=1}^{d} \frac{2^{d-k} \bmod k}{k}\right\} + \sum_{k=d+1}^{\infty} \frac{2^{d-k}}{k}\right\}
\end{aligned}$$

Nel primo passaggio la sommatoria è stata divisa in due sommatorie per separare la prima che possiede una parte intera, dalla seconda che invece è sicuramente minore di 1.[2]

Nel secondo passaggio è stato aggiunto *mod k*, giustificato dal fatto che ci interessa solo la parte frazionaria del quoziente della divisione per k.

Adesso si può notare come i termini della seconda sommatoria diventino piccoli molto in fretta, quindi a seconda della precisione di macchina disponibile, bastano poche iterazioni per un'approssimazione sufficiente a trovare la cifra in posizione d+1, ma anche qualcuna delle successive.

Per l'implementazione si rimanda alla *Sezione 3*.

2.2 Il calcolo di Pi Greco

Dopo essere riusciti con log(2), Borwein e Plouffe sono andati alla ricerca di formule analoghe che funzionassero con altre costanti. Ed è così che hanno trovato la formula per il calcolo di π:

$$\begin{aligned}
\pi &= \sum_{k=0}^{\infty} \left[\frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{5}{8k+5} - \frac{1}{8k+6}\right)\right] \\
&= 4 \cdot S_1 - 2 \cdot S_4 - S_5 - S_6, \\
S_j &= \sum_{k=0}^{\infty} \frac{1}{16^k \cdot (8k+j)}
\end{aligned}$$

[1] La definizione di parte frazionaria è: $\{x\} = x - \lfloor x \rfloor$.

[2] Questo è dato dal fatto che $2^{d-k} < 1$ per $k > d$.

Dove la sommatoria viene divisa in quattro sommatorie più piccole permettendo una notazione più compatta.

Da qui si può fare un ragionamento molto simile al precedente: per calcolare la cifre *esadecimali* che partono alla posizione $d+1$, occorre calcolare $\{16^d \cdot \pi\}$, ricordando che $\{\}$ indica la parte frazionaria. Ovvero:

$$\{16^d \pi\} = \{4 \cdot \{16^d S_1\} - \{2 \cdot \{16^d S_4\}\} - \{16^d S_5\} - \{16^d S_6\}\}$$

Per calcolare in modo efficiente gli addendi di questa somma, applichiamo le stesse trasformazioni fatte nel paragrafo precedente:

$$\begin{aligned}\{16^d S_j\} &= \left\{ \left\{ \sum_{k=0}^{d} \frac{16^{d-k}}{8k+j} \right\} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j} \right\} \\ &= \left\{ \left\{ \sum_{k=0}^{d} \frac{16^{d-k} \bmod 8k+j}{8k+j} \right\} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j} \right\}\end{aligned}$$

Si nota subito la somiglianza con il risultato trovato per $\log(2)$, con la sola differenza che adesso andranno calcolati quattro diversi risultati per poi combinarli opportunamente insieme.

Il calcolo della prima sommatoria potrebbe sembrare costoso visto l'esponente molto grande al numeratore, ma è reso possibile dall'operazione di *modulo*. Infatti viene utilizzato un algoritmo ottimizzato per occupare poco spazio, spiegato nella sezione *Sezione 3.1*.

3 Implementazione degli algoritmi

3.1 Elevamento a potenza in modulo

Operazioni come $b^e \bmod k$, vengono calcolate utilizzato un algoritmo noto con il nome di *Right-to-Left binary exponentiation*[2, p. 463].

Non è possibile calcolare prima b^e come verrebbe spontaneo, perché potrebbe essere un numero troppo grosso da rappresentare. Per superare questo problema basta notare come:

$$c = (a \cdot b) \bmod m$$

sia uguale a

$$c = (a \bmod m \cdot b \bmod m) \bmod m$$

Per cui ad ogni iterazione salveremo solo il modulo del risultato intermedio, e non tutto il prodotto.

Adesso per ottimizzare il numero di operazioni necessarie possiamo scrivere e in notazione binaria:

$$e = \sum_{i=0}^{n-1} a_i 2^i$$

Dove a_i può essere 0 o 1, $a_{n-i} = 1$, e n è il numero minimo di bit necessario per rappresentare e.

Quindi:

$$b^e = \prod_{i=0}^{n-1} (b^{2^i})^{a_i}$$

Il programma finale si può scrivere in pseudocodice in questo modo:

```
function ModPow(b, e, k)
    if k = 1 then return 0
    result := 1
    b := b mod k
    while e > 0
        if (e mod 2 == 1):
            result := (result * b) mod k
        e := e / 2
        b := (b * b) mod k
    return result
```

3.2 Altre scelte implementative

La quantità di cifre calcolate a partire della posizione specificata, dipende dalla precisione della macchina utilizzata. In particolare Bailey stesso ha scritto che utilizzando aritmetica a 64 bit si hanno almeno nove cifre corrette, mentre con aritmetica a 128 bit se ne hanno almeno ventiquattro.[1, p. 4-5]

Nel programma presentato vengono utilizzati numeri in virgola mobile da 128 bit, implementati mediante una libreria.

4 Test e verifica della correttezza

Il programma è stato sviluppato seguendo il modello del *Test Driven Development*, questo significa che sono stati realizzati i test per ogni singola funzione prima della funzione stessa.

Ogni file sorgente è accompagnato da un secondo file contenente i relativi test.

La maggior parte della suite di test utilizza la tecnica dell'oracolo, in pratica viene confrontato l'output della funzione implementata con risultati noti a priori.

Per esempio nel calcolo dell'elevamento a potenza in modulo vengono calcolati:

$$2^{100} \bmod 2 = 0$$

$$50^{19800} \bmod 6 = 4$$

$$367^{447} \bmod 739 = 687$$

I risultati corretti sono stati trovati utilizzando il calcolatore online *Wolfram Alpha*[3].

Per quanto riguarda invece il calcolo vero e proprio delle cifre di log(2) e π, oltre all'oracolo, sono stati effettuati anche test sull'*overlapping* delle cifre.

Questo significa che vengono calcolate le prime otto cifre di π: 243F6A88, e le cifre a partire dalla quinta: 6A8885A3. Si verifica poi che ci sia corrispondenza tra le quattro finali del primo calcolo e le quattro iniziali del secondo calcolo.

Più in generale, questo metodo è utilizzabile anche per capire quante cifre successive alla posizione desiderata sono corrette.

Oltre a quelle già descritte, viene eseguita anche una semplice funzione per verificare di stare usando veramente numeri in virgola mobile da 128 bit. Il test consiste nel calcolare $\frac{1}{49} \cdot 49$ e verificare che il risultato sia 1.

Su sistemi a 64 bit fallirebbe poiché $\frac{1}{49}$ non è rappresentabile senza errore.

5 Risultati

Vengono riportati alcuni risultati di esempio ottenuti eseguendo il programma:

```
Binary digits of log2 starting at position 1: 10110001
Binary digits of log2 starting at position 100: 00011111
Binary digits of log2 starting at position 10000: 00101101
Binary digits of log2 starting at position 1000000: 11010100
Binary digits of log2 starting at position 100000000: 01100111
```

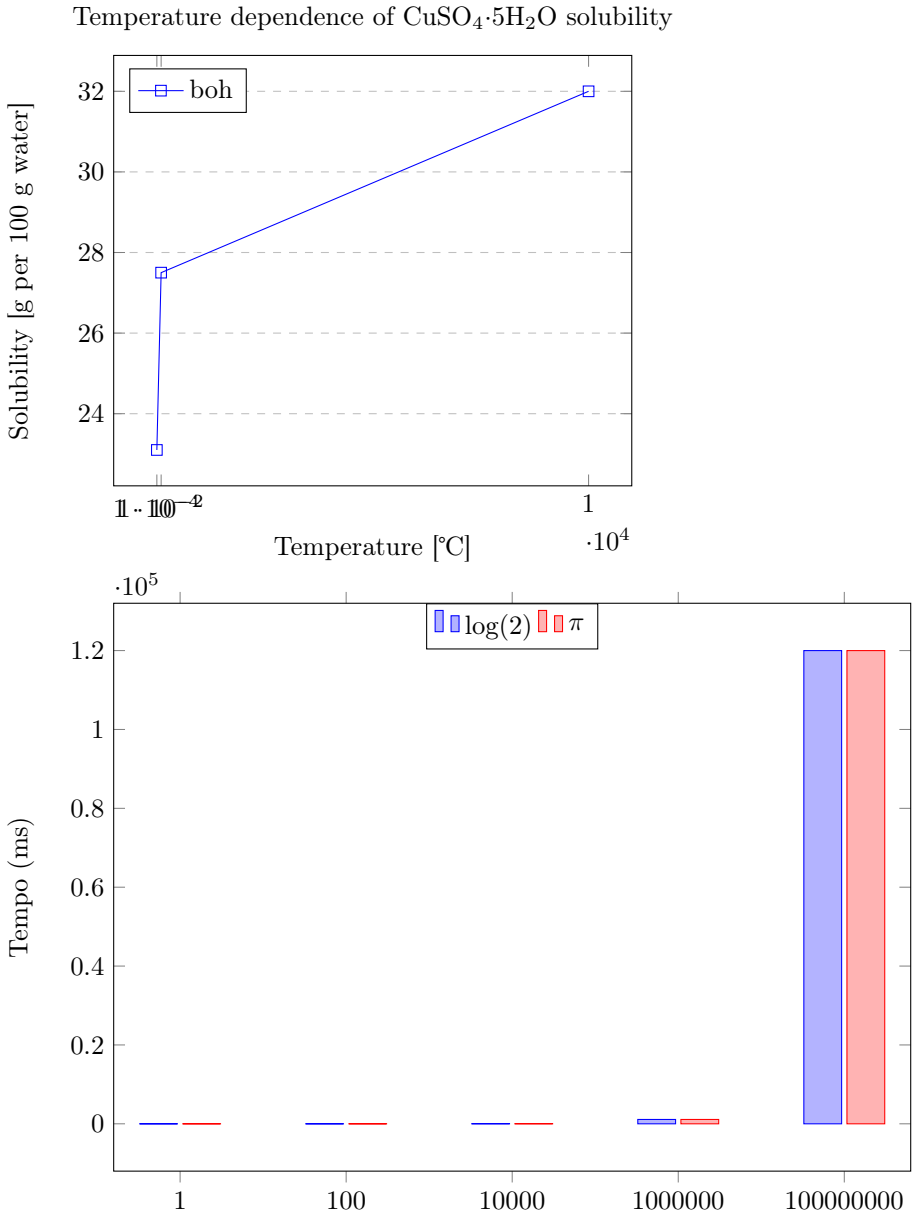
```
Hexadecimal digits of pi starting at position 1: 243F6A88
Hexadecimal digits of pi starting at position 100: C29B7C97
Hexadecimal digits of pi starting at position 10000: 68AC8FCF
Hexadecimal digits of pi starting at position 1000000: 26C65E52
Hexadecimal digits of pi starting at position 10000000: 17AF5863
```

5.1 Tempo di computazione

Sperimentalmente si può vedere come il tempo di calcolo sia piuttosto lineare rispetto a n.

Infatti i tempi necessari per ricavare i risultati precedenti, sul mio laptop, sono rappresentati nel grafico che segue:

[3]Link: http://www.wolframalpha.com/

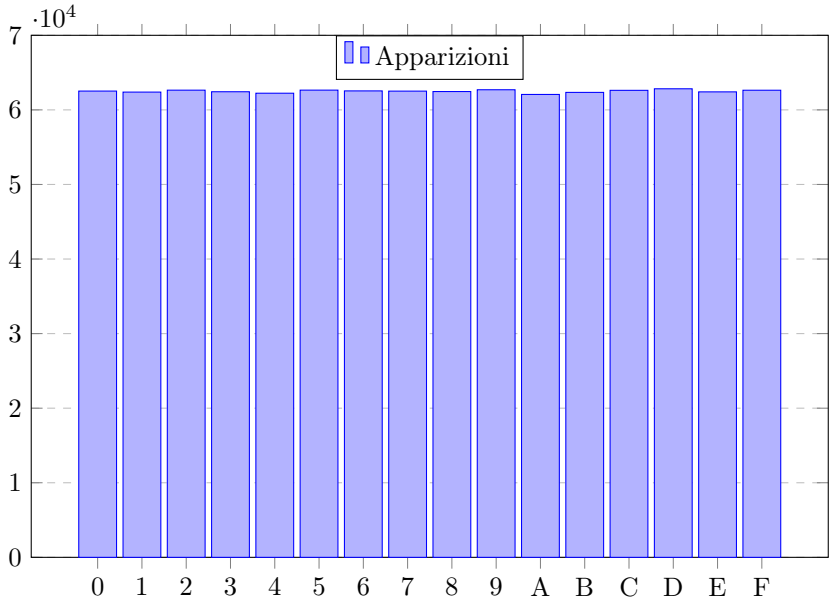


5.2 Possibili (futuri) utilizzi

Partendo dal fatto che le cifre in π **sembrano**[4] comparire equiprobabilmente, si potrebbe dedurre che prima o poi sia possibile incontrare qualunque sequenza di cifre.

[4]Non è stato dimostrato per il momento.

Per questo motivo si può pensare di codificare un file, che non è altro che una sequenza di cifre, *all'interno* di π. Sarebbe sufficiente memorizzare solo la posizione della cifra iniziale e la sua lunghezza, recuperarlo usando il calcolo dell'n-sima cifra sarebbe poi *relativamente facile*.
Sebbene per il momento il calcolo richiederebbe troppo tempo rendendo questa tecnica inutilizzabile, riporto i dati ottenuti sulla frequenza di comparsa di ciascuna cifra all'interno del primo milione di cifre:



Risulta evidente che ogni cifra appare più o meno equiprobabilmente, quindi forse in futuro ci saranno risvolti reali sull'utilizzo di una costante come il Pi greco per la codifica di file.

6 Bibliografia

[1] H. D. Bailey, The BBP Algorithm for Pi, 2006;

[2] D. Knuth, The Art of Computer Programming Vol. 2, 2009;